Controlling EIB/KNX Devices from Linux using USB

Heinz W. Werntges
University of Applied Sciences Wiesbaden

Jens Neumann and Vladimir Vinarski
Loewe Opta GmbH, Kompetenzzentrum Hannover

Introduction

Why USB?
Why Linux?
About the authors

Introduction

- About the authors
  - Jens Neumann & Vladimir Vinarski
    - Authors of the Open Source tool „Linux EIB Home Server“
      - See http://sourceforge.net/projects/eibcontrol
    - Participation in EMBASSI project
    - Affiliation: Loewe Opta Competence Center Hannover

  - Heinz Werntges
    - Teaches computer science at FH Wiesbaden since 2003
    - Background: Physics + commercial IT
    - New to EIB/KNX: First contact about 1 year ago
    - Motivation: Make EIB/KNX more popular and accessible, especially for the Linux community

Introduction

Konnex Association

KNX access via	Windows	Linux
RS.232	Available	Available
IP	Available	Available
USB	Available	Missing

Task:
Fill this gap!

Introduction

- Why USB instead of RS.232 ?
  - PC view:
    - RS.232 is „legacy hardware“, gets replaced
    - Notebook PCs: Rarely equipped with RS.232 anymore
  - EIB/KNX view:
    - RS.232 transmissions are slower than USB and are sometimes riddled with timing issues
    - Special FT1.2 protocol required
    - USB downside: USB adds complexity
  - Why not skip USB and go right for IP?
    - Cost issues

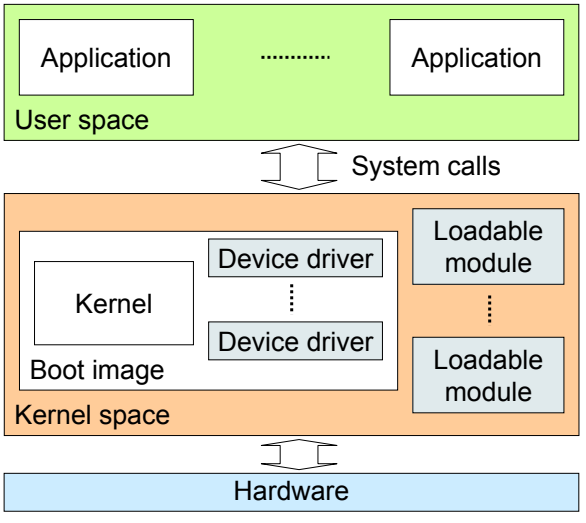
Introduction

- Why use Linux instead of Windows?
  - It is **Open Source**
    - Adapt it to your particular project‘s / product‘s needs
    - Well-established for (not-too-small) embedded systems
    - Royalty-free!
  - It is **powerful**
    - Suffices for most practical purposes
    - Real-time options available, strong on networks (IP !), …
    - Very reliable, good success record on servers
  - Because **it‘s there**…
    - Don‘t miss the Linux community and its many technology „addicts“
    - Well-established in academia, and …
    - … it is always good to have an alternative.

Background on Linux and USB

Background

- Linux: Kernel, device drivers, and modules



E.g., the **Linux EIB Home Server**

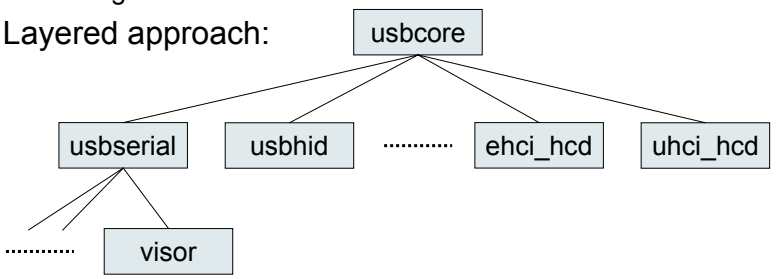
Q: How do we „talk“ to KNX devices?

Q: New module needed?

