ckb-daemon

Generated by Doxygen 1.8.14

Contents

Chapter 1

Todo List

Global _usbsend (usbdevice ∗kb, const uchar ∗messages, int count, const char ∗file, int line)

A lot of different conditions are combined in this code. Don't think, it is good in every combination...

Check whether this is the same in the macOS variant. It is not dramatic, but if errors occur, it can certainly irritate the devices completely if they receive incomplete data streams. Do we have errors with the messages "Wrote YY bytes (expected 64)" in the system logs? If not, we do not need to look any further.

Global closeusb (usbdevice ∗kb)

What is not yet comprehensible is the call to updateconnected() BEFORE os_closeusb(). Should that be in the other sequence? Or is updateconnected() not displaying the connected usb devices, but the representation which uinput devices are loaded? Questions about questions ...

Global devmain (usbdevice ∗kb)

Hope to find the need for dmutex usage later.
Should this function be declared as pthread_t∗ function, because of the defintion of pthread-create? But void∗ works also...

readcmd() gets a **line**, not **lines**. Have a look on that later.
Is the condition IS_CONNECTED valid? What functions change the condititon for the macro?

Global get_vtable (short vendor, short product)

Is the last point really a good decision and always correct?

Global os_inputmain (void ∗context)

This function is a collection of many tasks. It should be divided into several sub-functions for the sake of greater convenience:

Global os_resetusb (usbdevice ∗kb, const char ∗file, int line)

it seems that no one wants to try the reset again. But I'v seen it somewhere...

Global os_setupusb (usbdevice ∗kb)

in these modules a pullrequest is outstanding

Global os_usbsend (usbdevice ∗kb, const uchar ∗out_msg, int is_recv, const char ∗file, int line)

Since the handling of endpoints has already led to problems elsewhere, this implementation is extremely hardware-dependent and critical!
Eg. the new keyboard K95PLATINUMRGB has a version number significantly less than 2.0 - will it run with this implementation?

Global product_str (short product)

There are macros defined in usb.h to detect all the combinations below. the only difference is the parameter: The macros need the *kb∗*, product_str() needs the *product ID*

Global revertusb (usbdevice ∗kb)

Why is this useful? Are there problems seen with deactivating a device with older fw-version??? Why isn't this an error indicating reason and we return success (0)?

The return value of nk95cmd() is ignored (but sending the ioctl may produce an error and _nk95_cmd will indicate this), instead revertusb() returns success in any case.

Global udevthread

These two thread vasriables seem to be unused: usbtread, udevthread

Global udevthread

These two thread vasriables seem to be unused: usbtread, udevthread

Global usb_add_device (struct udev_device ∗dev)

So why the hell not a transformation between the string and the short presentation? Lets check if the string representation is used elsewhere.

Global usb_tryreset (usbdevice ∗kb)

Why does usb_tryreset() hide the information returned from resetusb()? Isn't it needed by the callers?

Global usbmain ()

Why isn't missing of uinput a fatal error?

lae. here the work has to go on...

Global usbmutex

We should have a look why this mutex is never used.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

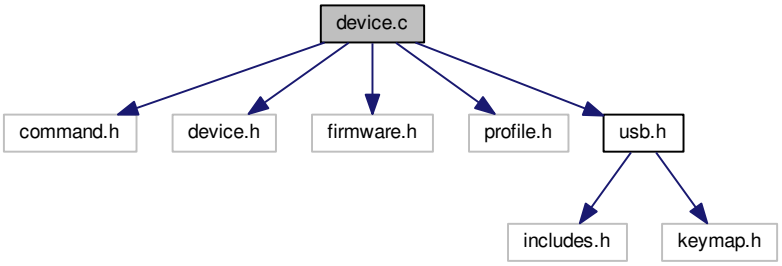
Chapter 3

File Documentation

3.1 device.c File Reference

```
#include "command.h"
#include "device.h"
#include "firmware.h"
#include "profile.h"
#include "usb.h"
```

Include dependency graph for device.c:



Functions

- int _start_dev (usbdevice *kb, int makeactive)
- int start_dev (usbdevice *kb, int makeactive)

Variables

- int hwload_mode = 1
- usbdevice keyboard [9]

  hwload_mode = 1 means read hardware once. should be enough
- pthread_mutex_t devlistmutex = PTHREAD_MUTEX_INITIALIZER

  remember all usb devices. Needed for closeusb().
- pthread_mutex_t devmutex [9] = { [0 ... 9 -1] = PTHREAD_MUTEX_INITIALIZER }
- pthread_mutex_t inputmutex [9] = { [0 ... 9 -1] = PTHREAD_MUTEX_INITIALIZER }

3.1.1 Function Documentation

3.1.1.1 _start_dev()

```
int _start_dev (
            usbdevice * kb,
            int makeactive )
```

_start_dev get fw-info and pollrate; if available, install new firmware; get all hardware profiles

Parameters

kb	the normal kb pointer to the usbdevice. Is also valid for mice.
makeactive	if set to 1, activate the device via setactive()

Returns

0 if success, other else

- This hacker code is tricky in mutliple aspects. What it means is:
  if hwload_mode == 0: just set pollrate to 0 and clear features in the bottom lines of the if-block.
  if hwload_mode == 1: if the device has FEAT_HWLOAD active, call getfwversion(). If it returns true, there was an error while detecting fw-version. Put error message, reset FEAT_HWLOAD and finalize as above.
  if hwload_mode == 2: if the device has FEAT_HWLOAD active, call getfwversion(). If it returns true, there was an error while detecting fw-version. Put error message and return directly from function with error.
  Why do not you just write it down?
- Now check if device needs a firmware update. If so, set it up and leave the function without error.
- Device needs a firmware update. Finish setting up but don't do anything.
- Load profile from device if the hw-pointer is not set yet and hw-loading is possible and allowed.
  return error if mode == 2 (load always) and loading got an error. Else reset HWLOAD feature, because hwload must be 1.

  That is real Horror code.

Definition at line 22 of file device.c.

References hwload_mode.

Referenced by start_dev().

```
22                                          {
23     // Get the firmware version from the device
24     if(kb->pollrate == 0){
32         if(!hwload_mode || (HAS_FEATURES(kb, FEAT_HWLOAD) && getfwversion(kb))){
33             if(hwload_mode == 2)
34                 // hwload=always. Report setup failure.
35                 return -1;
36             else if(hwload_mode){
37                 // hwload=once. Log failure, prevent trying again, and continue.
38                 ckb_warn("Unable to load firmware version/poll rate\n");
39                 kb->features &= ~FEAT_HWLOAD;
40             }
```

```
41              kb->pollrate = 0;
42              kb->features &= ~(FEAT_POLLRATE | FEAT_ADJRATE);
43              if(kb->fwversion == 0)
44                  kb->features &= ~(FEAT_FWVERSION | FEAT_FWUPDATE);
45          }
46      }
51      if(NEEDS_FW_UPDATE(kb)){
53          ckb_info("Device needs a firmware update. Please issue a fwupdate command.\n");
54          kb->features = FEAT_RGB | FEAT_FWVERSION | FEAT_FWUPDATE;
55          kb->active = 1;
56          return 0;
57      }
63      if(!kb->hw && hwload_mode && HAS_FEATURES(kb, FEAT_HWLOAD)){
64          if(hwloadprofile(kb, 1)){
65              if(hwload_mode == 2)
66                  return -1;
67              ckb_warn("Unable to load hardware profile\n");
68              kb->features &= ~FEAT_HWLOAD;
69          }
70      }
71      // Active software mode if requested
72      if(makeactive)
73          return setactive(kb, 1);
74      return 0;
75  }
```

Here is the caller graph for this function:

3.1.1.2 start_dev()

```
int start_dev (
              usbdevice * kb,
              int makeactive )
```

Definition at line 77 of file device.c.

References _start_dev(), and USB_DELAY_DEFAULT.

```
77                                          {
78      // Force USB interval to 10ms during initial setup phase; return to nominal 5ms after setup completes.
79      kb->usbdelay = 10;
80      int res = _start_dev(kb, makeactive);
81      kb->usbdelay = USB_DELAY_DEFAULT;
82      return res;
83  }
```

Here is the call graph for this function:

3.1.2 Variable Documentation

3.1.2.1 devlistmutex

```
pthread_mutex_t devlistmutex = PTHREAD_MUTEX_INITIALIZER
```

Definition at line 11 of file device.c.

3.1.2.2 devmutex

```
pthread_mutex_t devmutex[9] = { [0 ...  9 -1] = PTHREAD_MUTEX_INITIALIZER }
```

Definition at line 12 of file device.c.

Referenced by usb_rm_device().

3.1.2.3 hwload_mode

```
int hwload_mode = 1
```

Definition at line 7 of file device.c.

Referenced by _start_dev(), _usbrecv(), _usbsend(), and usb_tryreset().

3.1.2.4 inputmutex

```
pthread_mutex_t inputmutex[9] = { [0 ...  9 -1] = PTHREAD_MUTEX_INITIALIZER }
```

Definition at line 13 of file device.c.

3.1.2.5 keyboard

```
usbdevice keyboard[9]
```

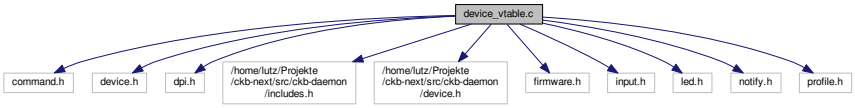
Definition at line 10 of file device.c.

Referenced by _setupusb(), closeusb(), os_closeusb(), os_inputmain(), os_setupusb(), usb_rm_device(), and usbadd().

3.2 device_vtable.c File Reference

```
#include "command.h"
#include "device.h"
#include "dpi.h"
#include "firmware.h"
#include "input.h"
#include "led.h"
#include "notify.h"
#include "profile.h"
```

Include dependency graph for device_vtable.c:



Functions

- static void cmd_none (usbdevice ∗kb, usbmode ∗dummy1, int dummy2, int dummy3, const char ∗dummy4)
- static int cmd_io_none (usbdevice ∗kb, usbmode ∗dummy1, int dummy2, int dummy3, const char ∗dummy4)
- static void cmd_macro_none (usbdevice ∗kb, usbmode ∗dummy1, int dummy2, const char ∗dummy3, const char ∗dummy4)
- static int loadprofile_none (usbdevice ∗kb)
- static void int1_void_none (usbdevice ∗kb, int dummy)
- static int int1_int_none (usbdevice ∗kb, int dummy)

Variables

- const devcmd vtable_keyboard
- const devcmd vtable_keyboard_nonrgb
- const devcmd vtable_mouse

3.2.1 Function Documentation

3.2.1.1 cmd_io_none()

```
static int cmd_io_none (
            usbdevice * kb,
            usbmode * dummy1,
            int dummy2,
            int dummy3,
            const char * dummy4 )  [static]
```

Definition at line 13 of file device_vtable.c.

```
13                                                                              {
14     return 0;
15 }
```

3.2.1.2 cmd_macro_none()

```
static void cmd_macro_none (
             usbdevice * kb,
             usbmode * dummy1,
             int dummy2,
             const char * dummy3,
             const char * dummy4 )  [static]
```

Definition at line 16 of file device_vtable.c.

```
16
        {
17 }
```

3.2.1.3 cmd_none()

```
static void cmd_none (
             usbdevice * kb,
             usbmode * dummy1,
             int dummy2,
             int dummy3,
             const char * dummy4 )  [static]
```

Definition at line 11 of file device_vtable.c.

```
11                                                                                {
12 }
```

3.2.1.4 int1_int_none()

```
static int int1_int_none (
             usbdevice * kb,
             int dummy )  [static]
```

Definition at line 23 of file device_vtable.c.

```
23                                              {
24     return 0;
25 }
```

3.2.1.5 int1_void_none()

```
static void int1_void_none (
             usbdevice * kb,
             int dummy )  [static]
```

Definition at line 21 of file device_vtable.c.

```
21                                                  {
22 }
```

3.2.1.6 loadprofile_none()

```
static int loadprofile_none (
             usbdevice * kb )  [static]
```

Definition at line 18 of file device_vtable.c.

```
18                                          {
19     return 0;
20 }
```

3.2.2 Variable Documentation

3.2.2.1 vtable_keyboard

```
const devcmd vtable_keyboard
```

Definition at line 28 of file device_vtable.c.

Referenced by get_vtable().

3.2.2.2 vtable_keyboard_nonrgb

```
const devcmd vtable_keyboard_nonrgb
```

Definition at line 75 of file device_vtable.c.

Referenced by get_vtable().

3.2.2.3 vtable_mouse

```
const devcmd vtable_mouse
```

Definition at line 122 of file device_vtable.c.

Referenced by get_vtable().

3.3 usb.c File Reference

```
#include "command.h"
#include "device.h"
#include "devnode.h"
#include "firmware.h"
#include "input.h"
#include "led.h"
#include "notify.h"
#include "profile.h"
```

Include dependency graph for usb.c:

Functions

- const char * vendor_str (short vendor)
  brief .
- const char * product_str (short product)
  brief .
- static const devcmd * get_vtable (short vendor, short product)
  brief .
- static void * devmain (usbdevice *kb)
  brief .
- static void * _setupusb (void *context)
  brief .
- void setupusb (usbdevice *kb)
- int revertusb (usbdevice *kb)
- int _resetusb (usbdevice *kb, const char *file, int line)
- int usb_tryreset (usbdevice *kb)
- int _usbsend (usbdevice *kb, const uchar *messages, int count, const char *file, int line)
- int _usbrecv (usbdevice *kb, const uchar *out_msg, uchar *in_msg, const char *file, int line)
- int closeusb (usbdevice *kb)

Variables

- pthread_mutex_t usbmutex = PTHREAD_MUTEX_INITIALIZER
  brief .
- volatile int reset_stop = 0
  brief .
- int features_mask = -1
  brief .
- int hwload_mode

3.3.1 Function Documentation

3.3.1.1 _resetusb()

```
int _resetusb (
            usbdevice * kb,
            const char * file,
            int line )
```

_resetusb Reset a USB device.

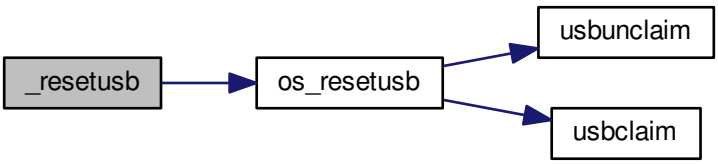
First reset the device via os_resetusb() after a long delay (it may send something to the host). If this worked (retval == 0), give the device another long delay Then perform the initialization via the device specific start() function entry in kb->vtable and if this is successful also, return the result of the device depenten updatergb() with force=true.

Definition at line 429 of file usb.c.

References DELAY_LONG, and os_resetusb().

```
429                                                         {
430     // Perform a USB reset
431     DELAY_LONG(kb);
432     int res = os_resetusb(kb, file, line);
433     if(res)
434         return res;
435     DELAY_LONG(kb);
436     // Re-initialize the device
437     if(kb->vtable->start(kb, kb->active) != 0)
438         return -1;
439     if(kb->vtable->updatergb(kb, 1) != 0)
440         return -1;
441     return 0;
442 }
```

Here is the call graph for this function:



3.3.1.2 _setupusb()

```
static void* _setupusb (
            void * context )  [static]
```

_setupusb A horrible function for setting up an usb device

Parameters

context	As _setupusb() is called as a new thread, the kb∗ is transferred as void∗

Returns

a ptread_t∗ 0, here casted as void∗. Retval is always null

The basic structure of the function is somewhat habituated. It is more like an assembler routine than a structured program. This is not really bad, but just getting used to.

After every action, which can be practically fault-prone, the routine goes into the same error handling: It goes via goto to one of two exit labels. The difference is whether or not an unlock has to be performed on the imutex variable. In both cases, closeusb() is called, then an unlock is performed on the dmutex.

The only case where this error handling is not performed is the correct return of the call to devmain(). Here simply the return value of devmain() is passed to the caller.

In either case, the routine terminates with a void∗ 0 because either devmain() has returned constant null or the routine itself returns zero.

The basic idea of this routine is the following:
First some initialization of kb standard structured and local vars is done.

- **kb** is set to the pointer given from start environment
- local vars **vendor** and **product** are set to the values from the corresponding fields of kb
- local var **vt and** the **kb-**>**vtable** are both set to the retval of *get_vtable()*
- **kb-**>**features** are set depending on the type of hardware connected:
  - set either to standard non rgb (all common flags like binding, notify, FW, hardware-loading etc) or in case of RGB-device set to standard + RGB, pollrate-change and fw-update
  - exclude all features which are disabled via feature_mask (set by daemon CLI parameters)
  - if it is a mouse, add adjust-rate
  - if it is a monochrome device, set the flag for RGB-protocol, but single color
- the standard delay time is initialized in kb->usbdelay
- A fixed 100ms wait is the start. **Although the DELAY_LONG macro is given a parameter, it is ignored. Occasionally refactor it.**
- The first relevant point is the operating system-specific opening of the interface in os_setupusb(). As a result, some parameters should be set in kb (name, serial, fwversion, epcount = number of usb endpoints), and all endpoints should be claimed with usbclaim(). Claiming is the only point where os_setupusb() can produce an error (-1, otherwise 0).
- The following two statements deal with possible errors when setting the kb values in the current routine: If the version or the name was not read correctly, they are set to default values:
  - serial is set to "<vendor>: <product> -NoID"
  - the name is set to "<vendor> <product>".
- Then the user level input subsystem is activated via os_openinput(). There are two file descriptors, one for the mouse and one for the keyboard. **As mentioned in structures.h, not the just opened FD numbers are stored under kb-**>**uinput_kb or kb-**>**uinput_mouse, but the values increased by 1!** The reason is, if the open fails or not open has been done until now, that struct member is set to 0, not to -1 or other negative value. So all usage of this kb->handle must be something like "`kb->handle - 1`", as you can find it in the code.

- The next action is to create a separate thread, which gets as parameter kb and starts with os_inputmain(). The thread is immediately detached so that it can return its resource completely independently if it should terminate.
- The same happens with os_setupindicators(), which initially initializes all LED variables in kb to off and then starts the _ledthread() thread with kb as parameter and then detaches it. Here again only the generation of the thread can fail.
- Via an entry in the vable (allocprofile, identical for all three vtable types), allocprofile() is called in profile.c. With a valid parameter kb, a usbprofile structure is allocated and stored as a kb->profile. Then initmode() is called for each of the initializable modes (MODE_COUNT, currently 6). This procedure creates the memory space for the mode information, initializes the range to 0, and then sets the light.forceupdate and dpi.forceupdate to true. This forces an update later in the initialization of the device.
  The first mode is set as the current mode and two force flags are set (this seems to be mode-intersecting flags for light and update).

  Warning

  There is no error handling for the allocprofile() and initmode() procedures. However, since they allocate storage areas, the subsequent assignments and initializations can run in a SEGV.
- Not completely understandable is why now via the vtable the function updateindicators() is called. But this actually happens in the just started thread _ledthread(). Either the initialization is wrong und must done here with force or the overview is lost, what happens when...
  Regardless: For a mouse nothing happens here, for a keyboard updateindicators_kb() is called via the entry in kb->vtable. The first parameter is kb again, the second is constant 1 (means force = true). This causes the LED status to be sent after a 5ms delay via os_sendindicators() (ioctl with a `usbdevfs_ctrltransfer`).
  The notification is sent to all currently open notification channels then.
  Setupindicators() and with it updateindicators_kb() can fail.
- From this point - if an error is detected - the error label is addressed by goto statement, which first performs an unlock on the imutex. This is interesting because the next statement is exactly this: An unlock on the imutex.
- Via vtable the *kb->start()* function is called next. This is the same for a mouse and an RGB keyboard: start_dev(), for a non RGB keyboard it is start_kb_nrgb().
  First parameter is as always kb, second is 0 (makeactive = false).
  - In start_kb_nrgb() set the keyboard into a so-called software mode (NK95_HWOFF) via ioctl with `usbdevfs_ctrltransfer` in function _nk95cmd(), which will in turn is called via macro nk95cmd() via start_kb_nrgb().
    Then two dummy values (active and pollrate) are set in the kb structure and ready.
  - start_dev() does a bit more - because this function is for both mouse and keyboard. start_dev() calls - after setting an extended timeout parameter - _start_dev(). Both are located in device.c.
  - First, _start_dev() attempts to determine the firmware version of the device, but only if two conditions are met: hwload-mode is not null (then hw-loading is disabled) and the device has the FEAT_HWLOAD feature. Then the firmware and the poll rate are fetched via getfwversion().
    If hwload_mode is set to "load only once" (==1), then the HWLOAD feature is masked, so that no further reading can take place.
  - Now check if device needs a firmware update. If so, set it up and leave the function without error.
  - Else load the hardware profile from device if the hw-pointer is not set and hw-loading is possible and allowed.
    Return error if mode == 2 (load always) and loading got an error. Else mask the HWLOAD feature, because hwload must be 1 and the error couold be a repeated hw-reading.
    Puh, that is real Horror code. It seems to be not faulty, but completely unreadable.
  - Finally, the second parameter of _startdev() is used to check whether the device is to be activated. Depending on the parameter, the active or the idle-member in the correspondig vtable is called. These are device-dependent again:

Device	active	idle
RGB Keyboard	cmd_active_kb() means: start the device with a lot of kb-specific initializers (software controlled mode)	cmd_idle_kb() set the device with a lot of kb-specific initializers into the hardware controlled mode)
non RGB Keyboard	cmd_io_none() means: Do nothing	cmd_io_none() means: Do nothing
Mouse	cmd_active_mouse() similar to cmd_active_kb()	cmd_idle_mouse similar to cmd_↩ idle_kb()

- If either *start()* succeeded or the next following usb_tryreset(), it goes on, otherwise again a hard abort occurs.
- Next, go to mkdevpath(). After securing the EUID (effective UID) especially for macOS, work starts really in _mkdevpath(). Create - no matter how many devices were registered - either the ckb0/ files **version**, **pid** and **connected** or the **cmd** command fifo, the first notification fifo **notify0**, **model** and **serial** as well as the **features** of the device and the **pollrate**.
- If all this is done and no error has occurred, a debug info is printed ("Setup finished for ckbx") updateconnected() writes the new device into the text file under ckb0/ and devmain() is called.

devmain()'s return value is returned by _setupusb() when we terminate.

