Controlling EIB/KNX devices from Linux using USB

Heinz W. Werntges[1]
Fachbereich Informatik, Univ. of Applied Sciences Wiesbaden

Jens Neumann and Vladimir Vinarski
Loewe Opta GmbH, Kompetenzzentrum Hannover

ABSTRACT

Modern PC equipment, most notably notebook PC's, expect peripheral equipment to be connected through the Universal Serial Bus (USB), while "legacy" hardware (like parallel or serial ports) is increasingly abandoned. USB devices are rarely plagued by obscure issues like critical signal timing and are at the same time very convenient for users. On the other side, EIB/KNX systems are traditionally connected to PC's via serial ports. Hence, in 2003, Konnex introduced Application Note 037/02 "KNX on USB protocol specification & KNX USB Interface Device Requirements" [1]. Meanwhile, USB based EIB/KNX bus coupling units are available, and the MS-Windows based ETS3 software [12] supports EIB access through USB without the need for special Windows driver installation.

On Linux platforms, USB support is generally somewhat lagging behind MS Windows-based developments. This is especially true for USB support for EIB/KNX. This study aims to fill that gap by providing an Open Source module that allows Linux programs to control EIB/KNX devices via a USB connection.

Care is taken to avoid any kernel modification, kernel-space driver installation, or low-level USB access. The module works in user space and relies on readily available USB HID devices like `/dev/usb/hiddev0`, thereby simplifying its application.

1. USB FOR LINUX

The Universal Serial Bus (USB) is well-supported by the popular Open Source operating system Linux. Support is accomplished by a collection of layered device drivers and includes both low level access and access specific to certain device classes. - USB serves a great variety of purposes. While this is a good thing to have, it makes software development for USB a bit complicated at first. Let's sort it out a bit:

For Linux, externally added USB components become "devices" when they are connected, i.e. they appear as special entries (nodes) in the file system. Typically, device nodes are stored below the `/dev` directory and are managed by a major and minor device number [2]. While the major device number selects the responsible device driver of the kernel, the minor device number is passed to this driver to allow for specific treatment of the current device by a driver that manages a whole device family.[2]
There are basically two kinds of devices: Block and character devices. USB components can represent any of those. E.g., the popular memory sticks represent block devices and are accessed like hard disks, while keyboards and terminals represent character devices. Traditionally, Linux device nodes are permanently available (you don't remove your hard disk regularly). On the other hand, USB's plug-and-play concept requires the Linux kernel to break and make associations between device nodes and actual devices dynamically, thus causing a need for a device discovery phase.

[1] werntges@informatik.fh-wiesbaden.de / To whom correspondence should be addressed
[2] Linux kernels since V 2.6 introduced an alternative mechanism to manage device nodes: E.g., USB nodes get dynamically added to /proc/bus/usb, where they may be searched.

2. HUMAN INTERFACE DEVICES (HID) and KNX HID

The Human Interface Devices (HID) form a sub-class of the character devices. There are many different kinds of HIDs [3], like keyboards and mice, game controllers and alike. With AN037 [1], Konnex specified USB interface devices to EIB/KNX as belonging to the HID class of devices. Konnex did not try to emulate the established RS.232 interface via USB, as is sometimes seen, and abandoned the FT1.2 protocol in favour of a new KNX HID protocol. Consequently,

Upon connection, HIDs identify themselves to the host system as belonging to a special HID class (like “3-button mouse” or “force feedback joystick”), according to [3], and usages according to usage codes as specified in [4]. Linux’s low-level USB drivers look for such signatures and assign specialized higher-level drivers (like a keyboard or joystick driver) according to the received usage code(s). When no special support is available, the device gets assigned to a generic “hiddev” driver.

Konnex did not try to register or use standardized HID usages for the multitude of EIB/KNX devices. Instead, a KNX HID identifies itself to the system with a “vendor-specific” usage and takes the exchange of HID reports as a means to tunnel EMI messages [5] between Konnex devices and the application.

Consequently, when trying to access EIB/KNX via USB from a Linux application, this application faces following tasks:

1. Discovery of the proper device / device node
2. Writing and reading HID reports using the appropriate driver API
3. Encapsulation of EMI messages by means of the KNX HID protocol

This contribution focuses on the first two tasks while just using the specifications in [1] to accomplish the third.