E.g., the **KNX USB Interface**

Background

- Linux: The USB module family
  - USB devices
    - Broad variety, both block and character devices
    - Kernel 2.6.12.1: **70** USB modules available
    - „Families“: **`atm, class, core, host, image, input, media, misc, mon, net, serial, storage`**
    - Plug-and-play option requires dynamic driver and device management
  - Layered approach:



The KNX USB Interface

The KNX USB Interface

- Specifications
  - AN037: KNX on USB
    - Protocol Specification & KNX USB Interface Device Requirements (2003.07.07)
  - AN070: USB adaptations (2005.03.17)
- Key features
  - **HID** class (HID = human interface device)
  - Interrupt transfer mode for data exchange, both in & out
  - Max. 64 octets of data per transfer
  - Reserved usage page ID: Non-standard control page
    - The multitude of KNX controls is not modelled with USB means; instead, KNX frames are tunneled through HID reports
  - New protocol for tunneling, different from FT1.2

The KNX USB Interface

- USB HID support by Linux
  - Based on **`usbcore`** module
  - Part of **`input`** module family
  - Specialized modules (e.g. for keyboard, mouse) available for popular usage page IDs
  - Generic module **`hiddev`** available for other page IDs
- **Result 1:**

  `hiddev` is the appropriate driver for KNX USB interfaces

  ➔ Modern Linux kernels already support KNX USB devices

  ➔ KNX USB programming may be restricted to user space

  (no new „KNX/USB“ kernel module needed!)

  Well – so far for the theory…

Device discovery

Device discovery

- What happens when you plug in a KNX USB device?
  - The USB core system exchanges control information
  - It adds the device to the list of connected USB devices
  - It assigns the appropriate driver (major device number)
  - It loads the required module(s) into the kernel
  - It issues an available minor device number

  ➔ **Device path determined!**

Device discovery

- Example for KNX USB interfaces
  - **`usbcore`** detects a generic HID → **`hiddev`** needed
  - Major device no = 180, minor device no = 96+x
  - x=0 if this is the first generic HID (x=1 if second, …)

  → Resulting device path is **`/dev/usb/hiddev0`**

- Alternative approach: **`usbdevfs`**
  - Ex: Second device on bus of first USB controller:
    `/proc/bus/usb/001/002`

  (Not used here)

Device discovery

- Information about USB devices
  - Selection by vendor ID and product ID
    - All devices on the USB bus can also be searched automatically
    - Device selection can be based on vendor & product ID when those are known *a priori*.
  - Tools
    - **`lsusb`** Lists device descriptors + bus topology
    - **`usbview`** X front-end, similar
    - **`lsmod`** Lists loaded modules, incl. USB modules

Device discovery

- **`lsusb`** output for a KNX USB device:

First 16 of 106 lines

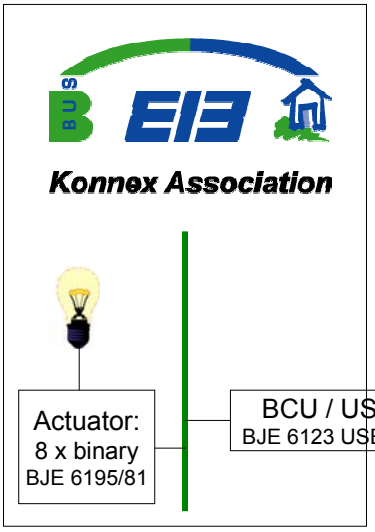
```
Bus 003 Device 002: ID 145c:1330
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               1.01
  bDeviceClass            0 (Defined at Interface level)
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0         8
  idVendor           0x145c
  idProduct          0x1330
  bcdDevice            1.01
  iManufacturer           1  Busch-Jaeger Elektro GmbH
  iProduct                2  KNX-USB Interface (Flush mounted)
  iSerial                 0
  bNumConfigurations      1
```

Use these parameters for device selection

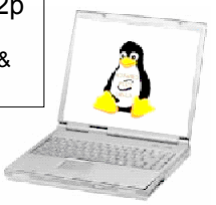
Test setup

Test setup

- Test equipment and procedure



IBM Thinkpad T42p
SuSE Linux 9.2
Kernels 2.6.8.14-24 &
2.6.12.1



Test sequences / reports:

1) ← **Device feature „get“ frame**
(Query supported EMI types)
2) → **Device feature „response“ frame**
3) ← **Device feature „set“ frame**
(Set active EMI type to EMI1)
4) ← **PC_SET_VAL.req**, tunnelled
(Activate Link Layer)
5) ← **L_DATA.req**, tunnelled
(Switch light on, group addr.=1/0/0)

Writing reports

Now that we know which device to open:
<u>How</u> do we write data to it?