- The remaining code lines are the two exit labels as described above

Definition at line 217 of file usb.c.

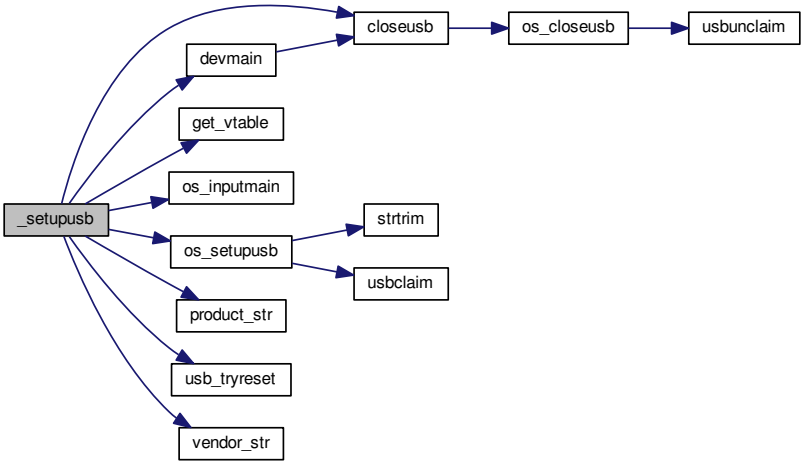
References closeusb(), DELAY_LONG, devmain(), features_mask, get_vtable(), IS_MONOCHROME, IS_MOU↩ SE, IS_RGB, keyboard, os_inputmain(), os_setupusb(), product_str(), USB_DELAY_DEFAULT, usb_tryreset(), and vendor_str().

Referenced by setupusb().

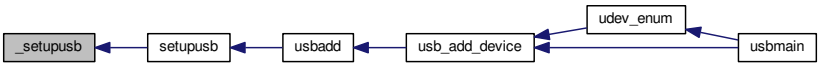
```
217                                     {
230     usbdevice* kb = context;
231     // Set standard fields
232     short vendor = kb->vendor, product = kb->product;
233     const devcmd* vt = kb->vtable = get_vtable(vendor, product);
234     kb->features = (IS_RGB(vendor, product) ? FEAT_STD_RGB : FEAT_STD_NRGB) &
    features_mask;
235     if(IS_MOUSE(vendor, product)) kb->features |= FEAT_ADJRATE;
236     if(IS_MONOCHROME(vendor, product)) kb->features |= FEAT_MONOCHROME;
237     kb->usbdelay = USB_DELAY_DEFAULT;
238
239     // Perform OS-specific setup
243     DELAY_LONG(kb);
244
250     if(os_setupusb(kb))
251         goto fail;
252
258     // Make up a device name and serial if they weren't assigned
259     if(!kb->serial[0])
260         snprintf(kb->serial, SERIAL_LEN, "%04x:%04x-NoID", kb->vendor, kb->product);
261     if(!kb->name[0])
262         snprintf(kb->name, KB_NAME_LEN, "%s %s", vendor_str(kb->vendor),
    product_str(kb->product));
263
264     // Set up an input device for key events
272     if(os_inputopen(kb))
273         goto fail;
277     if(pthread_create(&kb->inputthread, 0, os_inputmain, kb))
278         goto fail;
279     pthread_detach(kb->inputthread);
285     if(os_setupindicators(kb))
286         goto fail;
287
288     // Set up device
301     vt->allocprofile(kb);
312     vt->updateindicators(kb, 1);
317     pthread_mutex_unlock(imutex(kb));
351     if(vt->start(kb, 0) && usb_tryreset(kb))
```

```
352         goto fail_noinput;
358     // Make /dev path
359     if(mkdevpath(kb))
360         goto fail_noinput;
366     // Finished. Enter main loop
367     int index = INDEX_OF(kb, keyboard);
368     ckb_info("Setup finished for %s%d\n", devpath, index);
369     updateconnected();
372     return devmain(kb);
375     fail:
376     pthread_mutex_unlock(imutex(kb));
377     fail_noinput:
378     closeusb(kb);
379     pthread_mutex_unlock(dmutex(kb));
380     return 0;
381 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.1.3 _usbrecv()

```
int _usbrecv (
            usbdevice * kb,
            const uchar * out_msg,
            uchar * in_msg,
            const char * file,
            int line )
```

_usbrecv Request data from a USB device by first sending an output packet and then reading the response.

To fully understand this, you need to know about usb: All control is at the usb host (the CPU). If the device wants to communicate something to the host, it must wait for the host to ask. The usb protocol defines the cycles and periods in which actions are to be taken.

So in order to receive a data packet from the device, the host must first send a send request.
This is done by _usbrecv() in the first block by sending the MSG_SIZE large data block from **out_msg** via os↩ _usbsend() as it is a machine depending implementation. The usb target device is as always determined over kb.

For os_usbsend() to know that it is a receive request, the **is_recv** parameter is set to true (1). With this, os_usbsend () generates a control package for the hardware, not a data packet.

If sending of the control package is not successful, a maximum of 5 times the transmission is repeated (including the first attempt). If a non-cancelable error is signaled or the drive is stopped via reset_stop, _usbrecv() immediately returns 0.

After this, the function waits for the requested response from the device using os_usbrecv ().

os_usbrecv() returns 0, -1 or something else.
Zero signals a serious error which is not treatable and _usbrecv() also returns 0.
-1 means that it is a treatable error - a timeout for example - and therefore the next transfer attempt is started after a long pause (DELAY_LONG) if not reset_stop or the wrong hwload_mode require a termination with a return value of 0.

After 5 attempts, _usbrecv () returns and returns 0 as well as an error message.

When data is received, the number of received bytes is returned. This should always be MSG_SIZE, but os_↩ usbrecv() can also return less. It should not be more, because then there would be an unhandled buffer overflow, but it could be less. This would be signaled in os_usbrecv () with a message.

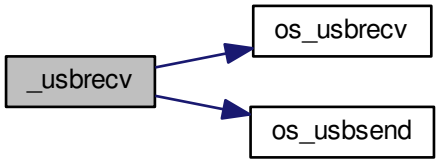
The buffers behind **out_msg** and **in_msg** are MSG_SIZE at least (currently 64 Bytes). More is ok but useless, less brings unpredictable behavior.

Definition at line 602 of file usb.c.

References DELAY_LONG, DELAY_MEDIUM, DELAY_SHORT, hwload_mode, os_usbrecv(), os_usbsend(), and reset_stop.

```
602                                                                                                 {
603     // Try a maximum of 3 times
604     for(int try = 0; try < 5; try++){
605         // Send the output message
606         DELAY_SHORT(kb);
607         int res = os_usbsend(kb, out_msg, 1, file, line);
608         if(res == 0)
609             return 0;
610         else if(res == -1){
611             // Retry on temporary failure
612             if(reset_stop)
613                 return 0;
614             DELAY_LONG(kb);
615             continue;
616         }
617         // Wait for the response
618         DELAY_MEDIUM(kb);
619         res = os_usbrecv(kb, in_msg, file, line);
620         if(res == 0)
621             return 0;
622         else if(res != -1)
623             return res;
624         if(reset_stop || hwload_mode != 2)
625             return 0;
626         DELAY_LONG(kb);
627     }
628     // Give up
629     ckb_err_fn("Too many send/recv failures. Dropping.\n", file, line);
630     return 0;
631 }
```

Here is the call graph for this function:



3.3.1.4 _usbsend()

```
int _usbsend (
              usbdevice * kb,
              const uchar * messages,
              int count,
              const char * file,
              int line )
```

_usbsend send a logical message completely to the given device

Todo A lot of different conditions are combined in this code. Don't think, it is good in every combination...

The main task of _usbsend () is to transfer the complete logical message from the buffer beginning with *messages* to **count** ∗ **MSG_SIZE**.
According to usb 2.0 specification, a USB transmits a maximum of 64 byte user data packets. For the transmission of longer messages we need a segmentation. And that is exactly what happens here.

The message is given one by one to os_usbsend() in MSG_SIZE (= 64) byte large bites.

Attention

This means that the buffer given as argument must be n ∗ MSG_SIZE Byte long.

An essential constant parameter which is relevant for os_usbsend() only is is_recv = 0, which means sending.

Now it gets a little complicated again:

- If os_usbsend() returns 0, only zero bytes could be sent in one of the packets, or it was an error (-1 from the systemcall), but not a timeout. How many Bytes were sent in total from earlier calls does not seem to matter, _usbsend() returns a total of 0.
- Returns os_usbsend() -1, first check if **reset_stop** is set globally or (incomprehensible) hwload_mode is not set to "always". In either case, _usbsend() returns 0, otherwise it is assumed to be a temporary transfer error and it simply retransmits the physical packet after a long delay.

- If the return value of os_usbsend() was neither 0 nor -1, it specifies the numer of bytes transferred.
  Here is an information hiding conflict with os_usbsend() (at least in the Linux version):
  If os_usbsend() can not transfer the entire packet, errors are thrown and the number of bytes sent is returned. _usbsend() interprets this as well and remembers the total number of bytes transferred in the local variable **total_sent**. Subsequently, however, transmission is continued with the next complete MSG_SIZE block and not with the first of the possibly missing bytes.

  Todo Check whether this is the same in the macOS variant. It is not dramatic, but if errors occur, it can certainly irritate the devices completely if they receive incomplete data streams. Do we have errors with the messages "Wrote YY bytes (expected 64)" in the system logs? If not, we do not need to look any further.

When the last packet is transferred, _usbsend() returns the effectively counted set of bytes (from **total_sent**). This at least gives the caller the opportunity to check whether something has been lost in the middle.

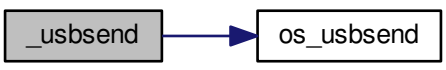
A bit strange is the structure of the program: Handling the **count** MSG_SIZE blocks to be transferred is done in the outer for (...) loop. Repeating the transfer with a treatable error is managed by the inner while(1) loop.
This must be considered when reading the code; The "break" on successful block transfer leaves the inner while, not the for (...).

Definition at line 535 of file usb.c.

References DELAY_LONG, DELAY_SHORT, hwload_mode, os_usbsend(), and reset_stop.

```
535                                                                                                 {
536     int total_sent = 0;
537     for(int i = 0; i < count; i++){
538         // Send each message via the OS function
539         while(1){
540             DELAY_SHORT(kb);
541             int res = os_usbsend(kb, messages + i * MSG_SIZE, 0, file, line);
542             if(res == 0)
543                 return 0;
544             else if(res != -1){
545                 total_sent += res;
546                 break;
547             }
548             // Stop immediately if the program is shutting down or hardware load is set to tryonce
549             if(reset_stop || hwload_mode != 2)
550                 return 0;
551             // Retry as long as the result is temporary failure
552             DELAY_LONG(kb);
553         }
554     }
555     return total_sent;
556 }
```

Here is the call graph for this function:



3.3.1.5 closeusb()

```
int closeusb (
            usbdevice * kb )
```

closeusb Close a USB device and remove device entry.

An imutex lock ensures first of all, that no communication is currently running from the viewpoint of the driver to the user input device (ie the virtual driver with which characters or mouse movements are sent from the daemon to the operating system as inputs).

If the **kb** has an acceptable value = 0, the index of the device is looked for and with this index os_inputclose() is called. After this no more characters can be sent to the operating system.

Then the connection to the usb device is capped by os_closeusb().

Todo What is not yet comprehensible is the call to updateconnected() BEFORE os_closeusb(). Should that be in the other sequence? Or is updateconnected() not displaying the connected usb devices, but the representation which uinput devices are loaded? Questions about questions ...

If there is no valid **handle**, only updateconnected() is called. We are probably trying to disconnect a connection under construction. Not clear.

The cmd pipe as well as all open notify pipes are deleted via rmdevpath ().
This means that nothing can happen to the input path - so the device-specific imutex is unlocked again and remains unlocked.

Also the dmutex is unlocked now, but only to join the thread, which was originally taken under **kb->thread** (which started with _setupusb()) with pthread_join() again. Because of the closed devices that thread would have to quit sometime

See also

the hack note with rmdevpath())

As soon as the thread is caught, the dmutex is locked again, which is what I do not understand yet: What other thread can do usb communication now?
If the vtabel exists for the given kb (why not? It seems to have race conditions here!!), via the vtable the actually device-specific, but still everywhere identical freeprofile() is called. This frees areas that are no longer needed. Then the **usbdevice** structure in its array is set to zero completely.

Error handling is rather unusual in closeusb(); Everything works (no matter what the called functions return), and closeusb() always returns zero (success).

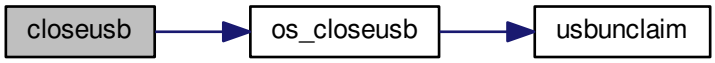
Definition at line 676 of file usb.c.

References keyboard, and os_closeusb().

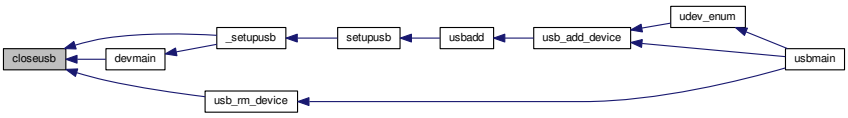
Referenced by _setupusb(), devmain(), and usb_rm_device().

```
676                             {
677     pthread_mutex_lock(imutex(kb));
678     if(kb->handle){
679         int index = INDEX_OF(kb, keyboard);
680         ckb_info("Disconnecting %s%d\n", devpath, index);
681         os_inputclose(kb);
682         updateconnected();
683         // Close USB device
684         os_closeusb(kb);
685     } else
686         updateconnected();
687     rmdevpath(kb);
688 
689     // Wait for thread to close
690     pthread_mutex_unlock(imutex(kb));
691     pthread_mutex_unlock(dmutex(kb));
692     pthread_join(kb->thread, 0);
693     pthread_mutex_lock(dmutex(kb));
694 
695     // Delete the profile and the control path
696     if(!kb->vtable)
697         return 0;
698     kb->vtable->freeprofile(kb);
699     memset(kb, 0, sizeof(usbdevice));
700     return 0;
701 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.1.6 devmain()

```
static void* devmain (
            usbdevice * kb )  [static]
```

devmain is called by _setupusb

Parameters

kb	the pointer to the device. Even if it has the name kb, it is valid also for a mouse (the whole driver seems to be implemented first for a keyboard).

Returns

always a nullptr

Synchronization

The syncing via mutexes is interesting:

1. *imutex* (the Input mutex)
   This one is locked in `setupusb()`. That function does only two things: Locking the mutex and trying to start a thread at `_setupusb()`. _setupusb() unlocks *imutex* after getting some buffers and initalizing internal structures from the indicators (this function often gets problems with error messages like "unable to read indicators" or "Timeout bla blubb").

   Warning

   have a look at *updateindicators()* later.
   if creating the thread is not successful, the *imutex* remains blocked. Have a look at setupusb() later.

2. *dmutex* (the Device mutex)
   This one is very interesting, because it is handled in devmain(). It seems that it is locked only in *_ledthread()*, which is a thread created in *os_setupindicators()*. os_setupindicators() again is called in *_setupusb()* long before calling devmain(). So this mutex is locked when we start the function as the old comment says.
   Before reading from the FIFO and direct afterwards an unlock..lock sequence is implemented here. Even if only the function readlines() should be surrounded by the unlock..lock, the variable definition of the line pointer is also included here. Not nice, but does not bother either. Probably the Unlock..lock is needed so that now another process can change the control structure *linectx* while we wait in readlines().

   Todo Hope to find the need for dmutex usage later.
   Should this function be declared as pthread_t* function, because of the defintion of pthread-create? But void* works also...

Attention

dmutex should still be locked when this is called

First a *readlines_ctx* buffer structure is initialized by `readlines_ctx_init()`.

After some setup functions, beginning in _setupusb() which has called devmain(), we read the command input-↩ Fifo designated to that device in an endless loop. This loop has two possible exits (plus reaction to signals, not mentioned here).

If the reading via readlines() is successful (we might have read multiple lines), the interpretation is done by readcmd() iff the connection to the device is still available (checked via IS_CONNECTED(kb)). This is true if the kb-structure has a handle and an event pointer both != Null). If not, the loop is left (the first exit point).

if nothing is in the line buffer (some magic interrupt?), continue in the endless while without any reaction.

Todo readcmd() gets a **line**, not **lines**. Have a look on that later.
Is the condition IS_CONNECTED valid? What functions change the condititon for the macro?

If interpretation and communication with the usb device got errors, they are signalled by readcmd() (non zero retcode). In this case the usb device is closed via closeusb() and the endless loop is left (the second exit point).

After leaving the endless loop the readlines-ctx structure and its buffers are freed by readlines_ctx_free().

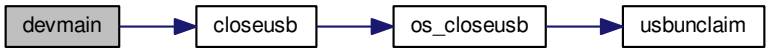
Definition at line 135 of file usb.c.

References closeusb().

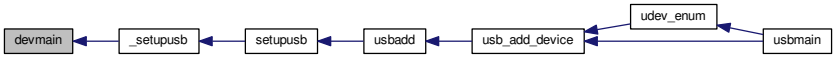
Referenced by _setupusb().

```
135                                 {
137     int kbfifo = kb->infifo - 1;
140     readlines_ctx linectx;
141     readlines_ctx_init(&linectx);
146     while(1){
153         pthread_mutex_unlock(dmutex(kb));
154         // Read from FIFO
155         const char* line;
156         int lines = readlines(kbfifo, linectx, &line);
157         pthread_mutex_lock(dmutex(kb));
158         // End thread when the handle is removed
159         if(!IS_CONNECTED(kb)) {
160             ckb_warn("devmain: not connected: %s\n", kb->name);
161             break;
162         }
166         if(lines){
169             if(readcmd(kb, line)){
175                 // USB transfer failed; destroy device
176                 ckb_warn("devmain: readcmd got error, %s\n", kb->name);
177                 closeusb(kb);
178                 break;
179             }
180         }
181     }
182     pthread_mutex_unlock(dmutex(kb));
185     readlines_ctx_free(linectx);
186     return 0;
187 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.1.7 get_vtable()

```
static const devcmd* get_vtable (
            short vendor,
            short product )  [static]
```

get_vtable returns the correct vtable pointer

Parameters

vendor	short usb vendor ID
product	short usb product ID

Returns

Depending on the type and model, the corresponding vtable pointer is returned (see below)

At present, we have three different vtables:

- `vtable_mouse` is used for all mouse types. This may be wrong with some newer mice?
- `vtable_keyboard` is used for all RGB Keyboards.
- `vtable_keyboard_nonrgb` for all the rest.

Todo Is the last point really a good decision and always correct?

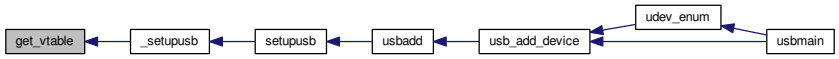
Definition at line 102 of file usb.c.

References IS_MOUSE, IS_RGB, vtable_keyboard, vtable_keyboard_nonrgb, and vtable_mouse.

Referenced by _setupusb().

```
102                                                          {
103     return IS_MOUSE(vendor, product) ? &vtable_mouse :
      IS_RGB(vendor, product) ? &vtable_keyboard : &
      vtable_keyboard_nonrgb;
104 }
```

Here is the caller graph for this function:



3.3.1.8 product_str()

```
const char* product_str (
            short product )
```

product_str returns a condensed view on what type of device we have.

At present, various models and their properties are known from corsair products. Some models differ in principle (mice and keyboards), others differ in the way they function (for example, RGB and non RGB), but they are very similar.

Here, only the first point is taken into consideration and we return a unified model string. If the model is not known with its number, *product_str* returns an empty string.

The model numbers and corresponding strings wwith the numbers in hex-string are defined in `usb.h`

At present, this function is used to initialize `kb->name` and to give information in debug strings.

Attention

The combinations below have to fit to the combinations in the macros mentioned above. So if you add a device with a new number, change both.

Todo There are macros defined in usb.h to detect all the combinations below. the only difference is the parameter: The macros need the *kb∗*, product_str() needs the *product ID*

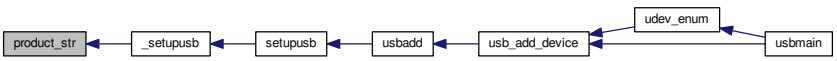
Definition at line 70 of file usb.c.

References P_HARPOON, P_K65, P_K65_LUX, P_K65_NRGB, P_K65_RFIRE, P_K70, P_K70_LUX, P_K70_↩ LUX_NRGB, P_K70_NRGB, P_K70_RFIRE, P_K70_RFIRE_NRGB, P_K95, P_K95_NRGB, P_K95_PLATINUM, P_M65, P_M65_PRO, P_SABRE_L, P_SABRE_N, P_SABRE_O, P_SABRE_O2, P_SCIMITAR, P_SCIMITAR_↩ PRO, P_STRAFE, and P_STRAFE_NRGB.

Referenced by _setupusb().

```
70                                      {
71     if(product == P_K95 || product == P_K95_NRGB || product ==
       P_K95_PLATINUM)
72         return "k95";
73     if(product == P_K70 || product == P_K70_NRGB || product ==
       P_K70_LUX || product == P_K70_LUX_NRGB || product ==
       P_K70_RFIRE || product == P_K70_RFIRE_NRGB)
74         return "k70";
75     if(product == P_K65 || product == P_K65_NRGB || product ==
       P_K65_LUX || product == P_K65_RFIRE)
76         return "k65";
77     if(product == P_STRAFE || product == P_STRAFE_NRGB)
78         return "strafe";
79     if(product == P_M65 || product == P_M65_PRO)
80         return "m65";
81     if(product == P_SABRE_O || product == P_SABRE_L || product ==
       P_SABRE_N || product == P_SABRE_O2 || product == P_HARPOON)
82         return "sabre";
83     if(product == P_SCIMITAR || product == P_SCIMITAR_PRO)
84         return "scimitar";
85     return "";
86 }
```

Here is the caller graph for this function:



3.3.1.9 revertusb()

```
int revertusb (
            usbdevice * kb )
```

revertusb sets a given device to inactive (hardware controlled) mode if not a fw-ugrade is indicated

First is checked, whether a firmware-upgrade is indicated for the device. If so, revertusb() returns 0.

Todo Why is this useful? Are there problems seen with deactivating a device with older fw-version??? Why isn't this an error indicating reason and we return success (0)?

Anyway, the following steps are similar to some other procs, dealing with low level usb handling:

- If we do not have an RGB device, a simple setting to Hardware-mode (NK95_HWON) is sent to the device via n95cmd().

  Todo The return value of nk95cmd() is ignored (but sending the ioctl may produce an error and _nk95_cmd will indicate this), instead revertusb() returns success in any case.

- If we have an RGB device, setactive() is called with second param active = false. That function will have a look on differences between keyboards and mice.
  More precisely setactive() is just a macro to call via the kb->vtable enties either the active() or the idle() function where the vtable points to. setactive() may return error indications. If so, revertusb() returns -1, otherwise 0 in any other case.

Definition at line 410 of file usb.c.

References NK95_HWON, and nk95cmd.