The Test Environment

All tests were performed with standard equipment:

- PC: IBM Thinkpad Notebook, T42p series
- OS: SuSE Linux 9.2, with Kernel 2.6.8-24 (as shipped)
- KNX USB hardware: Busch-Jäger EIB sensor USB, art. no. 6123 USB-82 and
  BCU art. no. 6120 U-102,
  vendor ID 0x145c, product ID 0x1330

Remark: Surprisingly, this KNX USB device supports neither EMI2 nor cEMI, thereby merging a very recent specification (KNX USB) with a much older one (EMI1), while at the same time with cEMI there is a more up-to-date alternative looking for wide-spread use. A chance missed for cEMI to become more wide-spread in the wake of USB?

Device Discovery

Let’s assume that we connect a KNX USB interface with the USB port of our Linux system through a regular USB cable. The Linux kernel will detect it, determine the responsible device driver, and assign it to an available device node.

An application that wants to communicate with the KNX system now needs to access the appropriate device node. Which one is it?

A simple yet effective approach works like this: The Linux test environment (see below) maintains 16 device nodes for generic HID devices: `/dev/usb/hiddev0` … `hiddev15`. Simply apply an `open()` system call to any of them in order to find out which ones are available. In our case, the KNX HID was the only generic HID connected and could always be found at `/dev/usb/hiddev0`. Monitoring of system messages when the device is connected using utility `dmesg` might also provide helpful hints. In the `/proc/bus/usb` tree of device nodes it may be found through utility `lsusb`.

More systematically, HID devices may be found by looking for their vendor ID and product ID. Utility `/sbin/lsusb` lists all currently connected USB devices including vendor ID and product ID. Call it once before and after connecting the KNX HID and compare the results. The difference should consist of a single line that names the KNX HID and its vendor ID and product ID. Optionally use option `-t` to obtain a tree view of the nodes below `/proc/bus/usb` to identify the device node matching the vendor and product ID there.

This procedure remotely resembles the discovery procedure of HID devices for MS-Windows, which is nicely described in [6].

Once the device node is found, we need to learn about the device's properties, of which there are plenty for a USB device. Use options `-d` *`vendorID`*`:`*`productID`* `-vv` to obtain the full details of your device, like in

```
    $ /sbin/lsusb -d 0x145C:0x1330 -vv
```

Fortunately, AN037 [1] already specifies many important properties. These should be reflected by your device details. In particular,

- The KNX HID should use interrupt transfers for both inbound and outbound reports.
- The KNX HID reports should have a length of 64 octets.
- The report ID should be 0x01.

Other parameters need to be determined from the `lsusb` output, but these depend on the API to be used (see there).

3. APIS

Our application does not need much to exchange data with the KNX system. All it needs to do is opening the device for reading and writing, and reading/writing HID reports of size 64 octets. Report details like EMI messages are left to higher-level functions of the application (see section "EMI messages" below).
The authors tested two alternatives for exchanging such HID reports with the operating system / devices.

System calls

The basic Linux API for communication with devices consists of system calls `open()`, `read()`,`write()`,`select()`,`ioctl()`,`close()` and a few more. Which of those are available depends on the nature of the device and the scope of the applying device driver.

For the task at hand, `open(), close(), write(), read(), and ioctl()` appear to suffice.
A look at the similar situation on MS Windows confirms this expectation: After a (lengthy) discovery phase, users end up with a device path which they can use to open/close the device and read/write from/to it, using system calls `CreateFile(), CloseHandle(), ReadFile()`, and `WriteFile()` [6].

Unfortunately, `write()` turned out to be not available for generic HIDs; instead a very cumbersome mechanism using `ioctl()` needs to be used. To our surprise, documentation on how to send a whole array of octets to a HID using `ioctl()` is virtually non-existent. This lack of documentation turned out to be a major obstacle for this study. Here are the essential lines of code used to send a HID report (the full code will be made available at [9]):

```
struct hiddev_usage_ref uref;
struct hiddev_report_info rinfo;

for( int uindex = 0; uindex < 63; uindex++ )
{
      uref.report_type = HID_REPORT_TYPE_OUTPUT;
      uref.report_id = HID_REPORT_ID_UNKNOWN; // or: 0x01;
      uref.field_index = 0;
      uref.usage_index = uindex;
      if( uindex == 0 ) uref.usage_code = 0xFFA10005;
      else              uref.usage_code = 0xFFA10006;
      uref.value = report[uindex];       // report[] = report buffer
      ioctl( fd, HIDIOCSUSAGE, &uref ); // Error treatment omitted
}
rinfo.report_type = HID_REPORT_TYPE_OUTPUT;
rinfo.report_id = 0x01;
rinfo.num_fields = 1;
ioctl( fd, HIDIOCSREPORT, &rinfo ) ; // Error treatment omitted
```