<u>Three approaches:</u>

1) Windows (for comparison)
2) Linux, „libhid“
3) **Linux, standard** (system calls)

Writing reports

1) On Windows (for comparison)

- Device discovery
  - Complicated, based on vendor & product ID, well-documented, fixed procedure) → device handle
- Writing reports: Very simple – a single line!

```
WriteFile(handle, report_buffer,
                  64, &n_written, NULL);
```

Derived from Endpoint Descriptor (out):

```
wMaxPacketSize     0x0040  bytes 64 once
```


Writing reports

2) Linux, „libhid“ approach

- **Libhid**:
  - An Open Source library for accessing HIDs on Linux systems
  - Bypasses **`usbhid`** module by directly calling into **`usbcore`**
- Device discovery
  - By vendor ID and product ID + „path“ into the HID usage tree
- Writing reports
  - Convenient report exchange, similar to the Windows API

```
unsigned char const PATHLEN = 3;
int const PATH_OUT[] =
{ 0xffa00001, 0xffa00001, 0xffa10005 };
```

Usage page

Usage data

```
hid_set_output_report(
       handle, PATH_OUT, PATHLEN,
       report_buffer, 64);
```

Writing reports

3) Linux, standard approach

- Device discovery
  - Currently: Determine the device path manually
  - Obtain device handle from path:

```
hnd = open(device_path, O_RDWR );
```

- Expectation:
  - Initially, some special system calls **`ioctl()`**
  - Then, system call **`write()`** should suffice to send a report.
- Surprises:
  - System call **`write()`** *not* available
  - Hardly any documentation found!
  - Instead, rather obscure **`ioctl()`** system calls to transfer all report octets into a kernel buffer before sending the report

Writing reports

3) Linux, standard approach: Writing reports
a) KNX USB report data structure

```
struct HidReport{                          // 12 Byte header
   unsigned char  reportId;
   unsigned char  packetInfo;
   unsigned char  dataLength;
   unsigned char  protocolVersion;
   unsigned char  headerLength;
   unsigned char  bodyLengthHi;
   unsigned char  bodyLengthLo;
   unsigned char  protocolId;
   unsigned char  emiId;
   unsigned char  manufactureCodeHi;
   unsigned char  manufactureCodeLo;
   unsigned char  emiMessageCode;
   unsigned char  frame[52];           // max 52 octets remaining
} __attribute__((__packed__));
```

Item(Global): Report ID, data= [0x01]
1

Watch out for big-endian vs. little-endian processors!

See AN037!

Writing reports

3) Linux, standard approach: Writing reports
b) Transmission loop

```
int hidReportSend( HidReport *hidr, int hnd ) {
   signed char *report = (signed char *)hidr;
   struct hiddev_usage_ref   uref;
   struct hiddev_report_info rinfo;

   uref.report_type = HID_REPORT_TYPE_OUTPUT;
   uref.report_id   = *report++;
   uref.field_index = 0; uref.usage_code = 0xFFA10005;

   for( int uindex = 0; uindex < 63; uindex++ ) {
     uref.usage_index = uindex;
     uref.value = *report++;
     ioctl( hnd, HIDIOCSUSAGE, &uref );
   }
   rinfo.report_type = HID_REPORT_TYPE_OUTPUT;
   rinfo.report_id =  0x01;   rinfo.num_fields = 1;
   ioctl( hnd, HIDIOCSREPORT, &rinfo ); return 0;
}
```

Item(Global): Logical Minimum, data= [0x80] 128
Item(Global): Logical Maximum, data= [0x7f] 127

Special treatment of initial report_ID byte!

Item(Global): **Usage**, data= [0x05] 5 (null)

Item(Global): **Usage Page**, data= [0xa1 0xff] 65441 (null)

Item(Global): Report Count, data= [0x3f] 63

Reading reports

How do we receive data
from Konnex devices?

Reading reports

1) Windows (for comparison)
   - Single call („ReadFile“), similar to sending. Convenient

2) Linux, „libhid“ approach
   - Convenient report exchange, similar to the Windows API
   - Two alternatives (control transfers, interrupt transfers)

3) Linux, standard approach
   - Control transfers: Reports received bytewise, similar to sending
   - Interrupt transfers: Reports received through system call **`read()`**
     - Array of data structures which contain the desired report bytes
     - Three different data structures available, selectable through flags
     - Use e.g. separate thread to prevent blocking I/O

   Note: Some work left to be done here.