```
410                             {
411     if(NEEDS_FW_UPDATE(kb))
412         return 0;
413     if(!HAS_FEATURES(kb, FEAT_RGB)){
414         nk95cmd(kb, NK95_HWON);
415         return 0;
416     }
417     if(setactive(kb, 0))
418         return -1;
419     return 0;
420 }
```

3.3.1.10 setupusb()

```
void setupusb (
             usbdevice * kb )
```

setupusb starts a thread with kb as parameter and _setupusb() as entrypoint.

Set up a USB device after its handle is open. Spawns a new thread _setupusb() with standard parameter kb. dmutex must be locked prior to calling this function. The function will unlock it when finished. In kb->thread the thread id is mentioned, because closeusb() needs this info for joining that thread again.

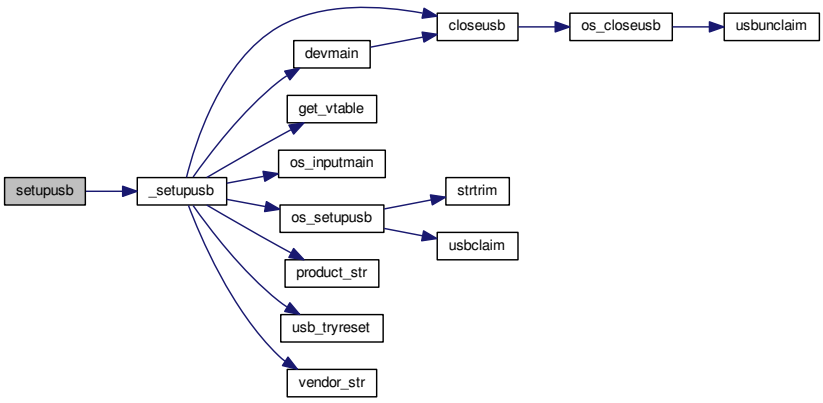
Definition at line 389 of file usb.c.

References _setupusb().

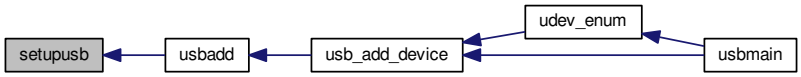
Referenced by usbadd().

```
389                             {
390     pthread_mutex_lock(imutex(kb));
391     if(pthread_create(&kb->thread, 0, _setupusb, kb))
392         ckb_err("Failed to create USB thread\n");
393 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.1.11 usb_tryreset()

```
int usb_tryreset (
            usbdevice * kb )
```

usb_tryreset does what the name means: Try to reset the usb via resetusb()

This function is called if an usb command ran into an error in case of one of the following two situations:

- When setting up a new usb device and the start() function got an error (

  See also

  _setupusb())

- If upgrading to a new firmware gets an error (

See also

cmd_fwupdate()).

The previous action which got the error will NOT be re-attempted.

In an endless loop usb_tryreset() tries to reset the given usb device via the macro resetusb().
This macro calls _resetusb() with debugging information.
_resetusb() sends a command via the operating system dependent function os_resetusb() and - if successful - reinitializes the device. os_resetusb() returns -2 to indicate a broken device and all structures should be removed for it.
In that case, the loop is terminated, an error message is produced and usb_tryreset() returns -1.

In case resetusb() has success, the endless loop is left via a return 0 (success).
If the return value from resetusb() is -1, the loop is continued with the next try.

If the global variable **reset_stop** is set directly when the function is called or after each try, usb_tryreset() stops working and returns -1.

Todo Why does usb_tryreset() hide the information returned from resetusb()? Isn't it needed by the callers?

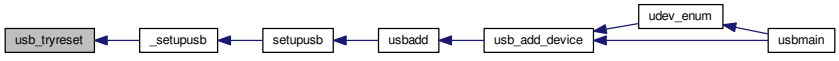
Definition at line 468 of file usb.c.

References hwload_mode, reset_stop, and resetusb.

Referenced by _setupusb().

```
468                                 {
469     if(reset_stop)
470         return -1;
471     ckb_info("Attempting reset...\n");
472     while(1){
473         int res = resetusb(kb);
474         if(!res){
475             ckb_info("Reset success\n");
476             return 0;
477         }
478         if(res == -2 || reset_stop)
479             break;
480     }
481     ckb_err("Reset failed. Disconnecting.\n");
482     return -1;
483 }
```

Here is the caller graph for this function:



3.3.1.12 vendor_str()

```
const char* vendor_str (
            short vendor )
```

uncomment the following Define to see USB packets sent to the device

vendor_str returns "corsair" iff the given *vendor* argument is equal to *V_CORSAIR* (0x1bc) else it returns ""

Attention

There is also a string defined V_CORSAIR_STR, which returns the device number as string in hex "1b1c".

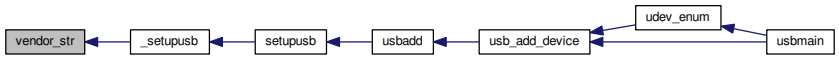
Definition at line 43 of file usb.c.

References V_CORSAIR.

Referenced by _setupusb().

```
43                                      {
44     if(vendor == V_CORSAIR)
45         return "corsair";
46     return "";
47 }
```

Here is the caller graph for this function:



3.3.2 Variable Documentation

3.3.2.1 features_mask

```
int features_mask = -1
```

features_mask Mask of features to exclude from all devices

That bit mask ist set to enable all (-1). When interpreting the input parameters, some of these bits can be cleared.
At the moment binding, notifying and mouse-acceleration can be disabled via command line.
Have a look at *main()* in main.c for details.

Definition at line 35 of file usb.c.

Referenced by _setupusb().

3.3.2.2 hwload_mode

```
int hwload_mode
```

Definition at line 7 of file device.c.

Referenced by _start_dev(), _usbrecv(), _usbsend(), and usb_tryreset().

3.3.2.3 reset_stop

```
volatile int reset_stop = 0
```

reset_stop is boolean: Reset stopper for when the program shuts down.

Is set only by *quit()* to true (1) to inform several usb_∗ functions to end their loops and tries.

Definition at line 25 of file usb.c.

Referenced by _usbrecv(), _usbsend(), and usb_tryreset().

3.3.2.4 usbmutex

```
pthread_mutex_t usbmutex = PTHREAD_MUTEX_INITIALIZER
```

usbmutex is a never referenced mutex!

Todo We should have a look why this mutex is never used.

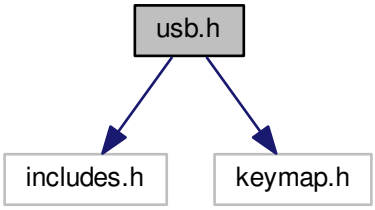
Definition at line 17 of file usb.c.

3.4 usb.h File Reference

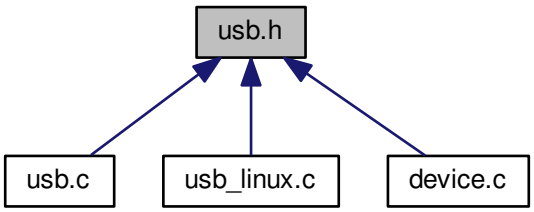
Definitions for using USB interface.

```
#include "includes.h"
#include "keymap.h"
```

Include dependency graph for usb.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define V_CORSAIR 0x1b1c

  For the following Defines please see "Detailed Description".
- #define V_CORSAIR_STR "1b1c"
- #define P_K65 0x1b17
- #define P_K65_STR "1b17"
- #define P_K65_NRGB 0x1b07
- #define P_K65_NRGB_STR "1b07"
- #define P_K65_LUX 0x1b37
- #define P_K65_LUX_STR "1b37"
- #define P_K65_RFIRE 0x1b39
- #define P_K65_RFIRE_STR "1b39"

- #define IS_K65(kb) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_K65 || (kb)->product == P_K65_NRGB || (kb)->product == P_K65_LUX || (kb)->product == P_K65_RFIRE))
- #define P_K70 0x1b13
- #define P_K70_STR "1b13"
- #define P_K70_NRGB 0x1b09
- #define P_K70_NRGB_STR "1b09"
- #define P_K70_LUX 0x1b33
- #define P_K70_LUX_STR "1b33"
- #define P_K70_LUX_NRGB 0x1b36
- #define P_K70_LUX_NRGB_STR "1b36"
- #define P_K70_RFIRE 0x1b38
- #define P_K70_RFIRE_STR "1b38"
- #define P_K70_RFIRE_NRGB 0x1b3a
- #define P_K70_RFIRE_NRGB_STR "1b3a"
- #define IS_K70(kb) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_K70 || (kb)->product == P_K70_NRGB || (kb)->product == P_K70_RFIRE || (kb)->product == P_K70_RFIRE_NRGB || (kb)->product == P_K70_LUX || (kb)->product == P_K70_LUX_NRGB))
- #define P_K95 0x1b11
- #define P_K95_STR "1b11"
- #define P_K95_NRGB 0x1b08
- #define P_K95_NRGB_STR "1b08"
- #define P_K95_PLATINUM 0x1b2d
- #define P_K95_PLATINUM_STR "1b2d"
- #define IS_K95(kb) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_K95 || (kb)->product == P_K95_NRGB || (kb)->product == P_K95_PLATINUM))
- #define P_STRAFE 0x1b20
- #define P_STRAFE_STR "1b20"
- #define P_STRAFE_NRGB 0x1b15
- #define P_STRAFE_NRGB_STR "1b15"
- #define IS_STRAFE(kb) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_STRAFE || (kb)->product == P_STRAFE_NRGB))
- #define P_M65 0x1b12
- #define P_M65_STR "1b12"
- #define P_M65_PRO 0x1b2e
- #define P_M65_PRO_STR "1b2e"
- #define IS_M65(kb) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_M65 || (kb)->product == P_M65_PRO))
- #define P_SABRE_O 0x1b14 /* optical */
- #define P_SABRE_O_STR "1b14"
- #define P_SABRE_L 0x1b19 /* laser */
- #define P_SABRE_L_STR "1b19"
- #define P_SABRE_N 0x1b2f /* new? */
- #define P_SABRE_N_STR "1b2f"
- #define P_SABRE_O2 0x1b32 /* Observed on a CH-9000111-EU model SABRE */
- #define P_SABRE_O2_STR "1b32"
- #define P_HARPOON 0x1b3c /* Harpoon test */
- #define P_HARPOON_STR "1b3c"
- #define IS_SABRE(kb) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_SABRE_O || (kb)->product == P_SABRE_L || (kb)->product == P_SABRE_N || (kb)->product == P_SABRE_O2 || (kb)->product == P_HARPOON))
- #define P_SCIMITAR 0x1b1e
- #define P_SCIMITAR_STR "1b1e"
- #define P_SCIMITAR_PRO 0x1b3e
- #define P_SCIMITAR_PRO_STR "1b3e"

- #define IS_SCIMITAR(kb) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_SCIMITAR || (kb)->product == P_SCIMITAR_PRO))
- #define IS_RGB(vendor, product) ((vendor) == (V_CORSAIR) && (product) != (P_K65_NRGB) && (product) != (P_K70_NRGB) && (product) != (P_K95_NRGB))

  RGB vs non-RGB test (note: non-RGB Strafe is still considered "RGB" in that it shares the same protocol. The difference is denoted with the "monochrome" feature).
- #define IS_MONOCHROME(vendor, product) ((vendor) == (V_CORSAIR) && (product) == (P_STRAFE_NRGB))

  *The difference between non RGB and monochrome is, that monochrome has lights, but just in one color. nonRGB has no lights. Change this if new **monochrome** devices are added.*
- #define IS_RGB_DEV(kb) IS_RGB((kb)->vendor, (kb)->product)

  For calling with a usbdevice, vendor and product are extracted and IS_RGB() is returned.*
- #define IS_MONOCHROME_DEV(kb) IS_MONOCHROME((kb)->vendor, (kb)->product)

  For calling with a usbdevice, vendor and product are extracted and IS_MONOCHROME() is returned.*
- #define IS_FULLRANGE(kb) (IS_RGB((kb)->vendor, (kb)->product) && (kb)->product != P_K65 && (kb)->product != P_K70 && (kb)->product != P_K95)

  Full color range (16.8M) vs partial color range (512)
- #define IS_MOUSE(vendor, product) ((vendor) == (V_CORSAIR) && ((product) == (P_M65) || (product) == (P_M65_PRO) || (product) == (P_SABRE_O) || (product) == (P_SABRE_L) || (product) == (P_SABRE_N) || (product) == (P_SCIMITAR) || (product) == (P_SCIMITAR_PRO) || (product) == (P_SABRE_O2)))

  Mouse vs keyboard test.
- #define IS_MOUSE_DEV(kb) IS_MOUSE((kb)->vendor, (kb)->product)

  For calling with a usbdevice, vendor and product are extracted and IS_MOUSE() is returned.*
- #define DELAY_SHORT(kb) usleep((int)(kb)->usbdelay * 1000)

  USB delays for when the keyboards get picky about timing That was the original comment, but it is used anytime. The short delay is used before any send or receive.
- #define DELAY_MEDIUM(kb) usleep((int)(kb)->usbdelay * 10000)

  the medium delay is used after sending a command before waiting for the answer.
- #define DELAY_LONG(kb) usleep(100000)

  The longest delay takes place where something went wrong (eg when resetting the device)
- #define USB_DELAY_DEFAULT 5

  *This constant is used to initialize **kb->usbdelay**. It is used in many places (see macros above) but often also overwritten to the fixed value of 10. Pure Hacker code.*
- #define resetusb(kb) _resetusb(kb, __FILE_NOPATH__, __LINE__)

  resetusb() is just a macro to call _resetusb() with debuggin constants (file, lineno)
- #define usbsend(kb, messages, count) _usbsend(kb, messages, count, __FILE_NOPATH__, __LINE__)

  usbsend macro is used to wrap _usbsend() with debugging information (file and lineno)
- #define usbrecv(kb, out_msg, in_msg) _usbrecv(kb, out_msg, in_msg, __FILE_NOPATH__, __LINE__)

  usbrecv macro is used to wrap _usbrecv() with debugging information (file and lineno)
- #define nk95cmd(kb, command) _nk95cmd(kb, (command) >> 16 & 0xFF, (command) & 0xFFFF, __FILE_NOPATH__, __LINE__)

  nk95cmd() macro is used to wrap _nk95cmd() with debugging information (file and lineno). the command structure is different:
  Just the bits 23..16 are used as bits 7..0 for bRequest
  Bits 15..0 are used as wValue
- #define NK95_HWOFF 0x020030

  Hardware-specific commands for the K95 nonRGB,.
- #define NK95_HWON 0x020001

  Hardware playback on.
- #define NK95_M1 0x140001

  Switch to mode 1.
- #define NK95_M2 0x140002

  Switch to mode 2.
- #define NK95_M3 0x140003

  Switch to mode 3.

Functions

- const char * vendor_str (short vendor)

  uncomment the following Define to see USB packets sent to the device
- const char * product_str (short product)

  product_str returns a condensed view on what type of device we have.
- int usbmain ()

  Start the USB main loop. Returns program exit code when finished.
- void usbkill ()

  Stop the USB system.
- void setupusb (usbdevice *kb)

  setupusb starts a thread with kb as parameter and _setupusb() as entrypoint.
- int os_setupusb (usbdevice *kb)

  os_setupusb OS-specific setup for a specific usb device.
- void * os_inputmain (void *context)

  os_inputmain is run in a separate thread and will be detached from the main thread, so it needs to clean up its own resources.
- int revertusb (usbdevice *kb)

  revertusb sets a given device to inactive (hardware controlled) mode if not a fw-ugrade is indicated
- int closeusb (usbdevice *kb)

  closeusb Close a USB device and remove device entry.
- void os_closeusb (usbdevice *kb)

  os_closeusb unclaim it, destroy the udev device and clear data structures at kb
- int _resetusb (usbdevice *kb, const char *file, int line)

  _resetusb Reset a USB device.
- int os_resetusb (usbdevice *kb, const char *file, int line)

  os_resetusb is the os specific implementation for resetting usb
- int _usbsend (usbdevice *kb, const uchar *messages, int count, const char *file, int line)

  _usbsend send a logical message completely to the given device
- int _usbrecv (usbdevice *kb, const uchar *out_msg, uchar *in_msg, const char *file, int line)

  _usbrecv Request data from a USB device by first sending an output packet and then reading the response.
- int os_usbsend (usbdevice *kb, const uchar *out_msg, int is_recv, const char *file, int line)

  os_usbsend sends a data packet (MSG_SIZE = 64) Bytes long
- int os_usbrecv (usbdevice *kb, uchar *in_msg, const char *file, int line)

  os_usbrecv receives a max MSGSIZE long buffer from usb device
- void os_sendindicators (usbdevice *kb)

  os_sendindicators update the indicators for the special keys (Numlock, Capslock and what else?)
- int _nk95cmd (usbdevice *kb, uchar bRequest, ushort wValue, const char *file, int line)

  _nk95cmd If we control a non RGB keyboard, set the keyboard via ioctl with usbdevfs_ctrltransfer
- int usb_tryreset (usbdevice *kb)

  usb_tryreset does what the name means: Try to reset the usb via resetusb()

3.4.1 Detailed Description

Vendor/product codes

The list of defines in the first part of the file describes the various types of equipment from Corsair and summarizes them according to specific characteristics.
Each device type is described with two defines:

- On the one hand the device ID with which the device can be recognized on the USB as a short
- and on the other hand the same representation as a string, but without leading "0x".

First entry-pair is the Provider ID (vendorID) from Corsair.

Block No.	contains	Devices are bundled via
1	The first block contains the K65-like keyboards, regardless of their properties (RGB, ...).	In summary, they can be queried using the macro IS_K65().
2	the K70-like Keyboards with all their configuration types	summarized by IS_K70().
3	the K95 series keyboards	collected with the macro IS_K95().
4	strafe keyboards	IS_STRAFE()
5	M65 mice with and without RGB	IS_M65()
6	The SABRE and HARPOON mice. Maybe this will be divided int two different blocks later because of different nummber of special keys	IS_SABRE()
7	The Scimitar mouse devices	IS_SCIMITAR()

3.4.2 Macro Definition Documentation

3.4.2.1 DELAY_LONG

```
#define DELAY_LONG(
              kb ) usleep(100000)
```

Definition at line 153 of file usb.h.

Referenced by _resetusb(), _setupusb(), _usbrecv(), and _usbsend().

3.4.2.2 DELAY_MEDIUM

```
#define DELAY_MEDIUM(
              kb ) usleep((int)(kb)->usbdelay * 10000)
```

Definition at line 150 of file usb.h.

Referenced by _usbrecv().

3.4.2.3 DELAY_SHORT

```
#define DELAY_SHORT(
              kb ) usleep((int)(kb)->usbdelay * 1000)
```

Definition at line 147 of file usb.h.

Referenced by _usbrecv(), and _usbsend().

3.4.2.4 IS_FULLRANGE

```
#define IS_FULLRANGE(
              kb ) (IS_RGB((kb)->vendor, (kb)->product) && (kb)->product != P_K65 && (kb)->product
!= P_K70 && (kb)->product != P_K95)
```

Definition at line 136 of file usb.h.

3.4.2.5 IS_K65

```
#define IS_K65(
              kb ) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_K65 || (kb)->product ==
P_K65_NRGB || (kb)->product == P_K65_LUX || (kb)->product == P_K65_RFIRE))
```

Definition at line 49 of file usb.h.

3.4.2.6 IS_K70

```
#define IS_K70(
              kb ) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_K70 || (kb)->product ==
P_K70_NRGB || (kb)->product == P_K70_RFIRE || (kb)->product == P_K70_RFIRE_NRGB || (kb)->product
== P_K70_LUX || (kb)->product == P_K70_LUX_NRGB))
```

Definition at line 63 of file usb.h.

3.4.2.7 IS_K95

```
#define IS_K95(
              kb ) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_K95 || (kb)->product ==
P_K95_NRGB || (kb)->product == P_K95_PLATINUM))
```

Definition at line 71 of file usb.h.

3.4.2.8 IS_M65

```
#define IS_M65(
              kb ) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_M65 || (kb)->product ==
P_M65_PRO))
```

Definition at line 83 of file usb.h.

3.4.2.9 IS_MONOCHROME

```
#define IS_MONOCHROME(
              vendor,
              product ) ((vendor) == (V_CORSAIR) && (product) == (P_STRAFE_NRGB))
```

Definition at line 127 of file usb.h.

Referenced by _setupusb().

3.4.2.10 IS_MONOCHROME_DEV

```
#define IS_MONOCHROME_DEV(
              kb ) IS_MONOCHROME((kb)->vendor, (kb)->product)
```

Definition at line 133 of file usb.h.

3.4.2.11 IS_MOUSE

```
#define IS_MOUSE(
              vendor,
              product ) ((vendor) == (V_CORSAIR) && ((product) == (P_M65) || (product) == (P↩
_M65_PRO) || (product) == (P_SABRE_O) || (product) == (P_SABRE_L) || (product) == (P_SABRE_N) ||
(product) == (P_SCIMITAR) || (product) == (P_SCIMITAR_PRO) || (product) == (P_SABRE_O2)))
```

Definition at line 139 of file usb.h.

Referenced by _setupusb(), get_vtable(), and os_inputmain().

3.4.2.12 IS_MOUSE_DEV

```
#define IS_MOUSE_DEV(
              kb ) IS_MOUSE((kb)->vendor, (kb)->product)
```

Definition at line 142 of file usb.h.