This code essentially copies 64 octets from user buffer `report[]` to a kernel buffer by filling and passing data structures `hiddev_usage_ref`. From the output details of `lsusb` we learned the required parameters: There is only one report descriptor ("bNumDescriptors 1"), hence `rinfo.num_fields=1;` and `uref.field_index=0;` . The (innermost) usage page is 0xFFA1, and usage data are 0x05, 0x06 for output and 0x03, 0x04 for input, so `uref.usage_code = 0xFFA10005;` and `uref.usage_code = 0xFFA10005;` for output. Constant `HID_REPORT_ID_UNKNOWN` may be replaced by 0x01, as we know from `lsusb` (as well as from [1]) that this is the proper report ID. In this case, only 63 octets may be transmitted, and we drop the first octet of the report buffer (which is the report ID!).

There are two mechanisms for reading inbound reports: A simple call of `read()`, or the reversal of the writing procedure outlined above. i.e. an `ioctl()` call with code `HIDIOCGREPORT`, followed by 64 `ioctl()` calls with code `HIDIOCGUSAGE`, with `uref.usage_index` ranging from 0 to 63.

"libhid"

The Open Source project "libhid" [7] (successor of "hidlib") provides an alternative access path to HIDs. This API directly supports device discovery mechanisms based on vendor ID and product ID as well as convenient routines for writing as well as reading whole reports. Its authors also announced plans to port the library to Windows, thus providing a single and convenient multi-platform API for HID access.

In order to do this, the library somewhat interferes with Linux's device driver stack by "stealing" a HID device from its regular driver, accessing the device through a lower-level USB driver, and by restoring things afterwards.
We tested the library by elaborating on a source file "test_libhid.c" that is provided by the software package. For device discovery, we only needed to specify the vendor and product IDs: `HIDInterfaceMatcher matcher = { 0x145c, 0x1330, NULL, NULL, 0 };`
Following an included description, we set the required `PATH_IN` and `PATH_OUT` arrays:

```
  unsigned char const PATHLEN = 3;
  int const PATH_IN[]  = { 0xffa00001, 0xffa00001, 0xffa10003 };
  int const PATH_OUT[] = { 0xffa00001, 0xffa00001, 0xffa10005 };
```

Those values again are derived from the `lsusb` output.
Sending a report is then done by a simple call:

```
  hid_set_output_report(hid, PATH_OUT, PATHLEN, reportBuffer, 64);
```

Receiving a report works similarly:

```
  hid_get_input_report(hid, PATH_IN, PATHLEN, reportInBuffer, 64);
```

4. EMI MESSAGES

At the interface level, EMI messages only need to be turned into HID reports according to the KNX USB protocol in [1], and vice versa. In addition, [1] specifies a few special data frames for internal purposes. As mentioned earlier, this paper focuses on the Linux APIs to HID devices, not on protocol implementation. Nonetheless, two examples should illustrate what needs to be done here (all octets in hex and padded with "0" octets up to 64 in total):

1. *Query the KNX USB device for the supported EMI types*
   This is accomplished by a "device feature get frame", which in our case consists of the following octets
   ```
     01,13,09, 00, 08, 00,01, 0F, 01, 00,00, 01
   ```

2. *Activate Link Layer (using EMI1)*
   We'll learn that our KNX USB device only supports EMI1, and it does not support bus monitor mode, so we first set the active EMI type to "1" (not shown here) and then use an EMI1 PC_SET_VAL.req to activate link layer access by sending:
   ```
     01,13,0D, 00, 08, 00,05, 01, 01, 00,00, 46,01,00,60,12
   ```

Note that the first octet (01) is easily recognized as the report ID.

5. RESULTS

The involved test reports consisted of

- a query of the KNX USB device for the supported EMI type, including reading of the corresponding input report (case 4.1),
- a report to set the active EMI type to EMI1,
- a report containing a EMI1 PC_SET_VAL.req message to activate link layer access (case 4.2),

- and a report containing a EMI1 L_data.req message that switched a light on (group address 1.0.0)

All reports had been successfully tested under MS Windows XP Professional by employing the Windows DDK [11] and "C" code following the descriptions in [6].

Tests with System calls

All `ioctl()` write calls returned successfully. The involved parameters had to be just the ones listed here, or else the calls failed. Using `0xFFA10005` or `0xFFA10006` as the usage code worked both; it remained unclear when to use which code. Setting `uref.report_id=0x01;` required to drop the first octet from the report, as more than 63 octet would be refused then. Apparently, the kernel drivers add the report ID automatically when specified. The `ioctl()` calls however did not have any noticeable effect, even though they all returned with success codes.

Attempts to call `read()` for the expected result report of test case 4.1 ended with the call blocking, which is to be expected when there is no inbound report in the driver's input buffers. Using `ioctl()` calls for the same purpose (as outlined earlier) always resulted in "receiving" a sequence of "0" octets.

While there is hope to eventually reach a breakthrough, using system calls on the basis of the generic "hiddev" driver is not an option at the moment. The lack of documentation on how to access HIDs with Linux through "hiddev" is striking and prompts for a detailed analysis of the involved kernel sources.

Tests with "libhid"

The good news is that reading and writing HID reports using "libhid" worked. The KNX USB module answered, and the light could be switched on and off. However, calls frequently returned error codes. In some of these cases, they still had the desired effect, mostly though they just failed. Success and failure turned out not to be reproducible: When calling the test program many times, the calls succeeded or failed in seemingly random order.

The library also failed to disconnect cleanly from the USB driver stack: An `open()` call to device `/dev/usb/hiddev0` failed after testing with "libhid" and resumed normal operation only after unplugging & re-plugging.

"libhid" still comes with a low release number (0.2.15), indicating that issues like the observed ones might still be expected. Its ease of use is tempting, and – it works. However, the observed issues need to be resolved before any serious EIB/KNX application could rely on it.

6. OUTLOOK

Both the system call and the "libhid" approach will be further investigated by the authors, until reliable operation is reached. Results and test material will be made available as Open Source software at [9]. As usual for Open Source projects, contributions are welcome.

Once reliable operation is achieved, the module will be incorporated into the Open Source project "EIB Home Server" [8] as an alternative driver to the existing FT1.2 over RS.232 mechanism. The home server essentially consists of two components:

- **homedriver**: Controlling of the EIB/KNX bus and communication interface between the home server and the EIB/KNX bus, i.e. the electronic devices.
- **homeserver**: Communication interface between the homedriver and application programs called clients. The number of application programs is unlimited.

The existing home server implementation will be completed to be able to send EMI1 messages via USB to the EIB/KNX bus or back to the home server. Since the home server also supports Windows, where USB access to EIB/KNX is already working, USB support on Windows is an obvious next step.

While the home server project addresses the controlling aspects of EIB/KNX systems, there is still room for software that can change parameters of EIB/KNX devices, or can even program new EIB/KNX systems, at least in simple cases. For commercial purposes on Windows platforms, there are well-established tools like EIBA's ETS3 software [12]. Open Source components for Linux especially appeal to early adopters of new technologies with good technical skills but limited budgets, or to system integrators aiming for cost-efficient "intelligent" components that can be produced free of licence fees.

Over the past few years, new high-level programming languages like Ruby [10] became available. They are highly dynamic, strictly object-oriented, and come with powerful class libraries, thus making software development easier and more efficient than ever. One key feature is their ability to easily "bind" to low-level libraries written in "C/C++" – like the one being developed here for KNX/USB access. Such binding will provide convenient, object-oriented and highly efficient programming access to EIB/KNX as a basis for multiple application-oriented projects yet to come.

7. REFERENCES

[1] Konnex: Application Note 037 "KNX on USB", KNX Handbook V 1.1 Rev. 1, 2003.
[2] A. Rubini, J. Corbet: Linux Device Drivers – 2nd ed, O'Reilly & Associates, Inc., 2001.
[3] Device Class Definition for Human Interface Devices, V 1.11, http://www.usb.org/developers/devclass_docs/HID1_11.pdf, 2001-06-27.
[4] HID Usage Tables, V 1.11, http://www.usb.org/developers/devclass_docs/HID1_11.pdf, 2001-06-27.
[5] Konnex: "External Message Interfaces", KNX Handbook V 1.1 Rev. 1, Vol. 3, Part 6, Chapter 3, 2003.
[6] J. Axelson: USB Complete, 2nd ed, ch. 16, Lakeview Research, 2001.
[7] Martin F. Krafft, Arnaud Quette, Charles Lepple : Open Source project « libhid » v 0.2.15, http://libhid.alioth.debian.org/ , 2005-04-26.
[8] Part of BMBF sponsored project „Embassi" (see http://www.embassi.de and http://sourceforge.net/projects/eibcontrol/)
[9] Details will be made available at http://www.informatik.fh-wiesbaden.de/~werntges/proj/
[10] http://www.ruby-lang.org
[11] Windows Driver Development Kit, see http://www.microsoft.com/whdc/devtools/ddk/default.mspx
[12] ETS3 Professional, see http://www.eiba.com/en/software/ets3/index.html#Prof

Acknowledgment

The authors like to thank Dr. Th. Weinzierl and Mr. J. Demarest (Konnex) for technical support on KNX USB testing details.