Results

Results

1) Windows (for comparison)
   - Worked as expected

Results	Call ok	Call error
Effect ok	++	-
No effect	-	-

2) Linux, „libhid“ approach
   - Kernel 2.6.8.14:
     - Non-reproducible results!

New:
     - Kernel reported timeout problems
     - Discussion with libhid authors:
       - Device timing ok?
       - Libhid: No outbound interrupt transfers

Results	Call ok	Call error
Effect ok	+	+
No effect	++	++

The libhid approach is currently NOT an option!

   - Kernel 2.6.12.1:
     - Calls always return ok, but without effect. Kernel: Timeouts!

Results	Call ok	Call error
Effect ok	--	--
No effect	++	--

Results

3) Linux, standard approach
   - Status of conference proceedings, end of July 2005:

Results	Call ok	Call error
Effect ok	-	-
No effect	++	-

→ Kernel debugging required!

   - **Breakthrough** on last Sunday:

Results	Call ok	Call error
Effect ok	++	-
No effect	-	-

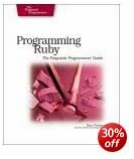
A user space driver is indeed all it takes to make Linux applications talk to KNX devices through a KNX USB interface!

Outlook

Outlook

- Upgrade of Linux/Windows **EIB Homeserver**
  - Will support also USB, both on Linux and Windows
  - Needs adaptation from FT1.2 to „AN037“ protocol
  - Current hardware requires downgrade from EMI2 to EMI1 format
    Note: Authors expected upgrade to cEMI format.

- Support for high-level programming languages like **Ruby**
  - API to KNX USB through extension modules (.dll, .so)
  - Foundation for efficient, high-level programming projects
    - User interfaces
    - Automation tasks



- Open Source **user space driver,** to be supplied on the Web
  - Some work left to be done for the **`read()`** system calls
  - See http://www.informatik.fh-wiesbaden.de/~werntges/proj

Outlook

- A personal vision
  - Low-cost, ubiquitous USB access to KNX systems
    - Maybe integrated into power supplies?
  - An open source „supplement“ to ETS3
    - Near-term goal: Change KNX installation parameters just with a PC and free software – no electrician or high-end panel involved.
    - Far-term goal: An Open Source, platform-independent tool for programming EIB/KNX systems in non-commercial environments
      - Skill requirements: High
      - Price: Low (free tool)
      - Convenience: Low
      - Productivity: Low

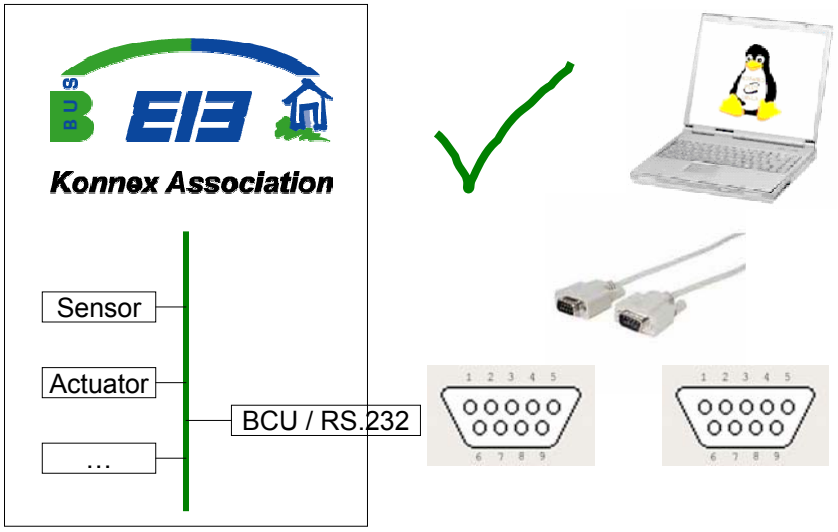
Let‘s offer something to the technical enthusiasts with good skills but low budget. We need those early adopters as „multiplies“!