3.4.2.13 IS_RGB

```
#define IS_RGB(
              vendor,
              product ) ((vendor) == (V_CORSAIR) && (product) != (P_K65_NRGB) && (product) !=
(P_K70_NRGB) && (product) != (P_K95_NRGB))
```

Definition at line 122 of file usb.h.

Referenced by _setupusb(), get_vtable(), and os_inputmain().

3.4.2.14 IS_RGB_DEV

```
#define IS_RGB_DEV(
             kb ) IS_RGB((kb)->vendor, (kb)->product)
```

Definition at line 130 of file usb.h.

3.4.2.15 IS_SABRE

```
#define IS_SABRE(
             kb ) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_SABRE_O || (kb)->product
== P_SABRE_L || (kb)->product == P_SABRE_N || (kb)->product == P_SABRE_O2 || (kb)->product ==
P_HARPOON))
```

Definition at line 95 of file usb.h.

3.4.2.16 IS_SCIMITAR

```
#define IS_SCIMITAR(
             kb ) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_SCIMITAR || (kb)->product
== P_SCIMITAR_PRO))
```

Definition at line 101 of file usb.h.

3.4.2.17 IS_STRAFE

```
#define IS_STRAFE(
             kb ) ((kb)->vendor == V_CORSAIR && ((kb)->product == P_STRAFE || (kb)->product
== P_STRAFE_NRGB))
```

Definition at line 77 of file usb.h.

3.4.2.18 NK95_HWOFF

```
#define NK95_HWOFF 0x020030
```

See also

`usb2.0 documentation for details`. Set Hardware playback off

Definition at line 299 of file usb.h.

3.4.2.19 NK95_HWON

```
#define NK95_HWON 0x020001
```

Definition at line 302 of file usb.h.

Referenced by revertusb().

3.4.2.20 NK95_M1

```
#define NK95_M1 0x140001
```

Definition at line 305 of file usb.h.

3.4.2.21 NK95_M2

```
#define NK95_M2 0x140002
```

Definition at line 308 of file usb.h.

3.4.2.22 NK95_M3

```
#define NK95_M3 0x140003
```

Definition at line 311 of file usb.h.

3.4.2.23 nk95cmd

```
#define nk95cmd(
              kb,
              command ) _nk95cmd(kb, (command) >> 16 & 0xFF, (command) & 0xFFFF, __FILE_NOPA↩
TH__, __LINE__)
```

Definition at line 294 of file usb.h.

Referenced by revertusb().

3.4.2.24 P_HARPOON

```
#define P_HARPOON 0x1b3c /* Harpoon test */
```

Definition at line 93 of file usb.h.

Referenced by product_str().

3.4.2.25 P_HARPOON_STR

```
#define P_HARPOON_STR "1b3c"
```

Definition at line 94 of file usb.h.

3.4.2.26 P_K65

```
#define P_K65 0x1b17
```

Definition at line 41 of file usb.h.

Referenced by product_str().

3.4.2.27 P_K65_LUX

```
#define P_K65_LUX 0x1b37
```

Definition at line 45 of file usb.h.

Referenced by product_str().

3.4.2.28 P_K65_LUX_STR

```
#define P_K65_LUX_STR "1b37"
```

Definition at line 46 of file usb.h.

3.4.2.29 P_K65_NRGB

```
#define P_K65_NRGB 0x1b07
```

Definition at line 43 of file usb.h.

Referenced by product_str().

3.4.2.30 P_K65_NRGB_STR

```
#define P_K65_NRGB_STR "1b07"
```

Definition at line 44 of file usb.h.

3.4.2.31 P_K65_RFIRE

```
#define P_K65_RFIRE 0x1b39
```

Definition at line 47 of file usb.h.

Referenced by product_str().

3.4.2.32 P_K65_RFIRE_STR

```
#define P_K65_RFIRE_STR "1b39"
```

Definition at line 48 of file usb.h.

3.4.2.33 P_K65_STR

```
#define P_K65_STR "1b17"
```

Definition at line 42 of file usb.h.

3.4.2.34 P_K70

```
#define P_K70 0x1b13
```

Definition at line 51 of file usb.h.

Referenced by product_str().

3.4.2.35 P_K70_LUX

```
#define P_K70_LUX 0x1b33
```

Definition at line 55 of file usb.h.

Referenced by product_str().

3.4.2.36 P_K70_LUX_NRGB

```
#define P_K70_LUX_NRGB 0x1b36
```

Definition at line 57 of file usb.h.

Referenced by product_str().

3.4.2.37 P_K70_LUX_NRGB_STR

```
#define P_K70_LUX_NRGB_STR "1b36"
```

Definition at line 58 of file usb.h.

3.4.2.38 P_K70_LUX_STR

```
#define P_K70_LUX_STR "1b33"
```

Definition at line 56 of file usb.h.

3.4.2.39 P_K70_NRGB

```
#define P_K70_NRGB 0x1b09
```

Definition at line 53 of file usb.h.

Referenced by product_str().

3.4.2.40 P_K70_NRGB_STR

```
#define P_K70_NRGB_STR "1b09"
```

Definition at line 54 of file usb.h.

3.4.2.41 P_K70_RFIRE

```
#define P_K70_RFIRE 0x1b38
```

Definition at line 59 of file usb.h.

Referenced by product_str().

3.4.2.42 P_K70_RFIRE_NRGB

```
#define P_K70_RFIRE_NRGB 0x1b3a
```

Definition at line 61 of file usb.h.

Referenced by product_str().

3.4.2.43 P_K70_RFIRE_NRGB_STR

```
#define P_K70_RFIRE_NRGB_STR "1b3a"
```

Definition at line 62 of file usb.h.

3.4.2.44 P_K70_RFIRE_STR

```
#define P_K70_RFIRE_STR "1b38"
```

Definition at line 60 of file usb.h.

3.4.2.45 P_K70_STR

```
#define P_K70_STR "1b13"
```

Definition at line 52 of file usb.h.

3.4.2.46 P_K95

```
#define P_K95 0x1b11
```

Definition at line 65 of file usb.h.

Referenced by product_str().

3.4.2.47 P_K95_NRGB

```
#define P_K95_NRGB 0x1b08
```

Definition at line 67 of file usb.h.

Referenced by _nk95cmd(), and product_str().

3.4.2.48 P_K95_NRGB_STR

```
#define P_K95_NRGB_STR "1b08"
```

Definition at line 68 of file usb.h.

3.4.2.49 P_K95_PLATINUM

```
#define P_K95_PLATINUM 0x1b2d
```

Definition at line 69 of file usb.h.

Referenced by product_str().

3.4.2.50 P_K95_PLATINUM_STR

```
#define P_K95_PLATINUM_STR "1b2d"
```

Definition at line 70 of file usb.h.

3.4.2.51 P_K95_STR

```
#define P_K95_STR "1b11"
```

Definition at line 66 of file usb.h.

3.4.2.52 P_M65

```
#define P_M65 0x1b12
```

Definition at line 79 of file usb.h.

Referenced by product_str().

3.4.2.53 P_M65_PRO

```
#define P_M65_PRO 0x1b2e
```

Definition at line 81 of file usb.h.

Referenced by product_str().

3.4.2.54 P_M65_PRO_STR

```
#define P_M65_PRO_STR "1b2e"
```

Definition at line 82 of file usb.h.

3.4.2.55 P_M65_STR

```
#define P_M65_STR "1b12"
```

Definition at line 80 of file usb.h.

3.4.2.56 P_SABRE_L

```
#define P_SABRE_L 0x1b19 /* laser */
```

Definition at line 87 of file usb.h.

Referenced by product_str().

3.4.2.57 P_SABRE_L_STR

```
#define P_SABRE_L_STR "1b19"
```

Definition at line 88 of file usb.h.

3.4.2.58 P_SABRE_N

```
#define P_SABRE_N 0x1b2f /* new?  */
```

Definition at line 89 of file usb.h.

Referenced by product_str().

3.4.2.59 P_SABRE_N_STR

```
#define P_SABRE_N_STR "1b2f"
```

Definition at line 90 of file usb.h.

3.4.2.60 P_SABRE_O

```
#define P_SABRE_O 0x1b14 /* optical */
```

Definition at line 85 of file usb.h.

Referenced by product_str().

3.4.2.61 P_SABRE_O2

```
#define P_SABRE_O2 0x1b32 /* Observed on a CH-9000111-EU model SABRE */
```

Definition at line 91 of file usb.h.

Referenced by product_str().

3.4.2.62 P_SABRE_O2_STR

```
#define P_SABRE_O2_STR "1b32"
```

Definition at line 92 of file usb.h.

3.4.2.63 P_SABRE_O_STR

```
#define P_SABRE_O_STR "1b14"
```

Definition at line 86 of file usb.h.

3.4.2.64 P_SCIMITAR

```
#define P_SCIMITAR 0x1b1e
```

Definition at line 97 of file usb.h.

Referenced by product_str().

3.4.2.65 P_SCIMITAR_PRO

```
#define P_SCIMITAR_PRO 0x1b3e
```

Definition at line 99 of file usb.h.

Referenced by product_str().

3.4.2.66 P_SCIMITAR_PRO_STR

```
#define P_SCIMITAR_PRO_STR "1b3e"
```

Definition at line 100 of file usb.h.

3.4.2.67 P_SCIMITAR_STR

```
#define P_SCIMITAR_STR "1b1e"
```

Definition at line 98 of file usb.h.

3.4.2.68 P_STRAFE

```
#define P_STRAFE 0x1b20
```

Definition at line 73 of file usb.h.

Referenced by product_str().

3.4.2.69 P_STRAFE_NRGB

```
#define P_STRAFE_NRGB 0x1b15
```

Definition at line 75 of file usb.h.

Referenced by product_str().

3.4.2.70 P_STRAFE_NRGB_STR

```
#define P_STRAFE_NRGB_STR "1b15"
```

Definition at line 76 of file usb.h.

3.4.2.71 P_STRAFE_STR

```
#define P_STRAFE_STR "1b20"
```

Definition at line 74 of file usb.h.

3.4.2.72 resetusb

```
#define resetusb(
              kb ) _resetusb(kb, __FILE_NOPATH__, __LINE__)
```

Definition at line 212 of file usb.h.

Referenced by usb_tryreset().

3.4.2.73 USB_DELAY_DEFAULT

```
#define USB_DELAY_DEFAULT 5
```

Definition at line 158 of file usb.h.

Referenced by _setupusb(), and start_dev().

3.4.2.74 usbrecv

```
#define usbrecv(
              kb,
              out_msg,
              in_msg ) _usbrecv(kb, out_msg, in_msg, __FILE_NOPATH__, __LINE__)
```

Parameters

kb	THE usbdevice*
[IN]	out_msg What information does the caller want from the device?
[OUT]	in_msg Here comes the answer; The names represent the usb view, not the view of this function! So INput from usb is OUTput of this function.

Definition at line 254 of file usb.h.

3.4.2.75 usbsend

```
#define usbsend(
              kb,
              messages,
              count ) _usbsend(kb, messages, count, __FILE_NOPATH__, __LINE__)
```

Parameters

kb	THE usbdevice*
[IN]	messages a Pointer to the first byte of the logical message
[IN]	count how many MSG_SIZE buffers is the logical message long?

Definition at line 237 of file usb.h.

3.4.2.76 V_CORSAIR

```
#define V_CORSAIR 0x1b1c
```

Warning

When adding new devices please update src/ckb/fwupgradedialog.cpp as well.
It should contain the same vendor/product IDs for any devices supporting firmware updates.
In the same way, all other corresponding files have to be supplemented or modified: Currently known for this are **usb_linux.c** and **usb_mac.c**

Definition at line 38 of file usb.h.

Referenced by usb_add_device(), and vendor_str().

3.4.2.77 V_CORSAIR_STR

```
#define V_CORSAIR_STR "1b1c"
```

Definition at line 39 of file usb.h.

Referenced by udev_enum(), and usb_add_device().

3.4.3 Function Documentation

3.4.3.1 _nk95cmd()

```
int _nk95cmd (
            usbdevice * kb,
            uchar bRequest,
            ushort wValue,
            const char * file,
            int line )
```

Parameters

kb	THE usbdevice*
bRequest	the byte array with the usb request
wValue	a usb wValue
file	for error message
line	for error message

Returns

1 (true) on failure, 0 (false) on success.

To send control packets to a non RGB non color K95 Keyboard, use this function. Normally it is called via the nk95cmd() macro.

If it is the wrong device for which the function is called, 0 is returned and nothing done. Otherwise a usbdevfs_↩ ctrltransfer structure is filled and an USBDEVFS_CONTROL ioctl() called.

bRequest↩ Type	bRequest	wValue	EP	size	Timeout	data
0x40	see table below to switch hardware-modus at Keyboard	wValue	device	MSG_SIZE	5ms	the message buffer pointer
Host to Device, Type=Vendor, Recipient=Device	bRequest parameter	given wValue Parameter	device 0	0 data to write	5000	null

If a 0 or a negative error number is returned by the ioctl, an error message is shown depending on the errno or "No data written" if retval was 0. In either case 1 is returned to indicate the error. If the ioctl returned a value $>$ 0, 0 is returned to indicate no error.

Currently the following combinations for bRequest and wValue are used:

Device	what it might to do	constant	bRequest	wValue
non RGB Keyboard	set HW-modus on (leave the ckb driver)	HWON	0x0002	0x0030

Device	what it might to do	constant	bRequest	wValue
non RGB Keyboard	set HW-modus off (initialize the ckb driver)	HWOFF	0x0002	0x0001
non RGB Keyboard	set light modus M1 in single-color keyboards	NK95_M1	0x0014	0x0001
non RGB Keyboard	set light modus M2 in single-color keyboards	NK95_M2	0x0014	0x0002
non RGB Keyboard	set light modus M3 in single-color keyboards	NK95_M3	0x0014	0x0003

See also

usb.h

Definition at line 181 of file usb_linux.c.

References P_K95_NRGB.

```
181                                                                                 {
182     if(kb->product != P_K95_NRGB)
183         return 0;
184     struct usbdevfs_ctrltransfer transfer = { 0x40, bRequest, wValue, 0, 0, 5000, 0 };
185     int res = ioctl(kb->handle - 1, USBDEVFS_CONTROL, &transfer);
186     if(res <= 0){
187         ckb_err_fn("%s\n", file, line, res ? strerror(errno) : "No data written");
188         return 1;
189     }
190     return 0;
191 }
```

3.4.3.2 _resetusb()

```
int _resetusb (
            usbdevice * kb,
            const char * file,
            int line )
```

Parameters

kb	THE usbdevice*
file	filename for error messages
line	line where it is called for error messages

Returns

Returns 0 on success, -1 if device should be removed

_resetusb Reset a USB device.

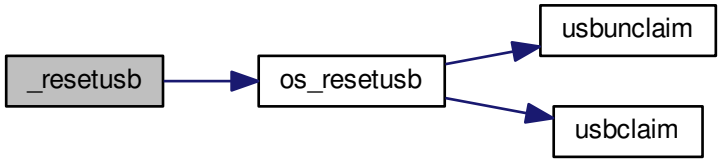
First reset the device via os_resetusb() after a long delay (it may send something to the host). If this worked (retval == 0), give the device another long delay Then perform the initialization via the device specific start() function entry in kb->vtable and if this is successful also, return the result of the device depenten updatergb() with force=true.

Definition at line 429 of file usb.c.

References DELAY_LONG, and os_resetusb().

```
429                                                                 {
430     // Perform a USB reset
431     DELAY_LONG(kb);
432     int res = os_resetusb(kb, file, line);
433     if(res)
434         return res;
435     DELAY_LONG(kb);
436     // Re-initialize the device
437     if(kb->vtable->start(kb, kb->active) != 0)
438         return -1;
439     if(kb->vtable->updatergb(kb, 1) != 0)
440         return -1;
441     return 0;
442 }
```

Here is the call graph for this function:



3.4.3.3 _usbrecv()

```
int _usbrecv (
              usbdevice * kb,
              const uchar * out_msg,
              uchar * in_msg,
              const char * file,
              int line )
```

Parameters

	kb	THE usbdevice*
	[IN]	out_msg What information does the caller want from the device?
	[OUT]	in_msg Here comes the answer; The names represent the usb view, not the view of this function! So INput from usb is OUTput of this function.
	[IN]	file for debugging
	[IN]	line for debugging
in	*reset_stop*	global variable is read

Returns

number of bytes read or zero on failure.

_usbrecv Request data from a USB device by first sending an output packet and then reading the response.

To fully understand this, you need to know about usb: All control is at the usb host (the CPU). If the device wants to communicate something to the host, it must wait for the host to ask. The usb protocol defines the cycles and periods in which actions are to be taken.

So in order to receive a data packet from the device, the host must first send a send request.
This is done by _usbrecv() in the first block by sending the MSG_SIZE large data block from **out_msg** via os_usbsend() as it is a machine depending implementation. The usb target device is as always determined over kb.

For os_usbsend() to know that it is a receive request, the **is_recv** parameter is set to true (1). With this, os_usbsend () generates a control package for the hardware, not a data packet.

If sending of the control package is not successful, a maximum of 5 times the transmission is repeated (including the first attempt). If a non-cancelable error is signaled or the drive is stopped via reset_stop, _usbrecv() immediately returns 0.

After this, the function waits for the requested response from the device using os_usbrecv ().

os_usbrecv() returns 0, -1 or something else.
Zero signals a serious error which is not treatable and _usbrecv() also returns 0.
-1 means that it is a treatable error - a timeout for example - and therefore the next transfer attempt is started after a long pause (DELAY_LONG) if not reset_stop or the wrong hwload_mode require a termination with a return value of 0.

After 5 attempts, _usbrecv () returns and returns 0 as well as an error message.

When data is received, the number of received bytes is returned. This should always be MSG_SIZE, but os_usbrecv() can also return less. It should not be more, because then there would be an unhandled buffer overflow, but it could be less. This would be signaled in os_usbrecv () with a message.

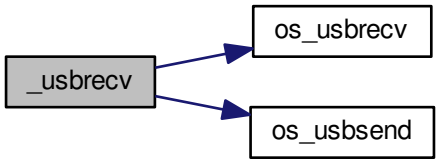
The buffers behind **out_msg** and **in_msg** are MSG_SIZE at least (currently 64 Bytes). More is ok but useless, less brings unpredictable behavior.

Definition at line 602 of file usb.c.

References DELAY_LONG, DELAY_MEDIUM, DELAY_SHORT, hwload_mode, os_usbrecv(), os_usbsend(), and reset_stop.

```
602                                                                                                 {
603     // Try a maximum of 3 times
604     for(int try = 0; try < 5; try++){
605         // Send the output message
606         DELAY_SHORT(kb);
607         int res = os_usbsend(kb, out_msg, 1, file, line);
608         if(res == 0)
609             return 0;
610         else if(res == -1){
611             // Retry on temporary failure
612             if(reset_stop)
613                 return 0;
614             DELAY_LONG(kb);
615             continue;
616         }
617         // Wait for the response
618         DELAY_MEDIUM(kb);
619         res = os_usbrecv(kb, in_msg, file, line);
620         if(res == 0)
621             return 0;
622         else if(res != -1)
623             return res;
624         if(reset_stop || hwload_mode != 2)
625             return 0;
626         DELAY_LONG(kb);
627     }
628     // Give up
629     ckb_err_fn("Too many send/recv failures. Dropping.\n", file, line);
630     return 0;
631 }
```

Here is the call graph for this function:



3.4.3.4 _usbsend()

```
int _usbsend (
            usbdevice * kb,
            const uchar * messages,
            int count,
            const char * file,
            int line )
```

Parameters

	kb	THE usbdevice*
	[IN]	messages a Pointer to the first byte of the logical message
	[IN]	count how many MSG_SIZE buffers is the logical message long?
	[IN]	file for debugging
	[IN]	line for debugging
in	*reset_stop*	global variable is read

Returns

number of Bytes sent (ideal == count * MSG_SIZE);
0 if a block could not be sent and it was not a timeout OR **reset_stop** was required or **hwload_mode** is not set to "always"

_usbsend send a logical message completely to the given device

Todo A lot of different conditions are combined in this code. Don't think, it is good in every combination...

The main task of _usbsend () is to transfer the complete logical message from the buffer beginning with *messages* to **count** * **MSG_SIZE**.
According to usb 2.0 specification, a USB transmits a maximum of 64 byte user data packets. For the transmission of longer messages we need a segmentation. And that is exactly what happens here.

The message is given one by one to os_usbsend() in MSG_SIZE (= 64) byte large bites.

Attention

This means that the buffer given as argument must be n ∗ MSG_SIZE Byte long.

An essential constant parameter which is relevant for os_usbsend() only is is_recv = 0, which means sending.

Now it gets a little complicated again:

- If os_usbsend() returns 0, only zero bytes could be sent in one of the packets, or it was an error (-1 from the systemcall), but not a timeout. How many Bytes were sent in total from earlier calls does not seem to matter, _usbsend() returns a total of 0.
- Returns os_usbsend() -1, first check if **reset_stop** is set globally or (incomprehensible) hwload_mode is not set to "always". In either case, _usbsend() returns 0, otherwise it is assumed to be a temporary transfer error and it simply retransmits the physical packet after a long delay.
- If the return value of os_usbsend() was neither 0 nor -1, it specifies the numer of bytes transferred.
  Here is an information hiding conflict with os_usbsend() (at least in the Linux version):
  If os_usbsend() can not transfer the entire packet, errors are thrown and the number of bytes sent is returned. _usbsend() interprets this as well and remembers the total number of bytes transferred in the local variable **total_sent**. Subsequently, however, transmission is continued with the next complete MSG_SIZE block and not with the first of the possibly missing bytes.

  Todo Check whether this is the same in the macOS variant. It is not dramatic, but if errors occur, it can certainly irritate the devices completely if they receive incomplete data streams. Do we have errors with the messages "Wrote YY bytes (expected 64)" in the system logs? If not, we do not need to look any further.

When the last packet is transferred, _usbsend() returns the effectively counted set of bytes (from **total_sent**). This at least gives the caller the opportunity to check whether something has been lost in the middle.

A bit strange is the structure of the program: Handling the **count** MSG_SIZE blocks to be transferred is done in the outer for (...) loop. Repeating the transfer with a treatable error is managed by the inner while(1) loop.
This must be considered when reading the code; The "break" on successful block transfer leaves the inner while, not the for (...).

Definition at line 535 of file usb.c.

References DELAY_LONG, DELAY_SHORT, hwload_mode, os_usbsend(), and reset_stop.

```
535                                                                                                 {
536     int total_sent = 0;
537     for(int i = 0; i < count; i++){
538         // Send each message via the OS function
539         while(1){
540             DELAY_SHORT(kb);
541             int res = os_usbsend(kb, messages + i * MSG_SIZE, 0, file, line);
542             if(res == 0)
543                 return 0;
544             else if(res != -1){
545                 total_sent += res;
546                 break;
547             }
548             // Stop immediately if the program is shutting down or hardware load is set to tryonce
549             if(reset_stop || hwload_mode != 2)
550                 return 0;
551             // Retry as long as the result is temporary failure
552             DELAY_LONG(kb);
553         }
554     }
555     return total_sent;
556 }
```

Here is the call graph for this function:

3.4.3.5 closeusb()

```
int closeusb (
            usbdevice * kb )
```

Parameters

[IN,OUT]	kb

Returns

Returns 0 (everytime. No error handling is done!)

closeusb Close a USB device and remove device entry.

An imutex lock ensures first of all, that no communication is currently running from the viewpoint of the driver to the user input device (ie the virtual driver with which characters or mouse movements are sent from the daemon to the operating system as inputs).

If the **kb** has an acceptable value = 0, the index of the device is looked for and with this index os_inputclose() is called. After this no more characters can be sent to the operating system.

Then the connection to the usb device is capped by os_closeusb().

Todo What is not yet comprehensible is the call to updateconnected() BEFORE os_closeusb(). Should that be in the other sequence? Or is updateconnected() not displaying the connected usb devices, but the representation which uinput devices are loaded? Questions about questions ...

If there is no valid **handle**, only updateconnected() is called. We are probably trying to disconnect a connection under construction. Not clear.

The cmd pipe as well as all open notify pipes are deleted via rmdevpath ().
This means that nothing can happen to the input path - so the device-specific imutex is unlocked again and remains unlocked.

Also the dmutex is unlocked now, but only to join the thread, which was originally taken under **kb->thread** (which started with _setupusb()) with pthread_join() again. Because of the closed devices that thread would have to quit sometime

See also

the hack note with rmdevpath())

As soon as the thread is caught, the dmutex is locked again, which is what I do not understand yet: What other thread can do usb communication now?
If the vtabel exists for the given kb (why not? It seems to have race conditions here!!), via the vtable the actually device-specific, but still everywhere identical freeprofile() is called. This frees areas that are no longer needed. Then the **usbdevice** structure in its array is set to zero completely.

Error handling is rather unusual in closeusb(); Everything works (no matter what the called functions return), and closeusb() always returns zero (success).

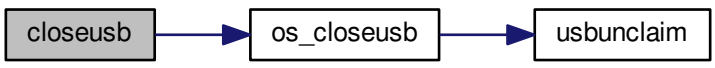
Definition at line 676 of file usb.c.

References keyboard, and os_closeusb().

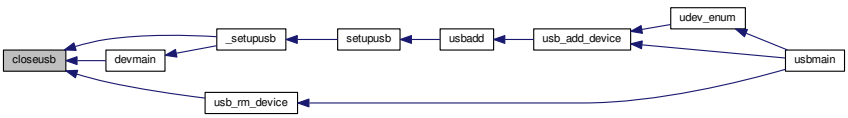
Referenced by _setupusb(), devmain(), and usb_rm_device().

```
676                             {
677     pthread_mutex_lock(imutex(kb));
678     if(kb->handle){
679         int index = INDEX_OF(kb, keyboard);
680         ckb_info("Disconnecting %s%d\n", devpath, index);
681         os_inputclose(kb);
682         updateconnected();
683         // Close USB device
684         os_closeusb(kb);
685     } else
686         updateconnected();
687     rmdevpath(kb);
688 
689     // Wait for thread to close
690     pthread_mutex_unlock(imutex(kb));
691     pthread_mutex_unlock(dmutex(kb));
692     pthread_join(kb->thread, 0);
693     pthread_mutex_lock(dmutex(kb));
694 
695     // Delete the profile and the control path
696     if(!kb->vtable)
697         return 0;
698     kb->vtable->freeprofile(kb);
699     memset(kb, 0, sizeof(usbdevice));
700     return 0;
701 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.4.3.6 os_closeusb()

```
void os_closeusb (
            usbdevice * kb )
```

Parameters

[IN,OUT]	kb THE usbdevice*

os_closeusb unclaim it, destroy the udev device and clear data structures at kb

os_closeusb is the linux specific implementation for closing an active usb port.
If a valid handle is given in the kb structure, the usb port is unclaimed (usbunclaim()).
The device in unrefenced via library function udev_device_unref().
handle, udev and the first char of kbsyspath are cleared to 0 (empty string for kbsyspath).

Definition at line 423 of file usb_linux.c.

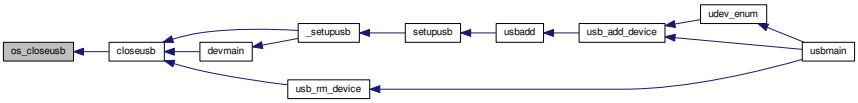
References kbsyspath, keyboard, and usbunclaim().

Referenced by closeusb().

```
423                                 {
424     if(kb->handle){
425         usbunclaim(kb, 0);
426         close(kb->handle - 1);
427     }
428     if(kb->udev)
429         udev_device_unref(kb->udev);
430     kb->handle = 0;
431     kb->udev = 0;
432     kbsyspath[INDEX_OF(kb, keyboard)][0] = 0;
433 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



3.4.3.7 os_inputmain()

```
void* os_inputmain (
            void * context )
```

Parameters

context	THE usbdevice* ; Because os_inputmain() is started as a new thread, its formal parameter is named "context".

Returns

null

os_inputmain is run in a separate thread and will be detached from the main thread, so it needs to clean up its own resources.

Todo This function is a collection of many tasks. It should be divided into several sub-functions for the sake of greater convenience:

1. set up an URB (Userspace Ressource Buffer) to communicate with the USBDEVFS_* ioctl()s
2. perform the ioctl()
3. interpretate the information got into the URB buffer or handle error situations and retry operation or leave the endless loop
4. inform the os about the data
5. loop endless via 2.
6. if endless loop has gone, deinitalize the interface, free buffers etc.
7. return null

Here the actions in detail:

Monitor input transfers on all endpoints for non-RGB devices For RGB, monitor all but the last, as it's used for input/output

Get an usbdevfs_urb data structure and clear it via memset()

Hopefully the buffer lengths are equal for all devices with congruent types. You can find out the correctness for your device with lsusb –v or similar on macOS. Currently the following combinations are known and implemented:

device	detect with macro combination	endpoint #	buffer-length
each	none	0	8
RGB Mouse	IS_RGB && IS_MOUSE	1	10
RGB Keyboard	IS_RGB && !IS_MOUSE	1	21
RGB Mouse or Keyboard	IS_RGB	2	MSG_SIZE (64)
non RGB Mouse or Keyboard	!IS_RGB	1	4
non RGB Mouse or Keyboard	!IS_RGB	2	15

Now submit all the URBs via ioctl(USBDEVFS_SUBMITURB) with type USBDEVFS_URB_TYPE_INTERRUPT (the endpoints are defined as type interrupt). Endpoint number is 0x80..0x82 or 0x83, depending on the model.

The userSpaceFS knows the URBs now, so start monitoring input

if the ioctl returns something != 0, let's have a deeper look what happened. Broken devices or shutting down the entire system leads to closing the device and finishing this thread.

If just an EPIPE ocurred, give the device a CLEAR_HALT and resubmit the URB.

A correct REAPURB returns a Pointer to the URB which we now have a closer look into. Lock all following actions with imutex.

Process the input depending on type of device. Interprete the actual size of the URB buffer

device	detect with macro combination	seems to be end-point #	actual buffer-length	function called
mouse (RGB and non RGB)	IS_MOUSE	nA	8, 10 or 11	hid_mouse_↩ translate()
mouse (RGB and non RGB)	IS_MOUSE	nA	MSG_SIZE (64)	corsair_↩ mousecopy()
RGB Keyboard	IS_RGB && !IS_↩ MOUSE	1	8 (BIOS Mode)	hid_kb_translate()
RGB Keyboard	IS_RGB && !IS_↩ MOUSE	2	5 or 21, KB inactive!	hid_kb_translate()
RGB Keyboard	IS_RGB && !IS_↩ MOUSE	3?	MSG_SIZE	corsair_kbcopy()
non RGB Keyboard	!IS_RGB && !IS_↩ MOUSE	nA	nA	hid_kb_translate()

The input data is transformed and copied to the kb structure. Now give it to the OS and unlock the imutex afterwards.

Re-submit the URB for the next run.

If the endless loop is terminated, clean up by discarding the URBs via ioctl(USBDEVFS_DISCARDURB), free the URB buffers and return a null pointer as thread exit code.

Definition at line 226 of file usb_linux.c.

References IS_MOUSE, IS_RGB, and keyboard.

Referenced by _setupusb().

```
226                                 {
227     usbdevice* kb = context;
228     int fd = kb->handle - 1;
229     short vendor = kb->vendor, product = kb->product;
230     int index = INDEX_OF(kb, keyboard);
231     ckb_info("Starting input thread for %s%d\n", devpath, index);
232 
237     int urbcount = IS_RGB(vendor, product) ? (kb->epcount - 1) : kb->epcount;
238 
240     struct usbdevfs_urb urbs[urbcount];
241     memset(urbs, 0, sizeof(urbs));
242 
256     urbs[0].buffer_length = 8;
257     if(IS_RGB(vendor, product)){
258         if(IS_MOUSE(vendor, product))
259             urbs[1].buffer_length = 10;
260         else
261             urbs[1].buffer_length = 21;
262         urbs[2].buffer_length = MSG_SIZE;
263         if(urbcount != 3)
264             urbs[urbcount - 1].buffer_length = MSG_SIZE;
265     } else {
266         urbs[1].buffer_length = 4;
267         urbs[2].buffer_length = 15;
268     }
269 
272     for(int i = 0; i < urbcount; i++){
```

```
273         urbs[i].type = USBDEVFS_URB_TYPE_INTERRUPT;
274         urbs[i].endpoint = 0x80 | (i + 1);
275         urbs[i].buffer = malloc(urbs[i].buffer_length);
276         int rv = ioctl(fd, USBDEVFS_SUBMITURB, urbs + i);
277         if (rv < 0) ckb_warn("os_inputmain 1: got retcode %d and errno = %d, %s\n", rv, errno, strerror(
    errno));
278     }
279 
281     while (1) {
282         struct usbdevfs_urb* urb = 0;
283         int rv;
284 
287         if ((rv = ioctl(fd, USBDEVFS_REAPURB, &urb))) {
288             ckb_warn("os_inputmain 2: got retcode %d and errno = %d, %s\n", rv, errno, strerror(errno));
289             if (errno == ENODEV || errno == ENOENT || errno == ESHUTDOWN)
290                 // Stop the thread if the handle closes
291                 break;
292             else if(errno == EPIPE && urb){
294                 rv = ioctl(fd, USBDEVFS_CLEAR_HALT, &urb->endpoint);
295                 ckb_warn("os_inputmain 3: got retcode %d and errno = %d, %s\n", rv, errno, strerror(errno))
    ;
296                 // Re-submit the URB
297                 if(urb)
298                     rv = ioctl(fd, USBDEVFS_SUBMITURB, urb);
299                     if (rv < 0) ckb_warn("os_inputmain 4: got retcode %d and errno = %d, %s\n", rv, errno,
    strerror(errno));
300                 urb = 0;
301             }
302         }
303 
307         if (urb) {
319             pthread_mutex_lock(imutex(kb));
320             if(IS_MOUSE(vendor, product)){
321                 switch(urb->actual_length){
322                 case 8:
323                 case 10:
324                 case 11:
325                     // HID mouse input
326                     hid_mouse_translate(kb->input.keys, &kb->input.rel_x, &kb->input.rel_y, -(urb->endpoint
     & 0xF), urb->actual_length, urb->buffer);
327                     break;
328                 case MSG_SIZE:
329                     // Corsair mouse input
330                     corsair_mousecopy(kb->input.keys, -(urb->endpoint & 0xF), urb->buffer);
331                     break;
332                 }
333             } else if(IS_RGB(vendor, product)){
334                 switch(urb->actual_length){
335                 case 8:
336                     // RGB EP 1: 6KRO (BIOS mode) input
337                     hid_kb_translate(kb->input.keys, -1, urb->actual_length, urb->buffer);
338                     break;
339                 case 21:
340                 case 5:
341                     // RGB EP 2: NKRO (non-BIOS) input. Accept only if keyboard is inactive
342                     if(!kb->active)
343                         hid_kb_translate(kb->input.keys, -2, urb->actual_length, urb->buffer);
344                     break;
345                 case MSG_SIZE:
346                     // RGB EP 3: Corsair input
347                     corsair_kbcopy(kb->input.keys, -(urb->endpoint & 0xF), urb->buffer);
348                     break;
349                 }
350             } else {
351                 // Non-RGB input
352                 hid_kb_translate(kb->input.keys, urb->endpoint & 0xF, urb->actual_length, urb->buffer);
353             }
356             inputupdate(kb);
357             pthread_mutex_unlock(imutex(kb));
359             rv = ioctl(fd, USBDEVFS_SUBMITURB, urb);
360             if (rv < 0) ckb_warn("os_inputmain 5: got retcode %d and errno = %d, %s\n", rv, errno, strerror
    (errno));
361             urb = 0;
362         }
363     }
364 
368     ckb_info("Stopping input thread for %s%d\n", devpath, index);
369     for(int i = 0; i < urbcount; i++){
370         ioctl(fd, USBDEVFS_DISCARDURB, urbs + i);
371         free(urbs[i].buffer);
372     }
373     return 0;
374 }
```

Here is the caller graph for this function:

3.4.3.8 os_resetusb()

```
int os_resetusb (
            usbdevice * kb,
            const char * file,
            int line )
```

Parameters

kb	THE usbdevice*
file	filename for error messages
line	line where it is called for error messages

Returns

Returns 0 on success, -2 if device should be removed and -1 if reset should by tried again

os_resetusb is the os specific implementation for resetting usb

Try to reset an usb device in a linux user space driver.

1. unclaim the device, but do not reconnect the system driver (second param resetting = true)
2. reset the device via USBDEVFS_RESET command
3. claim the device again. Returns 0 on success, -2 if device should be removed and -1 if reset should by tried again

Todo it seems that no one wants to try the reset again. But I'v seen it somewhere...

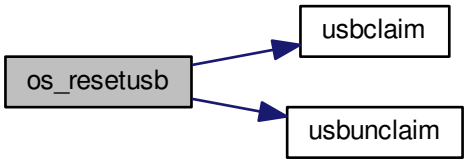
Definition at line 485 of file usb_linux.c.

References TEST_RESET, usbclaim(), and usbunclaim().

Referenced by _resetusb().

```
485                                                     {
486     ckb_warn("os_resetusb: resetting %s\n", kb->name);
487     TEST_RESET(usbunclaim(kb, 1));
488     TEST_RESET(ioctl(kb->handle - 1, USBDEVFS_RESET));
489     TEST_RESET(usbclaim(kb));
490     // Success!
491     return 0;
492 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

3.4.3.9 os_sendindicators()

```
void os_sendindicators (
            usbdevice * kb )
```

Parameters

kb	THE usbdevice*

os_sendindicators update the indicators for the special keys (Numlock, Capslock and what else?)

os_sendindicators update the indicators for the special keys (Numlock, Capslock and what else?)

Read the data from kb->ileds ans send them via ioctl() to the keyboard.

bRequest↩ Type	bRequest	wValue	EP	size	Timeout	data
0x21	0x09	0x0200	Interface 0	MSG_SIZE 1 Byte	timeout 0,5ms	the message buffer pointer
Host to Device, Type=Class, Recipient=Interface (why not endpoint?)	9 = SEND?	specific	0	1	500	struct* kb->ileds

The ioctl command is USBDEVFS_CONTROL.

Definition at line 206 of file usb_linux.c.

```
206                                     {
207     struct usbdevfs_ctrltransfer transfer = { 0x21, 0x09, 0x0200, 0x00, 1, 500, &kb->ileds };
208     int res = ioctl(kb->handle - 1, USBDEVFS_CONTROL, &transfer);
209     if(res <= 0)
210         ckb_err("%s\n", res ? strerror(errno) : "No data written");
211 }
```

3.4.3.10 os_setupusb()

```
int os_setupusb (
            usbdevice * kb )
```

Parameters

kb	THE usbdevice*

Returns

0 on success, -1 otherwise.

os_setupusb OS-specific setup for a specific usb device.

Perform the operating system-specific opening of the interface in os_setupusb(). As a result, some parameters should be set in kb (name, serial, fwversion, epcount = number of usb endpoints), and all endpoints should be claimed with usbclaim(). Claiming is the only point where os_setupusb() can produce an error (-1).

- Copy device description and serial
- Copy firmware version (needed to determine USB protocol)
- Do some output about connecting interfaces
- Claim the USB interfaces

Todo in these modules a pullrequest is outstanding

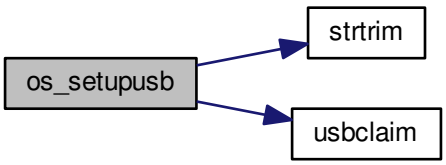
Definition at line 524 of file usb_linux.c.

References keyboard, strtrim(), and usbclaim().

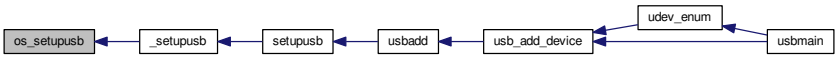
Referenced by _setupusb().

```
524                                 {
527     struct udev_device* dev = kb->udev;
528     const char* name = udev_device_get_sysattr_value(dev, "product");
529     if(name)
530         strncpy(kb->name, name, KB_NAME_LEN);
531     strtrim(kb->name);
532     const char* serial = udev_device_get_sysattr_value(dev, "serial");
533     if(serial)
534         strncpy(kb->serial, serial, SERIAL_LEN);
535     strtrim(kb->serial);
538     const char* firmware = udev_device_get_sysattr_value(dev, "bcdDevice");
539     if(firmware)
540         sscanf(firmware, "%hx", &kb->fwversion);
541     else
542         kb->fwversion = 0;
543     int index = INDEX_OF(kb, keyboard);
546     ckb_info("Connecting %s at %s%d\n", kb->name, devpath, index);
547
553     const char* ep_str = udev_device_get_sysattr_value(dev, "bNumInterfaces");
554     kb->epcount = 0;
555     if(ep_str)
556         sscanf(ep_str, "%d", &kb->epcount);
557     if(kb->epcount == 0){
558         // This shouldn't happen, but if it does, assume EP count based on what the device is supposed to
      have
559         kb->epcount = (HAS_FEATURES(kb, FEAT_RGB) ? 4 : 3);
560         ckb_warn("Unable to read endpoint count from udev, assuming %d...\n", kb->epcount);
561     }
562     if(usbclaim(kb)){
563         ckb_err("Failed to claim interfaces: %s\n", strerror(errno));
564         return -1;
565     }
566     return 0;
567 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.4.3.11 os_usbrecv()

```
int os_usbrecv (
            usbdevice * kb,
            uchar * in_msg,
            const char * file,
            int line )
```

Parameters

kb	THE usbdevice*
in_msg	the buffer to fill with the message received
file	for debugging
line	for debugging

Returns

-1 on timeout, 0 on hard error, numer of bytes received otherwise

os_usbrecv does what its name says:

The comment at the beginning of the procedure causes the suspicion that the firmware versionspecific distinction is missing for receiving from usb endpoint 3 or 4. The commented code contains only the reception from EP4, but this may be wrong for a software version 2.0 or higher (see the code for os-usbsend ()).

So all the receiving is done via an ioctl() like in os_usbsend. The ioctl() is given a struct usbdevfs_ctrltransfer, in which the relevant parameters are entered:

bRequest↩ Type	bRequest	wValue	EP	size	Timeout	data
0xA1	0x01	0x0200	endpoint to be addressed from epcount - 1	MSG_SIZE	5ms	the message buffer pointer
Device to Host, Type=Class, Recipi- ent=Interface	1 = RECEIVE?	specific	Interface #	64	5000	in_msg

The ioctl() returns the number of bytes received. Here is the usual check again:

- If the return value is -1 AND the error is a timeout (ETIMEOUT), os_usbrecv() will return -1 to indicate that it is probably a recoverable problem and a retry is recommended.
- For another negative value or other error identifier OR 0 bytes are received, 0 is returned as an identifier for a heavy error.
- In all other cases, the function returns the number of bytes received.

If this is not the entire blocksize (MSG_SIZE bytes), an error message is issued on the standard error channel [warning "Read YY bytes (expected 64)"].

Definition at line 127 of file usb_linux.c.

Referenced by _usbrecv().

```
127                                                                     {
128     int res;
129     // This is what CUE does, but it doesn't seem to work on linux.
130     /*if(kb->fwversion >= 0x130){
131         struct usbdevfs_bulktransfer transfer;
132         memset(&transfer, 0, sizeof(transfer));
133         transfer.ep = 0x84;
```

```
134         transfer.len = MSG_SIZE;
135         transfer.timeout = 5000;
136         transfer.data = in_msg;
137         res = ioctl(kb->handle - 1, USBDEVFS_BULK, &transfer);
138     } else {*/
139         struct usbdevfs_ctrltransfer transfer = { 0xa1, 0x01, 0x0300, kb->epcount - 1, MSG_SIZE, 5000,
    in_msg };
140         res = ioctl(kb->handle - 1, USBDEVFS_CONTROL, &transfer);
141     //}
142     if(res <= 0){
143         ckb_err_fn("%s\n", file, line, res ? strerror(errno) : "No data read");
144         if(res == -1 && errno == ETIMEDOUT)
145             return -1;
146         else
147             return 0;
148     } else if(res != MSG_SIZE)
149         ckb_warn_fn("Read %d bytes (expected %d)\n", file, line, res, MSG_SIZE);
150     return res;
151 }
```

Here is the caller graph for this function:

3.4.3.12 os_usbsend()

```
int os_usbsend (
            usbdevice * kb,
            const uchar * out_msg,
            int is_recv,
            const char * file,
            int line )
```

Parameters

kb	THE usbdevice*
out_msg	the MSGSIZE char long buffer to send
is_recv	if true, just send an ioctl for further reading packets. If false, send the data at **out_msg**.
file	for debugging
line	for debugging

Returns

-1 on timeout (try again), 0 on hard error, numer of bytes sent otherwise

os_usbsend has two functions:

- if is_recv == false, it tries to send a given MSG_SIZE buffer via the usb interface given with kb.

- otherwise a request is sent via the usb device to initiate the receiving of a message from the remote device.

The functionality for sending distinguishes two cases, depending on the version number of the firmware of the connected device:
If the firmware is less or equal 1.2, the transmission is done via an ioctl(). The ioctl() is given a struct usbdevfs_↩ ctrltransfer, in which the relevant parameters are entered:

bRequest↩ Type	bRequest	wValue	EP	size	Timeout	data
0x21	0x09	0x0200	endpoint / IF to be addressed from epcount-1	MSG_SIZE	5000 (=5ms)	the message buffer pointer
Host to Device, Type=Class, Recipi-ent=Interface	9 = Send data?	specific	last or pre-last device #	64	5000	out_msg

The ioctl command is USBDEVFS_CONTROL.

The same constellation is used if the device is requested to send its data (is_recv = true).

For a more recent firmware and is_recv = false, the ioctl command USBDEVFS_CONTROL is not used (this tells the bus to enter the control mode), but the bulk method is used: USBDEVFS_BULK. This is astonishing, because all of the endpoints are type Interrupt, not bulk.

Anyhow, forthis purpose a different structure is used for the ioctl() (struct **usbdevfs_bulktransfer**) and this is also initialized differently:
The length and timeout parameters are given the same values as above. The formal parameter out_msg is also passed as a buffer pointer. For the endpoints, the firmware version is differentiated again:
For a firmware version between 1.3 and <2.0 endpoint 4 is used, otherwise (it can only be >=2.0) endpoint 3 is used.

Todo Since the handling of endpoints has already led to problems elsewhere, this implementation is extremely hardware-dependent and critical!
Eg. the new keyboard K95PLATINUMRGB has a version number significantly less than 2.0 - will it run with this implementation?

The ioctl() - no matter what type - returns the number of bytes sent. Now comes the usual check:

- If the return value is -1 AND the error is a timeout (ETIMEOUT), os_usbsend() will return -1 to indicate that it is probably a recoverable problem and a retry is recommended.
- For another negative value or other error identifier OR 0 bytes sent, 0 is returned as a heavy error identifier.
- In all other cases, the function returns the number of bytes sent.

If this is not the entire blocksize (MSG_SIZE bytes), an error message is issued on the standard error channel [warning "Wrote YY bytes (expected 64)"].

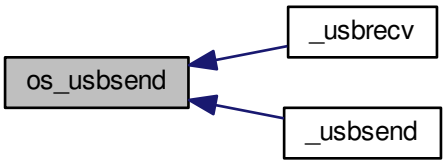
If DEBUG_USB is set during compilation, the number of bytes sent and their representation are logged to the error channel.

Definition at line 66 of file usb_linux.c.

Referenced by _usbrecv(), and _usbsend().

```
66                                                                                                  {
67     int res;
68     if(kb->fwversion >= 0x120 && !is_recv){
69         struct usbdevfs_bulktransfer transfer;
70         memset(&transfer, 0, sizeof(transfer));
71         transfer.ep = (kb->fwversion >= 0x130 && kb->fwversion < 0x200) ? 4 : 3;
72         transfer.len = MSG_SIZE;
73         transfer.timeout = 5000;
74         transfer.data = (void*)out_msg;
75         res = ioctl(kb->handle - 1, USBDEVFS_BULK, &transfer);
76     } else {
77         struct usbdevfs_ctrltransfer transfer = { 0x21, 0x09, 0x0200, kb->epcount - 1, MSG_SIZE, 5000, (
    void*)out_msg };
78         res = ioctl(kb->handle - 1, USBDEVFS_CONTROL, &transfer);
79     }
80     if(res <= 0){
81         ckb_err_fn("%s\n", file, line, res ? strerror(errno) : "No data written");
82         if(res == -1 && errno == ETIMEDOUT)
83             return -1;
84         else
85             return 0;
86     } else if(res != MSG_SIZE)
87         ckb_warn_fn("Wrote %d bytes (expected %d)\n", file, line, res, MSG_SIZE);
88 #ifdef DEBUG_USB
89     char converted[MSG_SIZE*3 + 1];
90     for(int i=0;i<MSG_SIZE;i++)
91         sprintf(&converted[i*3], "%02x ", out_msg[i]);
92     ckb_warn_fn("Sent %s\n", file, line, converted);
93 #endif
94     return res;
95 }
```

Here is the caller graph for this function:



3.4.3.13 product_str()

```
const char* product_str (
            short product )
```

Parameters

product	is the *short* USB device product ID

Returns

string to identify a type of device (see below)

product_str returns a condensed view on what type of device we have.

At present, various models and their properties are known from corsair products. Some models differ in principle (mice and keyboards), others differ in the way they function (for example, RGB and non RGB), but they are very similar.

Here, only the first point is taken into consideration and we return a unified model string. If the model is not known with its number, *product_str* returns an empty string.

The model numbers and corresponding strings wwith the numbers in hex-string are defined in `usb.h`

At present, this function is used to initialize `kb->name` and to give information in debug strings.

Attention

The combinations below have to fit to the combinations in the macros mentioned above. So if you add a device with a new number, change both.

Todo There are macros defined in usb.h to detect all the combinations below. the only difference is the parameter: The macros need the *kb∗*, product_str() needs the *product ID*

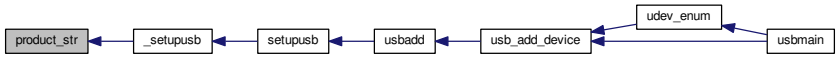
Definition at line 70 of file usb.c.

References P_HARPOON, P_K65, P_K65_LUX, P_K65_NRGB, P_K65_RFIRE, P_K70, P_K70_LUX, P_K70_↩ LUX_NRGB, P_K70_NRGB, P_K70_RFIRE, P_K70_RFIRE_NRGB, P_K95, P_K95_NRGB, P_K95_PLATINUM, P_M65, P_M65_PRO, P_SABRE_L, P_SABRE_N, P_SABRE_O, P_SABRE_O2, P_SCIMITAR, P_SCIMITAR_↩ PRO, P_STRAFE, and P_STRAFE_NRGB.

Referenced by _setupusb().

```
70                                      {
71     if(product == P_K95 || product == P_K95_NRGB || product ==
       P_K95_PLATINUM)
72         return "k95";
73     if(product == P_K70 || product == P_K70_NRGB || product ==
       P_K70_LUX || product == P_K70_LUX_NRGB || product ==
       P_K70_RFIRE || product == P_K70_RFIRE_NRGB)
74         return "k70";
75     if(product == P_K65 || product == P_K65_NRGB || product ==
       P_K65_LUX || product == P_K65_RFIRE)
76         return "k65";
77     if(product == P_STRAFE || product == P_STRAFE_NRGB)
78         return "strafe";
79     if(product == P_M65 || product == P_M65_PRO)
80         return "m65";
81     if(product == P_SABRE_O || product == P_SABRE_L || product ==
       P_SABRE_N || product == P_SABRE_O2 || product == P_HARPOON)
82         return "sabre";
83     if(product == P_SCIMITAR || product == P_SCIMITAR_PRO)
84         return "scimitar";
85     return "";
86 }
```

Here is the caller graph for this function:



3.4.3.14 revertusb()

```
int revertusb (
            usbdevice * kb )
```

Parameters

kb	THE usbdevice*

Returns

0 on success or if device needs firmware upgrade, -1 otherwise

revertusb sets a given device to inactive (hardware controlled) mode if not a fw-ugrade is indicated

First is checked, whether a firmware-upgrade is indicated for the device. If so, revertusb() returns 0.

Todo Why is this useful? Are there problems seen with deactivating a device with older fw-version??? Why isn't this an error indicating reason and we return success (0)?

Anyway, the following steps are similar to some other procs, dealing with low level usb handling:

- If we do not have an RGB device, a simple setting to Hardware-mode (NK95_HWON) is sent to the device via n95cmd().

  Todo The return value of nk95cmd() is ignored (but sending the ioctl may produce an error and _nk95_cmd will indicate this), instead revertusb() returns success in any case.

- If we have an RGB device, setactive() is called with second param active = false. That function will have a look on differences between keyboards and mice.
  More precisely setactive() is just a macro to call via the kb->vtable enties either the active() or the idle() function where the vtable points to. setactive() may return error indications. If so, revertusb() returns -1, otherwise 0 in any other case.

Definition at line 410 of file usb.c.

References NK95_HWON, and nk95cmd.

```
410                             {
411     if(NEEDS_FW_UPDATE(kb))
412         return 0;
413     if(!HAS_FEATURES(kb, FEAT_RGB)){
414         nk95cmd(kb, NK95_HWON);
415         return 0;
416     }
417     if(setactive(kb, 0))
418         return -1;
419     return 0;
420 }
```

3.4.3.15 setupusb()

```
void setupusb (
             usbdevice * kb )
```

Attention

Lock a device's dmutex (see device.h) before accessing the USB interface.

Parameters

kb	THE usbdevice* used everywhere
[OUT]	kb->thread is used to store the thread ID of the fresh created thread.

setupusb starts a thread with kb as parameter and _setupusb() as entrypoint.

Set up a USB device after its handle is open. Spawns a new thread _setupusb() with standard parameter kb. dmutex must be locked prior to calling this function. The function will unlock it when finished. In kb->thread the thread id is mentioned, because closeusb() needs this info for joining that thread again.

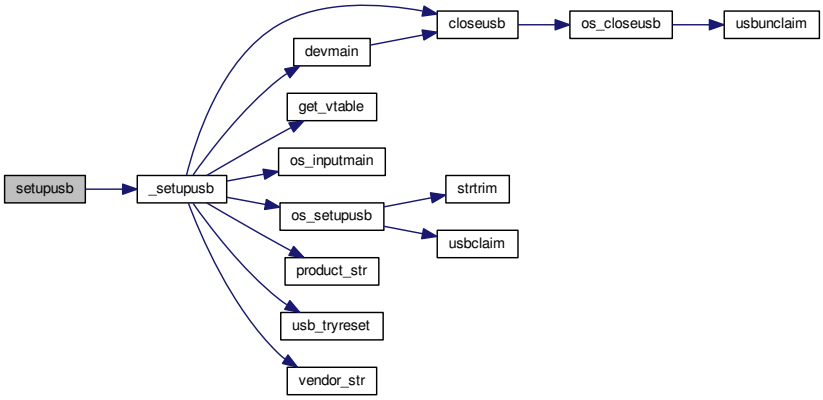
Definition at line 389 of file usb.c.

References _setupusb().

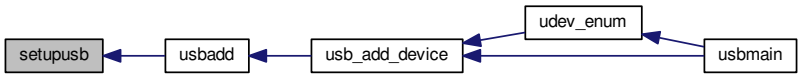
Referenced by usbadd().

```
389                             {
390     pthread_mutex_lock(imutex(kb));
391     if(pthread_create(&kb->thread, 0, _setupusb, kb))
392         ckb_err("Failed to create USB thread\n");
393 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.4.3.16 usb_tryreset()

```
int usb_tryreset (
            usbdevice * kb )
```

Parameters

in,out	*kb*	THE usbdevice*
in	*reset_stop*	global variable is read

Returns

0 on success, -1 otherwise

usb_tryreset does what the name means: Try to reset the usb via resetusb()

This function is called if an usb command ran into an error in case of one of the following two situations:

- When setting up a new usb device and the start() function got an error (

  See also

  _setupusb())

- If upgrading to a new firmware gets an error (

  See also

  cmd_fwupdate()).

  The previous action which got the error will NOT be re-attempted.

In an endless loop usb_tryreset() tries to reset the given usb device via the macro resetusb().
This macro calls _resetusb() with debugging information.
_resetusb() sends a command via the operating system dependent function os_resetusb() and - if successful - reinitializes the device. os_resetusb() returns -2 to indicate a broken device and all structures should be removed for it.
In that case, the loop is terminated, an error message is produced and usb_tryreset() returns -1.

In case resetusb() has success, the endless loop is left via a return 0 (success).
If the return value from resetusb() is -1, the loop is continued with the next try.

If the global variable **reset_stop** is set directly when the function is called or after each try, usb_tryreset() stops working and returns -1.

Todo Why does usb_tryreset() hide the information returned from resetusb()? Isn't it needed by the callers?

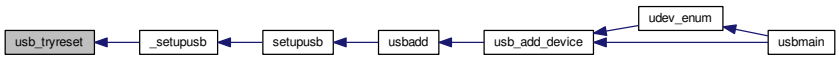
Definition at line 468 of file usb.c.

References hwload_mode, reset_stop, and resetusb.

Referenced by _setupusb().

```
468                                 {
469     if(reset_stop)
470         return -1;
471     ckb_info("Attempting reset...\n");
472     while(1){
473         int res = resetusb(kb);
474         if(!res){
475             ckb_info("Reset success\n");
476             return 0;
477         }
478         if(res == -2 || reset_stop)
479             break;
480     }
481     ckb_err("Reset failed. Disconnecting.\n");
482     return -1;
483 }
```

Here is the caller graph for this function:



3.4.3.17 usbkill()

```
void usbkill ( )
```

Stop the USB system.

Definition at line 815 of file usb_linux.c.

References udev.

```
815                 {
816     udev_unref(udev);
817     udev = 0;
818 }
```

3.4.3.18 usbmain()

```
int usbmain ( )
```

Start the USB main loop. Returns program exit code when finished.

usbmain is called by main() after setting up all other stuff.

Returns

0 normally or -1 if fatal error occurs (up to now only if no new devices are available)

First check whether the uinput module is loaded by the kernel.

Todo Why isn't missing of uinput a fatal error?

Create the udev object with udev_new() (is a function from libudev.h) terminate -1 if error

Enumerate all currently connected devices

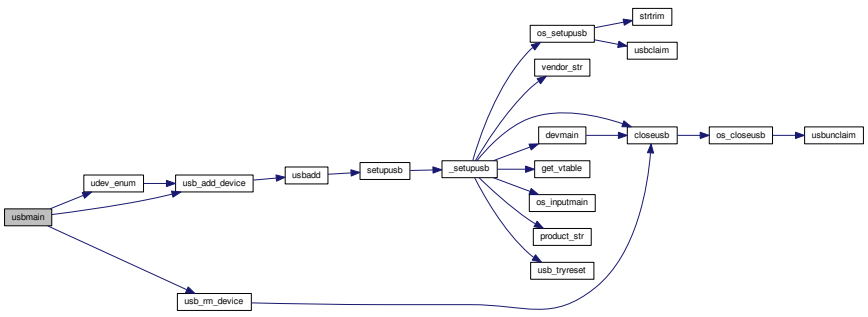
Todo lae. here the work has to go on...

Definition at line 752 of file usb_linux.c.

References udev, udev_enum(), usb_add_device(), and usb_rm_device().

```
752             {
757     // Load the uinput module (if it's not loaded already)
758     if(system("modprobe uinput") != 0)
759         ckb_warn("Failed to load uinput module\n");
760 
764     if(!(udev = udev_new())) {
765         ckb_fatal("Failed to initialize udev in usbmain(), usb_linux.c\n");
766         return -1;
767     }
768 
771     udev_enum();
772 
775     // Done scanning. Enter a loop to poll for device updates
776     struct udev_monitor* monitor = udev_monitor_new_from_netlink(udev, "udev");
777     udev_monitor_filter_add_match_subsystem_devtype(monitor, "usb", 0);
778     udev_monitor_enable_receiving(monitor);
779     // Get an fd for the monitor
780     int fd = udev_monitor_get_fd(monitor);
781     fd_set fds;
782     while(udev){
783         FD_ZERO(&fds);
784         FD_SET(fd, &fds);
785         // Block until an event is read
786         if(select(fd + 1, &fds, 0, 0, 0) > 0 && FD_ISSET(fd, &fds)){
787             struct udev_device* dev = udev_monitor_receive_device(monitor);
788             if(!dev)
789                 continue;
790             const char* action = udev_device_get_action(dev);
791             if(!action){
792                 udev_device_unref(dev);
793                 continue;
794             }
795             // Add/remove device
796             if(!strcmp(action, "add")){
797                 int res = usb_add_device(dev);
798                 if(res == 0)
799                     continue;
800                 // If the device matched but the handle wasn't opened correctly, re-enumerate (this
    sometimes solves the problem)
801                 if(res == -1)
802                     udev_enum();
803             } else if(!strcmp(action, "remove"))
804                 usb_rm_device(dev);
805             udev_device_unref(dev);
806         }
807     }
808     udev_monitor_unref(monitor);
809     return 0;
810 }
```

Here is the call graph for this function:



3.4.3.19 vendor_str()

```
const char* vendor_str (
            short vendor )
```

vendor_str Vendor/product string representations

Parameters

vendor	*short* vendor ID

Returns

a string: either "" or "corsair"

uncomment the following Define to see USB packets sent to the device

vendor_str returns "corsair" iff the given *vendor* argument is equal to *V_CORSAIR* (0x1bc) else it returns ""

Attention

There is also a string defined V_CORSAIR_STR, which returns the device number as string in hex "1b1c".

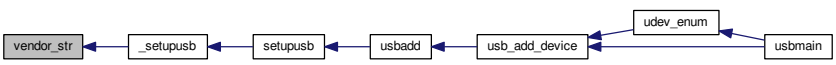
Definition at line 43 of file usb.c.

References V_CORSAIR.

Referenced by _setupusb().

```
43                                      {
44     if(vendor == V_CORSAIR)
45         return "corsair";
46     return "";
47 }
```

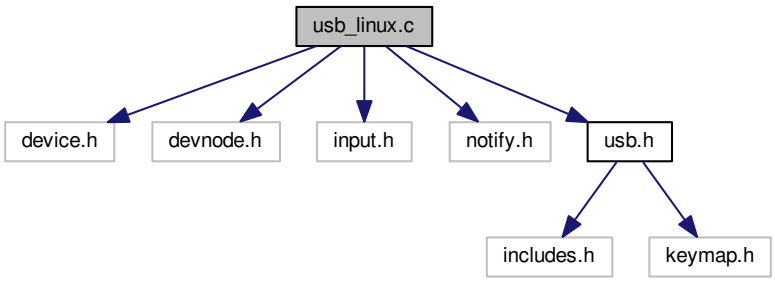
Here is the caller graph for this function:



3.5 usb_linux.c File Reference

```
#include "device.h"
#include "devnode.h"
#include "input.h"
#include "notify.h"
#include "usb.h"
```

Include dependency graph for usb_linux.c:



Data Structures

- struct _model

  typedef struct _model Structure to map string representation of vendor / product and its short number representation- More...

Macros

- #define TEST_RESET(op)

  TEST_RESET doesa "try / catch" for resetting the usb interface.
- #define N_MODELS (sizeof(models) / sizeof(_model))

Functions

- int os_usbsend (usbdevice ∗kb, const uchar ∗out_msg, int is_recv, const char ∗file, int line)

  os_usbsend sends a data packet (MSG_SIZE = 64) Bytes long
- int os_usbrecv (usbdevice ∗kb, uchar ∗in_msg, const char ∗file, int line)

  os_usbrecv receives a max MSGSIZE long buffer from usb device
- int _nk95cmd (usbdevice ∗kb, uchar bRequest, ushort wValue, const char ∗file, int line)

  _nk95cmd If we control a non RGB keyboard, set the keyboard via ioctl with usbdevfs_ctrltransfer
- void os_sendindicators (usbdevice ∗kb)
- void ∗ os_inputmain (void ∗context)

  os_inputmain This function is run in a separate thread and will be detached from the main thread, so it needs to clean up its own resources.
- static int usbunclaim (usbdevice ∗kb, int resetting)
- void os_closeusb (usbdevice ∗kb)
- static int usbclaim (usbdevice ∗kb)

- int os_resetusb (usbdevice *kb, const char *file, int line)
- void strtrim (char *string)
- int os_setupusb (usbdevice *kb)
- int usbadd (struct udev_device *dev, short vendor, short product)
- static int usb_add_device (struct udev_device *dev)

  Add a udev device.
- static void usb_rm_device (struct udev_device *dev)

  usb_rm_device find the usb port to remove and close it via closeusb().
- static void udev_enum ()

  udev_enum use the udev_enumerate_add_match_subsystem() to get all you need but only that.
- int usbmain ()
- void usbkill ()

  usbkill unreference the udev structure via libudev and mark static pointer with 0 as not set.

Variables

- static char kbsyspath [9][FILENAME_MAX]

  all open usb devices have their system path names here in this array.
- static struct udev * udev

  udev struct udef holds the information about the user space devices; used in usbmain() to monitor the usb changes.
- pthread_t usbthread

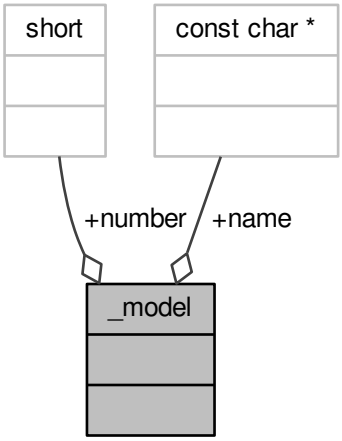
  struct udef is defined in /usr/include/libudev.h
- pthread_t udevthread
- static _model models [ ]

3.5.1 Data Structure Documentation

3.5.1.1 struct _model

Definition at line 620 of file usb_linux.c.

Collaboration diagram for _model:



Data Fields

const char *	name	
short	number	

3.5.2 Macro Definition Documentation

3.5.2.1 N_MODELS

```
#define N_MODELS (sizeof(models) / sizeof(_model))
```

Definition at line 659 of file usb_linux.c.

Referenced by usb_add_device().

3.5.2.2 TEST_RESET

```
#define TEST_RESET(
               op )
```

Value:

```
if(op){                                                                 \
        ckb_err_fn("resetusb failed: %s\n", file, line, strerror(errno));   \
        if(errno == EINTR || errno == EAGAIN)                           \
            return -1;              /* try again if status code says so */  \
        return -2;                  /* else, remove device */               \
    }
```

Definition at line 467 of file usb_linux.c.

Referenced by os_resetusb().

3.5.3 Function Documentation

3.5.3.1 _nk95cmd()

```
int _nk95cmd (
              usbdevice * kb,
              uchar bRequest,
              ushort wValue,
              const char * file,
              int line )
```

To send control packets to a non RGB non color K95 Keyboard, use this function. Normally it is called via the nk95cmd() macro.

If it is the wrong device for which the function is called, 0 is returned and nothing done. Otherwise a usbdevfs_↩ ctrltransfer structure is filled and an USBDEVFS_CONTROL ioctl() called.

bRequest↩ Type	bRequest	wValue	EP	size	Timeout	data
0x40	see table below to switch hardware-modus at Keyboard	wValue	device	MSG_SIZE	5ms	the message buffer pointer
Host to Device, Type=Vendor, Recipient=Device	bRequest parameter	given wValue Parameter	device 0	0 data to write	5000	null

If a 0 or a negative error number is returned by the ioctl, an error message is shown depending on the errno or "No data written" if retval was 0. In either case 1 is returned to indicate the error. If the ioctl returned a value $>$ 0, 0 is returned to indicate no error.

Currently the following combinations for bRequest and wValue are used:

Device	what it might to do	constant	bRequest	wValue
non RGB Keyboard	set HW-modus on (leave the ckb driver)	HWON	0x0002	0x0030
non RGB Keyboard	set HW-modus off (initialize the ckb driver)	HWOFF	0x0002	0x0001
non RGB Keyboard	set light modus M1 in single-color keyboards	NK95_M1	0x0014	0x0001
non RGB Keyboard	set light modus M2 in single-color keyboards	NK95_M2	0x0014	0x0002
non RGB Keyboard	set light modus M3 in single-color keyboards	NK95_M3	0x0014	0x0003

See also

usb.h

Definition at line 181 of file usb_linux.c.

References P_K95_NRGB.

```
181                                                                                 {
182     if(kb->product != P_K95_NRGB)
183         return 0;
184     struct usbdevfs_ctrltransfer transfer = { 0x40, bRequest, wValue, 0, 0, 5000, 0 };
185     int res = ioctl(kb->handle - 1, USBDEVFS_CONTROL, &transfer);
186     if(res <= 0){
187         ckb_err_fn("%s\n", file, line, res ? strerror(errno) : "No data written");
188         return 1;
189     }
190     return 0;
191 }
```

3.5.3.2 os_closeusb()

```
void os_closeusb (
            usbdevice * kb )
```

os_closeusb unclaim it, destroy the udev device and clear data structures at kb

os_closeusb is the linux specific implementation for closing an active usb port.
If a valid handle is given in the kb structure, the usb port is unclaimed (usbunclaim()).
The device in unrefenced via library function udev_device_unref().
handle, udev and the first char of kbsyspath are cleared to 0 (empty string for kbsyspath).

Definition at line 423 of file usb_linux.c.

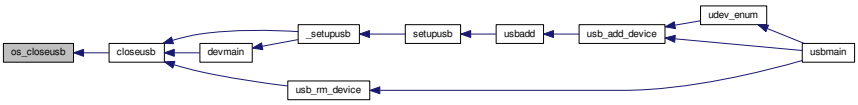
References kbsyspath, keyboard, and usbunclaim().

Referenced by closeusb().

```
423                                {
424     if(kb->handle){
425         usbunclaim(kb, 0);
426         close(kb->handle - 1);
427     }
428     if(kb->udev)
429         udev_device_unref(kb->udev);
430     kb->handle = 0;
431     kb->udev = 0;
432     kbsyspath[INDEX_OF(kb, keyboard)][0] = 0;
433 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



3.5.3.3 os_inputmain()

```
void* os_inputmain (
            void * context )
```

os_inputmain is run in a separate thread and will be detached from the main thread, so it needs to clean up its own resources.

Todo This function is a collection of many tasks. It should be divided into several sub-functions for the sake of greater convenience:

1. set up an URB (Userspace Ressource Buffer) to communicate with the USBDEVFS_∗ ioctl()s
2. perform the ioctl()
3. interpretate the information got into the URB buffer or handle error situations and retry operation or leave the endless loop
4. inform the os about the data
5. loop endless via 2.
6. if endless loop has gone, deinitalize the interface, free buffers etc.
7. return null

Here the actions in detail:

Monitor input transfers on all endpoints for non-RGB devices For RGB, monitor all but the last, as it's used for input/output

Get an usbdevfs_urb data structure and clear it via memset()

Hopefully the buffer lengths are equal for all devices with congruent types. You can find out the correctness for your device with lsusb –v or similar on macOS. Currently the following combinations are known and implemented:

device	detect with macro combination	endpoint #	buffer-length
each	none	0	8
RGB Mouse	IS_RGB && IS_MOUSE	1	10
RGB Keyboard	IS_RGB && !IS_MOUSE	1	21
RGB Mouse or Keyboard	IS_RGB	2	MSG_SIZE (64)
non RGB Mouse or Keyboard	!IS_RGB	1	4
non RGB Mouse or Keyboard	!IS_RGB	2	15

Now submit all the URBs via ioctl(USBDEVFS_SUBMITURB) with type USBDEVFS_URB_TYPE_INTERRUPT (the endpoints are defined as type interrupt). Endpoint number is 0x80..0x82 or 0x83, depending on the model.

The userSpaceFS knows the URBs now, so start monitoring input

if the ioctl returns something != 0, let's have a deeper look what happened. Broken devices or shutting down the entire system leads to closing the device and finishing this thread.

If just an EPIPE ocurred, give the device a CLEAR_HALT and resubmit the URB.

A correct REAPURB returns a Pointer to the URB which we now have a closer look into. Lock all following actions with imutex.

Process the input depending on type of device. Interprete the actual size of the URB buffer

device	detect with macro combination	seems to be endpoint #	actual buffer-length	function called
mouse (RGB and non RGB)	IS_MOUSE	nA	8, 10 or 11	hid_mouse_↩ translate()
mouse (RGB and non RGB)	IS_MOUSE	nA	MSG_SIZE (64)	corsair_↩ mousecopy()
RGB Keyboard	IS_RGB && !IS_↩ MOUSE	1	8 (BIOS Mode)	hid_kb_translate()

device	detect with macro combination	seems to be endpoint #	actual buffer-length	function called
RGB Keyboard	IS_RGB && !IS_↩ MOUSE	2	5 or 21, KB inactive!	hid_kb_translate()
RGB Keyboard	IS_RGB && !IS_↩ MOUSE	3?	MSG_SIZE	corsair_kbcopy()
non RGB Keyboard	!IS_RGB && !IS_↩ MOUSE	nA	nA	hid_kb_translate()

The input data is transformed and copied to the kb structure. Now give it to the OS and unlock the imutex afterwards.

Re-submit the URB for the next run.

If the endless loop is terminated, clean up by discarding the URBs via ioctl(USBDEVFS_DISCARDURB), free the URB buffers and return a null pointer as thread exit code.

Definition at line 226 of file usb_linux.c.

References IS_MOUSE, IS_RGB, and keyboard.

Referenced by _setupusb().

```
226                                 {
227     usbdevice* kb = context;
228     int fd = kb->handle - 1;
229     short vendor = kb->vendor, product = kb->product;
230     int index = INDEX_OF(kb, keyboard);
231     ckb_info("Starting input thread for %s%d\n", devpath, index);
232 
237     int urbcount = IS_RGB(vendor, product) ? (kb->epcount - 1) : kb->epcount;
238 
240     struct usbdevfs_urb urbs[urbcount];
241     memset(urbs, 0, sizeof(urbs));
242 
256     urbs[0].buffer_length = 8;
257     if(IS_RGB(vendor, product)){
258         if(IS_MOUSE(vendor, product))
259             urbs[1].buffer_length = 10;
260         else
261             urbs[1].buffer_length = 21;
262         urbs[2].buffer_length = MSG_SIZE;
263         if(urbcount != 3)
264             urbs[urbcount - 1].buffer_length = MSG_SIZE;
265     } else {
266         urbs[1].buffer_length = 4;
267         urbs[2].buffer_length = 15;
268     }
269 
272     for(int i = 0; i < urbcount; i++){
273         urbs[i].type = USBDEVFS_URB_TYPE_INTERRUPT;
274         urbs[i].endpoint = 0x80 | (i + 1);
275         urbs[i].buffer = malloc(urbs[i].buffer_length);
276         int rv = ioctl(fd, USBDEVFS_SUBMITURB, urbs + i);
277         if (rv < 0) ckb_warn("os_inputmain 1: got retcode %d and errno = %d, %s\n", rv, errno, strerror(
    errno));
278     }
279 
281     while (1) {
282         struct usbdevfs_urb* urb = 0;
283         int rv;
284 
287         if ((rv = ioctl(fd, USBDEVFS_REAPURB, &urb))) {
288             ckb_warn("os_inputmain 2: got retcode %d and errno = %d, %s\n", rv, errno, strerror(errno));
289             if (errno == ENODEV || errno == ENOENT || errno == ESHUTDOWN)
290                 // Stop the thread if the handle closes
291                 break;
292             else if(errno == EPIPE && urb){
294                 rv = ioctl(fd, USBDEVFS_CLEAR_HALT, &urb->endpoint);
295                 ckb_warn("os_inputmain 3: got retcode %d and errno = %d, %s\n", rv, errno, strerror(errno))
    ;
296                 // Re-submit the URB
297                 if(urb)
298                     rv = ioctl(fd, USBDEVFS_SUBMITURB, urb);
299                     if (rv < 0) ckb_warn("os_inputmain 4: got retcode %d and errno = %d, %s\n", rv, errno,
```

```
    strerror(errno));
300                 urb = 0;
301             }
302         }
303 
307         if (urb) {
319             pthread_mutex_lock(imutex(kb));
320             if(IS_MOUSE(vendor, product)){
321                 switch(urb->actual_length){
322                 case 8:
323                 case 10:
324                 case 11:
325                     // HID mouse input
326                     hid_mouse_translate(kb->input.keys, &kb->input.rel_x, &kb->input.rel_y, -(urb->endpoint
     & 0xF), urb->actual_length, urb->buffer);
327                     break;
328                 case MSG_SIZE:
329                     // Corsair mouse input
330                     corsair_mousecopy(kb->input.keys, -(urb->endpoint & 0xF), urb->buffer);
331                     break;
332                 }
333             } else if(IS_RGB(vendor, product)){
334                 switch(urb->actual_length){
335                 case 8:
336                     // RGB EP 1: 6KRO (BIOS mode) input
337                     hid_kb_translate(kb->input.keys, -1, urb->actual_length, urb->buffer);
338                     break;
339                 case 21:
340                 case 5:
341                     // RGB EP 2: NKRO (non-BIOS) input. Accept only if keyboard is inactive
342                     if(!kb->active)
343                         hid_kb_translate(kb->input.keys, -2, urb->actual_length, urb->buffer);
344                     break;
345                 case MSG_SIZE:
346                     // RGB EP 3: Corsair input
347                     corsair_kbcopy(kb->input.keys, -(urb->endpoint & 0xF), urb->buffer);
348                     break;
349                 }
350             } else {
351                 // Non-RGB input
352                 hid_kb_translate(kb->input.keys, urb->endpoint & 0xF, urb->actual_length, urb->buffer);
353             }
356             inputupdate(kb);
357             pthread_mutex_unlock(imutex(kb));
359             rv = ioctl(fd, USBDEVFS_SUBMITURB, urb);
360             if (rv < 0) ckb_warn("os_inputmain 5: got retcode %d and errno = %d, %s\n", rv, errno, strerror
    (errno));
361             urb = 0;
362         }
363     }
364 
368     ckb_info("Stopping input thread for %s%d\n", devpath, index);
369     for(int i = 0; i < urbcount; i++){
370         ioctl(fd, USBDEVFS_DISCARDURB, urbs + i);
371         free(urbs[i].buffer);
372     }
373     return 0;
374 }
```

Here is the caller graph for this function:

3.5.3.4 os_resetusb()

```
int os_resetusb (
            usbdevice * kb,
```

```
        const char * file,
        int line )
```

os_resetusb is the os specific implementation for resetting usb

Try to reset an usb device in a linux user space driver.

1. unclaim the device, but do not reconnect the system driver (second param resetting = true)
2. reset the device via USBDEVFS_RESET command
3. claim the device again. Returns 0 on success, -2 if device should be removed and -1 if reset should by tried again

Todo it seems that no one wants to try the reset again. But I'v seen it somewhere...

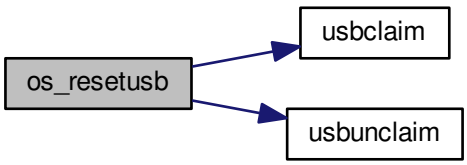
Definition at line 485 of file usb_linux.c.

References TEST_RESET, usbclaim(), and usbunclaim().

Referenced by _resetusb().

```
485                                                     {
486     ckb_warn("os_resetusb: resetting %s\n", kb->name);
487     TEST_RESET(usbunclaim(kb, 1));
488     TEST_RESET(ioctl(kb->handle - 1, USBDEVFS_RESET));
489     TEST_RESET(usbclaim(kb));
490     // Success!
491     return 0;
492 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

3.5.3.5 os_sendindicators()

```
void os_sendindicators (
            usbdevice * kb )
```

os_sendindicators update the indicators for the special keys (Numlock, Capslock and what else?)

os_sendindicators update the indicators for the special keys (Numlock, Capslock and what else?)

Read the data from kb->ileds ans send them via ioctl() to the keyboard.

bRequest↩ Type	bRequest	wValue	EP	size	Timeout	data
0x21	0x09	0x0200	Interface 0	MSG_SIZE 1 Byte	timeout 0,5ms	the message buffer pointer
Host to Device, Type=Class, Recipient=Interface (why not endpoint?)	9 = SEND?	specific	0	1	500	struct* kb->ileds

The ioctl command is USBDEVFS_CONTROL.

Definition at line 206 of file usb_linux.c.

```
206                                     {
207     struct usbdevfs_ctrltransfer transfer = { 0x21, 0x09, 0x0200, 0x00, 1, 500, &kb->ileds };
208     int res = ioctl(kb->handle - 1, USBDEVFS_CONTROL, &transfer);
209     if(res <= 0)
210         ckb_err("%s\n", res ? strerror(errno) : "No data written");
211 }
```

3.5.3.6 os_setupusb()

```
int os_setupusb (
            usbdevice * kb )
```

os_setupusb OS-specific setup for a specific usb device.

Perform the operating system-specific opening of the interface in os_setupusb(). As a result, some parameters should be set in kb (name, serial, fwversion, epcount = number of usb endpoints), and all endpoints should be claimed with usbclaim(). Claiming is the only point where os_setupusb() can produce an error (-1).

- Copy device description and serial
- Copy firmware version (needed to determine USB protocol)
- Do some output about connecting interfaces
- Claim the USB interfaces

Todo in these modules a pullrequest is outstanding

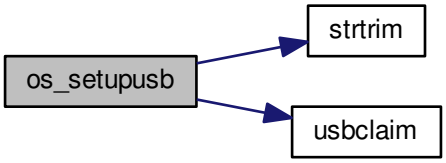
Definition at line 524 of file usb_linux.c.

References keyboard, strtrim(), and usbclaim().

Referenced by _setupusb().

```
524                                {
527     struct udev_device* dev = kb->udev;
528     const char* name = udev_device_get_sysattr_value(dev, "product");
529     if(name)
530         strncpy(kb->name, name, KB_NAME_LEN);
531     strtrim(kb->name);
532     const char* serial = udev_device_get_sysattr_value(dev, "serial");
533     if(serial)
534         strncpy(kb->serial, serial, SERIAL_LEN);
535     strtrim(kb->serial);
538     const char* firmware = udev_device_get_sysattr_value(dev, "bcdDevice");
539     if(firmware)
540         sscanf(firmware, "%hx", &kb->fwversion);
541     else
542         kb->fwversion = 0;
543     int index = INDEX_OF(kb, keyboard);
546     ckb_info("Connecting %s at %s%d\n", kb->name, devpath, index);
547
553     const char* ep_str = udev_device_get_sysattr_value(dev, "bNumInterfaces");
554     kb->epcount = 0;
555     if(ep_str)
556         sscanf(ep_str, "%d", &kb->epcount);
557     if(kb->epcount == 0){
558         // This shouldn't happen, but if it does, assume EP count based on what the device is supposed to
    have
559         kb->epcount = (HAS_FEATURES(kb, FEAT_RGB) ? 4 : 3);
560         ckb_warn("Unable to read endpoint count from udev, assuming %d...\n", kb->epcount);
561     }
562     if(usbclaim(kb)){
563         ckb_err("Failed to claim interfaces: %s\n", strerror(errno));
564         return -1;
565     }
566     return 0;
567 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

3.5.3.7 os_usbrecv()

```
int os_usbrecv (
            usbdevice * kb,
            uchar * in_msg,
            const char * file,
            int line )
```

os_usbrecv does what its name says:

The comment at the beginning of the procedure causes the suspicion that the firmware versionspecific distinction is missing for receiving from usb endpoint 3 or 4. The commented code contains only the reception from EP4, but this may be wrong for a software version 2.0 or higher (see the code for os-usbsend ()).

So all the receiving is done via an ioctl() like in os_usbsend. The ioctl() is given a struct usbdevfs_ctrltransfer, in which the relevant parameters are entered:

bRequest↩ Type	bRequest	wValue	EP	size	Timeout	data
0xA1	0x01	0x0200	endpoint to be addressed from epcount - 1	MSG_SIZE	5ms	the message buffer pointer
Device to Host, Type=Class, Recipi- ent=Interface	1 = RECEIVE?	specific	Interface #	64	5000	in_msg

The ioctl() returns the number of bytes received. Here is the usual check again:

- If the return value is -1 AND the error is a timeout (ETIMEOUT), os_usbrecv() will return -1 to indicate that it is probably a recoverable problem and a retry is recommended.
- For another negative value or other error identifier OR 0 bytes are received, 0 is returned as an identifier for a heavy error.
- In all other cases, the function returns the number of bytes received.

If this is not the entire blocksize (MSG_SIZE bytes), an error message is issued on the standard error channel [warning "Read YY bytes (expected 64)"].

Definition at line 127 of file usb_linux.c.

Referenced by _usbrecv().

```
127                                                                         {
128     int res;
129     // This is what CUE does, but it doesn't seem to work on linux.
130     /*if(kb->fwversion >= 0x130){
131         struct usbdevfs_bulktransfer transfer;
132         memset(&transfer, 0, sizeof(transfer));
133         transfer.ep = 0x84;
134         transfer.len = MSG_SIZE;
135         transfer.timeout = 5000;
136         transfer.data = in_msg;
137         res = ioctl(kb->handle - 1, USBDEVFS_BULK, &transfer);
138     } else {*/
139         struct usbdevfs_ctrltransfer transfer = { 0xa1, 0x01, 0x0300, kb->epcount - 1, MSG_SIZE, 5000,
    in_msg };
140         res = ioctl(kb->handle - 1, USBDEVFS_CONTROL, &transfer);
```

```
141     //}
142     if(res <= 0){
143         ckb_err_fn("%s\n", file, line, res ? strerror(errno) : "No data read");
144         if(res == -1 && errno == ETIMEDOUT)
145             return -1;
146         else
147             return 0;
148     } else if(res != MSG_SIZE)
149         ckb_warn_fn("Read %d bytes (expected %d)\n", file, line, res, MSG_SIZE);
150     return res;
151 }
```

Here is the caller graph for this function:

3.5.3.8 os_usbsend()

```
int os_usbsend (
            usbdevice * kb,
            const uchar * out_msg,
            int is_recv,
            const char * file,
            int line )
```

os_usbsend has two functions:

- if is_recv == false, it tries to send a given MSG_SIZE buffer via the usb interface given with kb.
- otherwise a request is sent via the usb device to initiate the receiving of a message from the remote device.

The functionality for sending distinguishes two cases, depending on the version number of the firmware of the connected device:
If the firmware is less or equal 1.2, the transmission is done via an ioctl(). The ioctl() is given a struct usbdevfs_↩ ctrltransfer, in which the relevant parameters are entered:

bRequest↩ Type	bRequest	wValue	EP	size	Timeout	data
0x21	0x09	0x0200	endpoint / IF to be addressed from epcount-1	MSG_SIZE	5000 (=5ms)	the message buffer pointer
Host to Device, Type=Class, Recipi-ent=Interface	9 = Send data?	specific	last or pre-last device #	64	5000	out_msg

The ioctl command is USBDEVFS_CONTROL.

The same constellation is used if the device is requested to send its data (is_recv = true).

For a more recent firmware and is_recv = false, the ioctl command USBDEVFS_CONTROL is not used (this tells the bus to enter the control mode), but the bulk method is used: USBDEVFS_BULK. This is astonishing, because all of the endpoints are type Interrupt, not bulk.

Anyhow, forthis purpose a different structure is used for the ioctl() (struct **usbdevfs_bulktransfer**) and this is also initialized differently:
The length and timeout parameters are given the same values as above. The formal parameter out_msg is also passed as a buffer pointer. For the endpoints, the firmware version is differentiated again:
For a firmware version between 1.3 and <2.0 endpoint 4 is used, otherwise (it can only be >=2.0) endpoint 3 is used.

Todo Since the handling of endpoints has already led to problems elsewhere, this implementation is extremely hardware-dependent and critical!
Eg. the new keyboard K95PLATINUMRGB has a version number significantly less than 2.0 - will it run with this implementation?

The ioctl() - no matter what type - returns the number of bytes sent. Now comes the usual check:

- If the return value is -1 AND the error is a timeout (ETIMEOUT), os_usbsend() will return -1 to indicate that it is probably a recoverable problem and a retry is recommended.
- For another negative value or other error identifier OR 0 bytes sent, 0 is returned as a heavy error identifier.
- In all other cases, the function returns the number of bytes sent.

If this is not the entire blocksize (MSG_SIZE bytes), an error message is issued on the standard error channel [warning "Wrote YY bytes (expected 64)"].

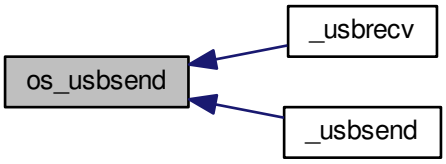
If DEBUG_USB is set during compilation, the number of bytes sent and their representation are logged to the error channel.

Definition at line 66 of file usb_linux.c.

Referenced by _usbrecv(), and _usbsend().

```
66                                                                                                  {
67     int res;
68     if(kb->fwversion >= 0x120 && !is_recv){
69         struct usbdevfs_bulktransfer transfer;
70         memset(&transfer, 0, sizeof(transfer));
71         transfer.ep = (kb->fwversion >= 0x130 && kb->fwversion < 0x200) ? 4 : 3;
72         transfer.len = MSG_SIZE;
73         transfer.timeout = 5000;
74         transfer.data = (void*)out_msg;
75         res = ioctl(kb->handle - 1, USBDEVFS_BULK, &transfer);
76     } else {
77         struct usbdevfs_ctrltransfer transfer = { 0x21, 0x09, 0x0200, kb->epcount - 1, MSG_SIZE, 5000, (
    void*)out_msg };
78         res = ioctl(kb->handle - 1, USBDEVFS_CONTROL, &transfer);
79     }
80     if(res <= 0){
81         ckb_err_fn("%s\n", file, line, res ? strerror(errno) : "No data written");
82         if(res == -1 && errno == ETIMEDOUT)
83             return -1;
84         else
85             return 0;
86     } else if(res != MSG_SIZE)
87         ckb_warn_fn("Wrote %d bytes (expected %d)\n", file, line, res, MSG_SIZE);
88 #ifdef DEBUG_USB
89     char converted[MSG_SIZE*3 + 1];
90     for(int i=0;i<MSG_SIZE;i++)
91         sprintf(&converted[i*3], "%02x ", out_msg[i]);
92     ckb_warn_fn("Sent %s\n", file, line, converted);
93 #endif
94     return res;
95 }
```

Here is the caller graph for this function:



3.5.3.9 strtrim()

```
void strtrim (
            char * string )
```

strtrim trims a string by removing leading and trailing spaces.

Parameters

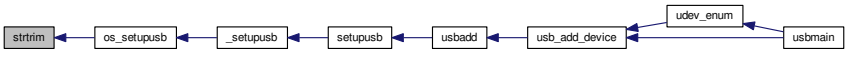
string	

Definition at line 499 of file usb_linux.c.

Referenced by os_setupusb().

```
499                         {
500     // Find last non-space
501     char* last = string;
502     for(char* c = string; *c != 0; c++){
503         if(!isspace(*c))
504             last = c;
505     }
506     last[1] = 0;
507     // Find first non-space
508     char* first = string;
509     for(; *first != 0; first++){
510         if(!isspace(*first))
511             break;
512     }
513     if(first != string)
514         memmove(string, first, last - first);
515 }
```

Here is the caller graph for this function:



3.5.3.10 udev_enum()

```
static void udev_enum ( )  [static]
```

Reduce the hits of the enumeration by limiting to usb as technology and corsair as idVendor. Then filter with udev↩
_enumerate_scan_devices () all hits.

The following call to udev_enumerate_get_list_entry() fetches the entire hitlist as udev_list_entry ∗.
Use udev_list_entry_foreach() to iterate through the hit set.
If both the device name exists (udev_list_entry_get_name) and the subsequent creation of a new udev_device (udev_device_new_from_syspath) is ok, the new device is added to the list with usb_add_device().

If the latter does not work, the new device is released again (udev_device_unref ()).
After the last iteration, the enumerator is released with udev_enumerate_unref ().

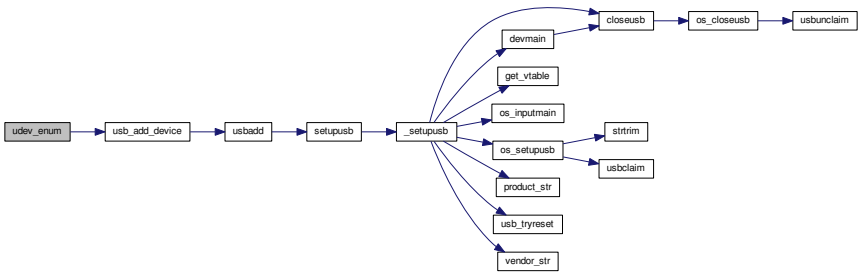
Definition at line 724 of file usb_linux.c.

References udev, usb_add_device(), and V_CORSAIR_STR.

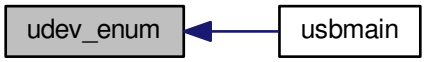
Referenced by usbmain().

```
724                         {
725     struct udev_enumerate* enumerator = udev_enumerate_new(udev);
726     udev_enumerate_add_match_subsystem(enumerator, "usb");
727     udev_enumerate_add_match_sysattr(enumerator, "idVendor", V_CORSAIR_STR);
728     udev_enumerate_scan_devices(enumerator);
729     struct udev_list_entry* devices, *dev_list_entry;
730     devices = udev_enumerate_get_list_entry(enumerator);
731
732     udev_list_entry_foreach(dev_list_entry, devices){
733         const char* path = udev_list_entry_get_name(dev_list_entry);
734         if(!path)
735             continue;
736         struct udev_device* dev = udev_device_new_from_syspath(udev, path);
737         if(!dev)
738             continue;
739         // If the device matches a recognized device ID, open it
740         if(usb_add_device(dev))
741             // Release device if not
742             udev_device_unref(dev);
743     }
744     udev_enumerate_unref(enumerator);
745 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.5.3.11 usb_add_device()

```
static int usb_add_device (
            struct udev_device * dev )  [static]
```

Parameters

dev	the functions usb_*_device get a struct udev* with the neccessary hardware-related information.

Returns

the retval of usbadd(): 0 if device was recognized/added or 1 if either vendor is not corsair or product is not mentioned in model[].

First get the idVendor via udev_device_get_sysattr_value(). If this is equal to the ID-string of corsair ("1b1c"), get the idProduct on the same way.
If we can find the model name in the model array, call usbadd() with the model number.

Todo So why the hell not a transformation between the string and the short presentation? Lets check if the string representation is used elsewhere.

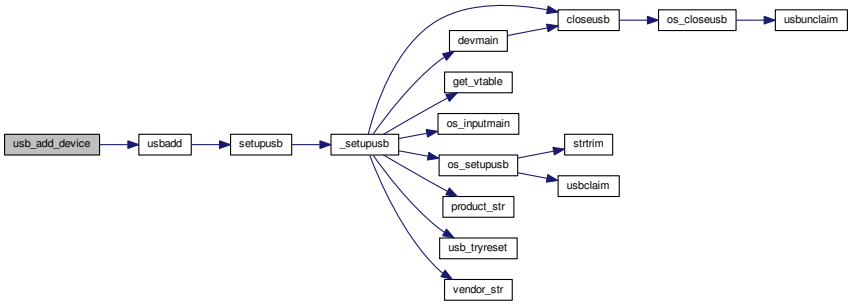
Definition at line 672 of file usb_linux.c.

References N_MODELS, usbadd(), V_CORSAIR, and V_CORSAIR_STR.

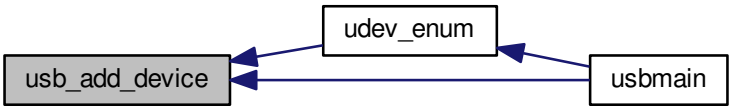
Referenced by udev_enum(), and usbmain().

```
672                                                 {
673     const char* vendor = udev_device_get_sysattr_value(dev, "idVendor");
674     if(vendor && !strcmp(vendor, V_CORSAIR_STR)){
675         const char* product = udev_device_get_sysattr_value(dev, "idProduct");
676         if(product){
677             for(_model* model = models; model < models +
      N_MODELS; model++){
678                 if(!strcmp(product, model->name)){
679                     return usbadd(dev, V_CORSAIR, model->number);
680                 }
681             }
682         }
683     }
684     return 1;
685 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.5.3.12 usb_rm_device()

```
static void usb_rm_device (
            struct udev_device * dev )  [static]
```

Parameters

dev	the functions usb_*_device get a struct udev* with the neccessary hardware-related information.

First try to find the system path of the device given in parameter dev. The index where the name is found is the same index we need to address the global keyboard array. That array holds all usbdevices.
Searching for the correct name in kbsyspath-array and closing the usb via closeusb() are protected by lock..unlock of the corresponding devmutex arraymember.

Definition at line 697 of file usb_linux.c.

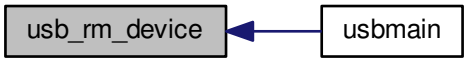
References closeusb(), devmutex, kbsyspath, and keyboard.

Referenced by usbmain().

```
697                                                 {
698     // Device removed. Look for it in our list of keyboards
699     const char* syspath = udev_device_get_syspath(dev);
700     if(!syspath || syspath[0] == 0)
701         return;
702     for(int i = 1; i < DEV_MAX; i++){
703         pthread_mutex_lock(devmutex + i);
704         if(!strcmp(syspath, kbsyspath[i]))
705             closeusb(keyboard + i);
706         pthread_mutex_unlock(devmutex + i);
707     }
708 }
```

Here is the call graph for this function:

Here is the caller graph for this function:



3.5.3.13 usbadd()

```
int usbadd (
              struct udev_device * dev,
              short vendor,
              short product )
```

Definition at line 569 of file usb_linux.c.

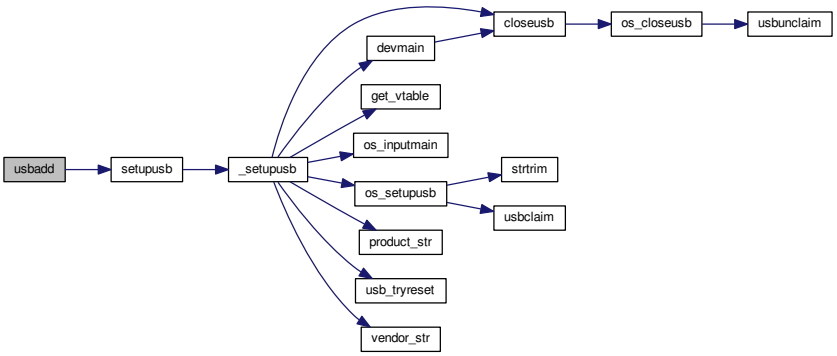
References kbsyspath, keyboard, and setupusb().

Referenced by usb_add_device().

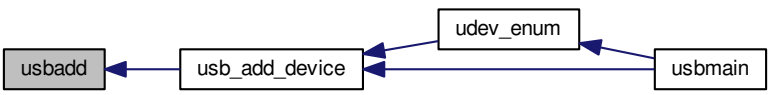
```
569                                                                 {
570     const char* path = udev_device_get_devnode(dev);
571     const char* syspath = udev_device_get_syspath(dev);
572     if(!path || !syspath || path[0] == 0 || syspath[0] == 0){
573         ckb_err("Failed to get device path\n");
574         return -1;
575     }
576     // Find a free USB slot
577     for(int index = 1; index < DEV_MAX; index++){
578         usbdevice* kb = keyboard + index;
```

```
579         if(pthread_mutex_trylock(dmutex(kb))){
580             // If the mutex is locked then the device is obviously in use, so keep going
581             if(!strcmp(syspath, kbsyspath[index])){
582                 // Make sure this existing keyboard doesn't have the same syspath (this shouldn't happen)
583                 return 0;
584             }
585             continue;
586         }
587         if(!IS_CONNECTED(kb)){
588             // Open the sysfs device
589             kb->handle = open(path, O_RDWR) + 1;
590             if(kb->handle <= 0){
591                 ckb_err("Failed to open USB device: %s\n", strerror(errno));
592                 kb->handle = 0;
593                 pthread_mutex_unlock(dmutex(kb));
594                 return -1;
595             } else {
596                 // Set up device
597                 kb->udev = dev;
598                 kb->vendor = vendor;
599                 kb->product = product;
600                 strncpy(kbsyspath[index], syspath, FILENAME_MAX);
601                 // Mutex remains locked
602                 setupusb(kb);
603                 return 0;
604             }
605         }
606         pthread_mutex_unlock(dmutex(kb));
607     }
608     ckb_err("No free devices\n");
609     return -1;
610 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.5.3.14 usbclaim()

```
static int usbclaim (
            usbdevice * kb )  [static]
```

usbclaim does claiming all EPs for the usb device gicen by kb.

Parameters

kb	THE usbdevice*

Returns

0 on success, -1 otherwise.

Claim all endpoints for a given device (remeber the decrementing of the file descriptor) via ioctl(USBDEVFS_DIS↩ CONNECT) and ioctl(USBDEVFS_CLAIMINTERFACE).

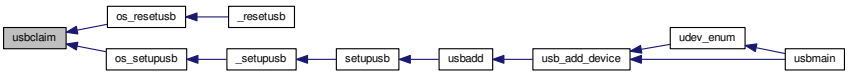
Error handling is done for the ioctl(USBDEVFS_CLAIMINTERFACE) only. If this fails, now an error message is thrown and -1 is returned. Function is called in usb_linux.c only, so it is declared as static now.

Definition at line 447 of file usb_linux.c.

Referenced by os_resetusb(), and os_setupusb().

```
447                                 {
448     int count = kb->epcount;
449     for (int i = 0; i < count; i++) {
450         int rv;
451         struct usbdevfs_ioctl ctl = { i, USBDEVFS_DISCONNECT, 0 };
452         if ((rv = ioctl(kb->handle - 1, USBDEVFS_IOCTL, &ctl))) {
453             ckb_err("usbclaim 1: got retcode %d and errno = %d, %s. Ignoring.\n", rv, errno, strerror(errno
    ));
454         }
455 
456         if ((rv = ioctl(kb->handle - 1, USBDEVFS_CLAIMINTERFACE, &i))) {
457             ckb_err("usbclaim 2: got retcode %d and errno = %d, %s. Aborting.\n", rv, errno, strerror(errno
    ));
458             return -1;
459         }
460     }
461     return 0;
462 }
```

Here is the caller graph for this function:



3.5.3.15 usbkill()

```
void usbkill ( )
```

Stop the USB system.

Definition at line 815 of file usb_linux.c.

References udev.

```
815                 {
816     udev_unref(udev);
817     udev = 0;
818 }
```

3.5.3.16 usbmain()

```
int usbmain ( )
```

Start the USB main loop. Returns program exit code when finished.

usbmain is called by main() after setting up all other stuff.

Returns

0 normally or -1 if fatal error occurs (up to now only if no new devices are available)

First check whether the uinput module is loaded by the kernel.

Todo Why isn't missing of uinput a fatal error?

Create the udev object with udev_new() (is a function from libudev.h) terminate -1 if error

Enumerate all currently connected devices

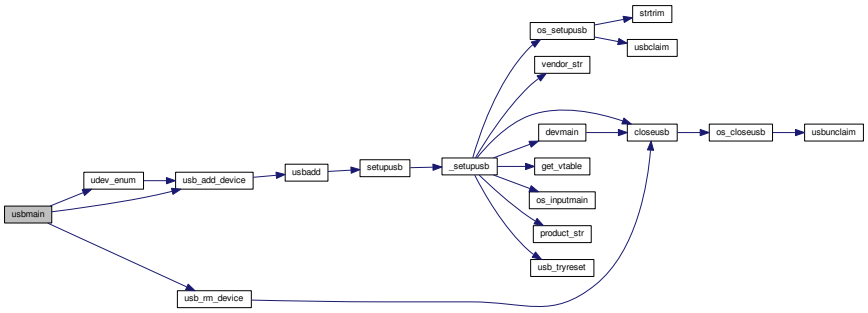
Todo lae. here the work has to go on...

Definition at line 752 of file usb_linux.c.

References udev, udev_enum(), usb_add_device(), and usb_rm_device().

```
752             {
757     // Load the uinput module (if it's not loaded already)
758     if(system("modprobe uinput") != 0)
759         ckb_warn("Failed to load uinput module\n");
760 
764     if(!(udev = udev_new())) {
765         ckb_fatal("Failed to initialize udev in usbmain(), usb_linux.c\n");
766         return -1;
767     }
768 
771     udev_enum();
772 
775     // Done scanning. Enter a loop to poll for device updates
776     struct udev_monitor* monitor = udev_monitor_new_from_netlink(udev, "udev");
777     udev_monitor_filter_add_match_subsystem_devtype(monitor, "usb", 0);
778     udev_monitor_enable_receiving(monitor);
779     // Get an fd for the monitor
780     int fd = udev_monitor_get_fd(monitor);
781     fd_set fds;
782     while(udev){
783         FD_ZERO(&fds);
784         FD_SET(fd, &fds);
785         // Block until an event is read
786         if(select(fd + 1, &fds, 0, 0, 0) > 0 && FD_ISSET(fd, &fds)){
787             struct udev_device* dev = udev_monitor_receive_device(monitor);
788             if(!dev)
789                 continue;
790             const char* action = udev_device_get_action(dev);
791             if(!action){
792                 udev_device_unref(dev);
793                 continue;
794             }
795             // Add/remove device
796             if(!strcmp(action, "add")){
797                 int res = usb_add_device(dev);
798                 if(res == 0)
799                     continue;
800                 // If the device matched but the handle wasn't opened correctly, re-enumerate (this
    sometimes solves the problem)
801                 if(res == -1)
802                     udev_enum();
803             } else if(!strcmp(action, "remove"))
804                 usb_rm_device(dev);
805             udev_device_unref(dev);
806         }
807     }
808     udev_monitor_unref(monitor);
809     return 0;
810 }
```

Here is the call graph for this function:



3.5.3.17 usbunclaim()

```
static int usbunclaim (
            usbdevice * kb,
            int resetting )  [static]
```

usbunclaim do an unclaiming of the usb device gicen by kb.

Parameters

kb	THE usbdevice*
resetting	boolean flag: If resseting is true, the caller will perform a bus reset command after unclaiming the device.

Returns

always 0.

Unclaim all endpoints for a given device (remeber the decrementing of the file descriptor) via ioctl(USBDEVFS_↩ DISCARDURB).

Afterwards - if ressetting is false - do a USBDEVFS_CONNECT for EP 0 and 1. If it is a non RGB device, connect EP 2 also. The comment mentions RGB keyboards only, but as I understand the code, this is valid also for RGB mice.

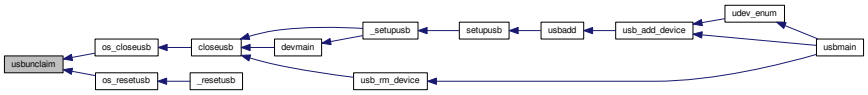
There is no error handling yet. Function is called in usb_linux.c only, so it is declared as static now.

Definition at line 393 of file usb_linux.c.

Referenced by os_closeusb(), and os_resetusb().

```
393                                                 {
394     int handle = kb->handle - 1;
395     int count = kb->epcount;
396     ckb_warn("usbunclaim called,%s resetting\n", (resetting)? "" : "not");
397     for (int i = 0; i < count; i++) {
398         ioctl(handle, USBDEVFS_RELEASEINTERFACE, &i);
399     }
400     // For RGB keyboards, the kernel driver should only be reconnected to interfaces 0 and 1 (HID), and
      only if we're not about to do a USB reset.
401     // Reconnecting any of the others causes trouble.
402     if (!resetting) {
403         struct usbdevfs_ioctl ctl = { 0, USBDEVFS_CONNECT, 0 };
404         ioctl(handle, USBDEVFS_IOCTL, &ctl);
405         ctl.ifno = 1;
406         ioctl(handle, USBDEVFS_IOCTL, &ctl);
407         // Also reconnect iface #2 (HID) for non-RGB keyboards
408         if(!HAS_FEATURES(kb, FEAT_RGB)){
409             ctl.ifno = 2;
410             ioctl(handle, USBDEVFS_IOCTL, &ctl);
411         }
412     }
413     return 0;
414 }
```

Here is the caller graph for this function:



3.5.4 Variable Documentation

3.5.4.1 kbsyspath

```
char kbsyspath[9][FILENAME_MAX]  [static]
```

Definition at line 11 of file usb_linux.c.

Referenced by os_closeusb(), usb_rm_device(), and usbadd().

3.5.4.2 models

```
_model models[]  [static]
```

Initial value:

```
= {

    {  "1b17" ,  0x1b17  },
    {  "1b07" ,  0x1b07  },
    {  "1b37" ,  0x1b37  },
    {  "1b39" ,  0x1b39  },
    {  "1b13" ,  0x1b13  },
    {  "1b09" ,  0x1b09  },
    {  "1b33" ,  0x1b33  },
    {  "1b36" ,  0x1b36  },
    {  "1b38" ,  0x1b38  },
    {  "1b3a" ,  0x1b3a  },
    {  "1b11" ,  0x1b11  },
    {  "1b08" ,  0x1b08  },
    {  "1b2d" ,  0x1b2d  },
    {  "1b20" ,  0x1b20  },
    {  "1b15" ,  0x1b15  },

    {  "1b12" ,  0x1b12  },
    {  "1b2e" ,  0x1b2e  },
    {  "1b14" ,  0x1b14   },
    {  "1b19" ,  0x1b19   },
    {  "1b2f" ,  0x1b2f   },
    {  "1b1e" ,  0x1b1e  },
    {  "1b3e" ,  0x1b3e  },
    {  "1b32" ,  0x1b32   },
    {  "1b3c" ,  0x1b3c   }
}
```

Attention

when adding new hardware this file hat to be changed too.

In this structure array *models*[] for each device the name (the device id as string in hex without leading 0x) and its usb device id as short must be entered in this array.

Definition at line 631 of file usb_linux.c.

3.5.4.3 udev

```
struct udev* udev  [static]
```

Definition at line 614 of file usb_linux.c.

Referenced by udev_enum(), usbkill(), and usbmain().

3.5.4.4 udevthread

```
pthread_t udevthread
```

Definition at line 617 of file usb_linux.c.

3.5.4.5 usbthread

```
pthread_t usbthread
```

Todo These two thread vasriables seem to be unused: usbtread, udevthread

Definition at line 617 of file usb_linux.c.

Index