Appendix

(Optional slides)

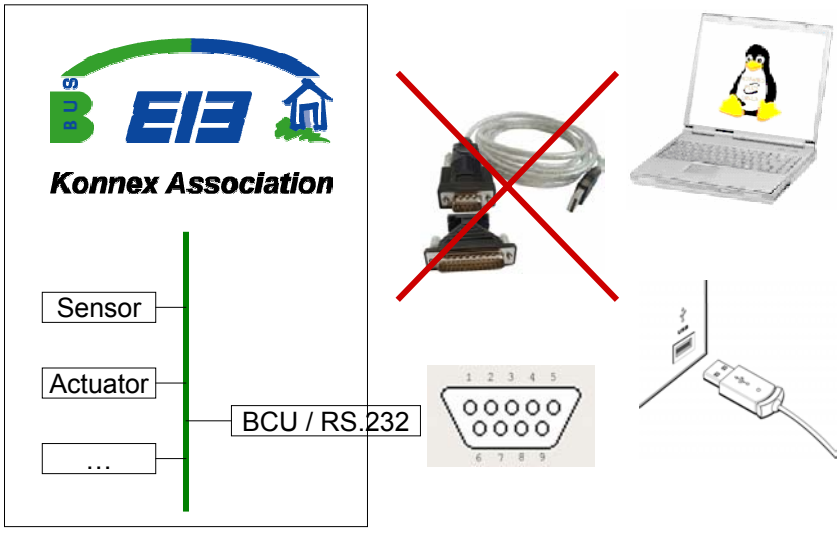
Introduction

- Situation before USB



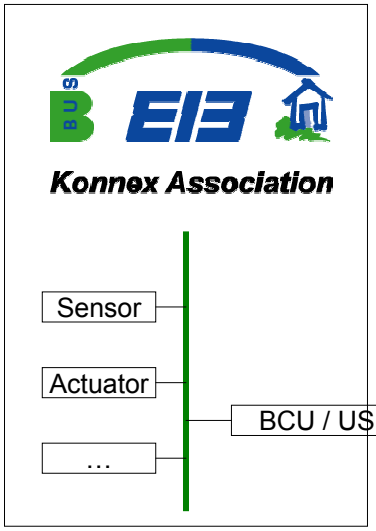
Introduction

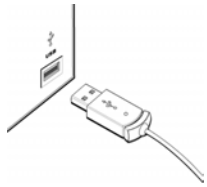
- Situation with USB (only at PC end)

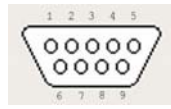


Introduction

- Situation with USB (only at KNX end)

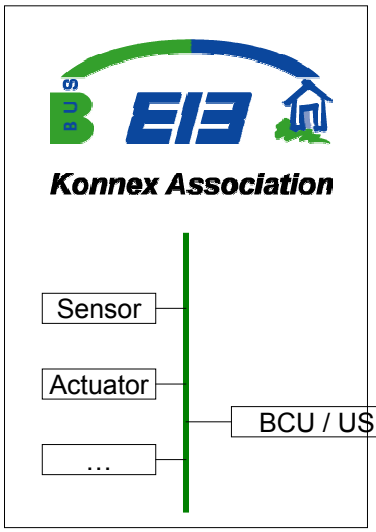


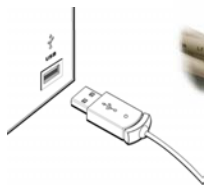




Introduction

- Situation with USB





Background

- Linux: Drivers, modules and devices
  - Kernel 2.6.12.1:
    - 1270 (!) modules in total
    - Most drivers may be either statically linked or dynamically loaded
  - Devices: Special files
    - Accessed as special files in the filesystem („everything is a file“)
    - Two types:
    - **Block devices**
      - Block is smallest access unit, random access
      - Ex: First IDE disk **`/dev/hda`**
    - **Char. devices**
      - Octet is smallest access unit; typically serial access
      - Ex: „COM1“ **`/dev/ttyS0`**
  - Note: One driver may serve multiple devices.

Background

- Linux: Drivers, modules and devices
  - Basic drivers are enumbered by the kernel
    - See „Major device no“ and **`/proc/devices`**
    - Ex: „180 usb“, „188 ttyUSB“
  - One driver/module can control multiple devices
    - See „Minor device no“ and `/proc/devices`
  - File system entries in **`/dev`** map filenames to device no‘s: