A Linux device driver for EIB-TP-UART-IC and kernel 2.6

(Version 0.03a0)

README

Reinhold Buchinger

Matrikelnr: 0125124

Oct. 2004

1 Changes from V0.02 to V0.03

With V0.03 the driver was ported to kernel 2.6. You must use V0.02 for kernel versions 2.2 and 2.4. Read the document Changelog to get information about added functionality in the new version. A description of the implementation can be found in the document Source Documentation.

The author of V 0.02 is Raffael Stocker(raffael.stocker@stud.fh-deggendorf.de). The author of the ported version 0.03 is Reinhold Buchinger (e0125124@student.tuwien.ac.at).

This file is a modified version of the readme of V0.02.

2 What is tpuart?

Tpuart is a Linux kernel (version 2.6) module for interfacing a PC to the European Installation Bus (EIB). It uses the Siemens-TPUART-interface for the serial port. All you need to use it is a PC with Linux and a standard serial port (must have a FIFO), the TPUART-interface and your EIB system, of course.

The EIB is a home automation system defined by the EIBA (EIB Association, <www.eiba.org>). The protocol specification should be downloadable from their website.

Information on the TPUART-interface and -chip can be downloaded from <http://www.georg-luber.de/chips.htm>.

3 What exactly does it do then?

Tpuart is a device driver. Neither more nor less. If you want a GUI-driven application to program and manage your EIB-system, then you are at the wrong place here. The driver's only purpose is to give you a low-level interface to the hardware. If you want to really use it for something, you will have to write your own application on top of the driver's interface.

4 Installing the driver

To install the driver first extract the archive's contents with

```
tar xvzf tpuart-<version-code>.tar.gz
```

This will create a subdirectory called tpuart-<version-code>. Since this is only a source archive, you will have to compile it. Enter

```
make
```

in the subdirectory of the driver. After successful compilation, the driver can be installed in the right place.

```
make install
```

as root will copy the driver module to /lib/modules/<current-kernel-version>/char. You can change the path if you modify the variable INSTALL_PATH in the Makefile. If this doesn't work on your distribution, you will have to install it by hand.
You can load the driver with the init script tpuart.init. The name of the variable DEVICE must be the same as MODULE_NAME in the Makefile (the default name is tpuart). If you want to specify options you can create a config file. The name of the config file has to be the name of the variable DEVICE with .conf ending and should be stored in /etc. You can change the path with the variable CONFIG_PATH.
All parts of the config file are optional. The format of the config file is:

```
owner <ownername>
group <groupname>
mode  <modename>
options tpuart_major=<majornumber> heartbeat_interval=<heartbeat_interval in sec>
```

With owner,group,mode you can specify the attributes of the entries in /dev. The default value for the major number is 0 which means dynamic configuration. The default value for heartbeat_interval (see section 5.4) is 30 seconds.
Type

./tpuart.init start

to load the driver and

./tpuart.init stop

to unload the driver.
If you want to load the driver automatically you can copy the init script in the directory your distribution uses for these scripts[1] and make a link to it from the appropriate run-level directory.
You should now be able to use the driver.

5 Using the driver

5.1 Read/write

For interfacing to the driver, four special files are created:

/dev/tpuart0 (using io-port 0x3f8)
/dev/tpuart1 (using io-port 0x2f8)
/dev/tpuart2 (using io-port 0x3e8)
/dev/tpuart3 (using io-port 0x2e8)

First you have to connect to the driver by calling `open(2)` in your application. The following code sample opens the driver for the device /dev/tpuart0 and allows read and write access.

```
int fd;
char *device = "/dev/tpuart0";

fd = open(device, O_RDWR);
if (fd == -1){
        perror("open");
}
```

At last you should disconnect from the driver via `close(2)`.

```
close(fd);
```

After connecting to the driver you can `read(2)` and `write(2)` messages using the following structure:

```
struct telegram_t {
        struct timeval timestamp;     /* time of message reception,
                                         leave empty when sending. */
        unsigned short data_length;   /* overall message length. Must
                                         always be set appropriately when
                                         sending! */
        unsigned char data[64];       /* message data */
};
```

Data can only be sent and received using this structure! Since it is not globally defined in the driver's header file, you have to define it by yourself (this is because struct timeval is different in the kernel, compared to the C-library).
The `data` array contains an EIB frame **without the checksum** as last octect. The checksum is calculated by the driver itself. The following code sample writes a test message to the bus. The `timestamp` value is only interesting if you read a message.

```
struct message {
        struct timeval timestamp;
        unsigned short length;
        unsigned char data [64];
};

unsigned char testMessage[] = {0xbc, 0x11, 0x7, 0x0, 0x1, 0xe1,
                               0x0, 0x80 };

struct message telegram;

/* the length of our messages */
telegram1.length = 8;
```

1 Most common directories uses are /etc/init.d, /etc/rc.d/init.d.

```
/* initialize messages */
memset (telegram.data, 0, 64);
memcpy (telegram.data, testMessage, telegram.length);

/* write the message */
if (write (fd, &telegram, sizeof (telegram)) <= 0) {
        perror ("write");
        }
```

Reading a message is just as easy. The following code sample reads a message and prints all available information to stdout.

```
struct message {
        struct timeval timestamp;  /* will hold the receive time */
        unsigned short length;     /* holds the message length in data */
        unsigned char data [64];   /* holds the message itself */
};

struct message telegram;
int i;

/* read a message */
if (read (fd, &telegram, sizeof (telegram)) <= 0) {
        perror ("read");
        return 1;
}

/* print received information */
fprintf (stdout, "Seconds: %10lu\t", (unsigned long)telegram.timestamp.tv_sec);
fprintf (stdout, "Microseconds: %10lu\n",
                                (unsigned long) telegram.timestamp.tv_usec);
fprintf (stdout, "Wall time: %s", ctime (&telegram.timestamp.tv_sec));
fprintf (stdout, "Data length: %u\n", telegram.length);

fprintf (stdout, "Data: ");
for (i = 0; i < telegram.length; i++) {
        fprintf (stdout, "0x%x ", telegram.data[i]);
}
```

5.2 I/O commands

Before you read and write from/to the EIB bus you should set the physical and group address(es) of your device. You can do this using the `ioctl(2)` call. Therefore you have to include `tpuart_main.h` where the different commands are defined. The code sample shows how to set one physical address to 0x1109 and two group addresses (0x1 and 0x2).

```
unsigned short addr = 0x1109;  /* our physical address */
unsigned short gaddr1 = 0x1;   /* our group addresses */
unsigned short gaddr2 = 0x2;

/* set our physical address */
ioctl (fd,TPUART_SET_PH_ADDR,&addr);

/* set our group address*/
ioctl (fd,TPUART_SET_GR_ADDR,&gaddr1);
ioctl (fd,TPUART_SET_GR_ADDR,&gaddr2);
```

You can set multiple physical and group addresses for one device and even up to 16 addresses at once using different ioctl commands. All the different I/O commands are summarized in Table 5.1.

ioctl-command	***description***
TPUART_SET_PH_ADDR	set a physical address of host
TPUART_HAVE_PH_ADDR	check if a physical address is set (return 0 or 1)
TPUART_UNSET_PH_ADDR	clear a physical address
TPUART_SET_PH_ADDR_ARRAY	set up to 16 physical addresses at once (excluding 0x0000)
TPUART_UNSET_PH_ADDR_ARRAY	clear up to 16 physical addresses
TPUART_SET_GR_ADDR	set a group address
TPUART_UNSET_GR_ADDR	clear a group address

ioctl-command	*description*
TPUART_HAVE_GR_ADDR	check, if a group address is set
TPUART_SET_GR_ADDR_ARRAY	set 16 group addresses
TPUART_UNSET_GR_ADDR_ARRAY	clear 16 group addresses
TPUART_RESET	send reset request to TPUART
TPUART_BUSMON_ON	turn busmonitor mode on
TPUART_BUSMON_OFF	turn busmonitor mode off (equals TPUART_RESET)
TPUART_SET_POLLING_SLOT	set polling slot
TPUART_GET_POLLING_SLOT	get polling slot
TPUART_SET_POLLING_ADDR	set polling address
TPUART_GET_POLLING_ADDR	get polling address
TPUART_SET_POLLING_RESP	set polling response
TPUART_GET_POLLING_RESP	get polling response
TPUART_GET_STATE	get tpuart_state (send a tpuart state request)
TPUART_GET_WRITE_STATUS	get state of last written telegram

Table 5.1: ioctl-commands

Some additional remarks on TPUART_GET_STATE and TPUART_GET_WRITE_STATUS are necessary. TPUART_GET_STATE sends a state request message to the TPUART. This call will always block for a short time until the response message has arrived. It returns the received TPUART-Control Field which has the following format:

TP-UART-Control Field							
7	6	5	4	3	2	1	0
SC	RE	TE	PE	TW	1	1	1

Fig. 5.1: TP-UART-Control Field

During error-free operation state 7 (the 3 least significant bits set) should be returned. The bits 3 to 7 have the following meanings:
SC = Slave Collision
RE = Receive Error (checksum, parity oder bit error)
TE = Transmitter Error (e.g. illegal control byte)
PE = Protocol Error (e.g. illegal control byte)
TW = Temperature Warning

TPUART_GET_WRITE_STATUS returns the status of the last successfully queued telegram. It allows you to receive error values if you use nonblocking write calls. The possible return values are 5 different constants which are defined in `tpuart_main.h`. If the telegram was successfully transmitted CONF is returned, SENDERR in the case of a transmit or receive error. If the heartbeat (see Section 5.4) died DIED_BEFORE_CONF is returned and if no information is available at this moment STATE_AWAITING is returned.

5.3 Blocking/non-blocking commands

By default the read and write calls are blocking. If a process calls read but no data is (yet) available, the process sleeps until some data arrives. If a process calls write and there is no space in the buffer the process blocks. Additionally the process sleeps until the transmission to the TPUART is over and a response is received.
The behavior of read and write is different if the `O_NONBLOCK` flag (an alternate name is `O_NDELAY`) is specified. This flag can be set with `open(2)` or with `fcntl(2)` which manipulates a file descriptor. Then the write and read operations return immediately. You get more information about the different return values in Section 5.5.

5.4 Heartbeat

The driver V0.02 could block infinite in read/write without noticing a dead TPUART. To avoid this, this version implements a heartbeat. Every `heartbeat_interval` seconds a state message is sent to the TPUART. `Heartbeat_interval` is a module parameter and can be specified in the config file (see Section 2). If within

this interval no state message is received the TPUART is considered to be dead. Thereupon a sleeping `read/write` or `ioctl` state request will return with the error value EREMOTEIO (remote I/O error). The heartbeat now tries to reset the TPUART in `heartbeat_interval` intervals. No further read/write, state request or turning the busmonitor on is possible until a successful reset. The interval between two resets could be abbreviated with manual ioctl resets. If we have recognized a successful reset the heartbeat continues with sending state messages and new tasks for the driver are possible.
Attention: The heartbeat is turned off in busmonitor mode, because the TPUART cannot handle state messages in this mode. Therefore the driver could block infinitely.

5.5 Return values of read/write

In the best case the read command returns the size of the received telegram. If an error occurred while copying the telegram to user space errno is set to EFAULT. If the count parameter of the write call is smaller than the size of the `telegram_t` structure EINVAL is returned. In the nonblocking case EAGAIN is returned if no telegram was received. The driver returns with EREMOTEIO set if the heartbeat couldn't detect the TPUART. This value is returned as long as no successful reset is achieved.

errno	***reason***
EFAULT	couldn't copy telegram to user space
EINVAL	size of telegram ≠ declared value
EAGAIN	no telegram available (non-blocking)
EREMOTEIO	heartbeat failed

Table 5.2: error values of read

The write command returns the size of the transmitted telegram if no error has occurred. The process can sleep twice, first if the buffer is full and second to wait for an answer from the TPUART. In the non-blocking case the driver returns immediately with EAGAIN if the buffer is full. EPERM is returned if the busmonitor mode is turned on because no writes are allowed in this mode. If the write call waits for an answer from the TPUART three error codes are possible. EIO is returned if a transmit or receive error was detected. ECOMM is returned if a negative L_Data.confirm was received. If the heartbeat fails EREMOTEIO is returned. Again this value is returned until the heartbeat achieved a successful reset.
In the non-blocking case it was not possible to receive these error values. Therefore the I/O command TPUART_GET_WRITE_STATUS was introduced (see Section 5.2).

errno	***reason***
EAGAIN	buffer full (non-blocking)
EPERM	busmonitor mode
EIO	transmit or receive error
ECOMM	negative Data confirm
EREMOTEIO	heartbeat failed

Table 5.3: error values of write

5.6 Asynchronous notification and polling

It is possible that a SIGIO signal is sent whenever new data arrives. Your program has to execute two steps to enable asynchronous notification. First you store the process ID of the owner process in the file structure using the `fcntl(2)` system call. This step is necessary for the kernel to know just who to notify. The signal will be sent to this process (or process group, if the value is negative). In order to actually enable asynchronous notification, the user program must set the FASYNC flag.

```
sigaction(SIGIO, &action, NULL);

fcntl(STDIN_FILENO, F_SETOWN, getpid());
oldflags= fcntl(STDIN_FILENO, F_GETFL);
fcntl(STDIN_FILENO, F_SETFL, oldflags | FASYNC);
```

Furthermore the `poll(2)`[2] system call is implemented. It allows a process to determine wheter it can read or write to one or more open files without blocking. If the tpuart is readable the POLLIN and POLLRDNORM flags will be set. If the device is writeable the POLLOUT and POLLWRNORM flags will be set. I refer to the man pages of `poll(2)` for the exact usage of this system call.

2 You can also make use of the `select(2)` system call which essentially has the same functionality.

6 Known bugs

The only reliable possibility to detect the end of an EIB frame is the detection of a timeout after the last byte. Naturally this timeout is 5.42 ms long but the tpuart subtracts 3 ms since it has the ability to send diagnostic and error information to the host. This adds up to a 2 ms timeout (approximately) to be met by the PC. This is something one cannot normally achieve with a non-real-time operating system.

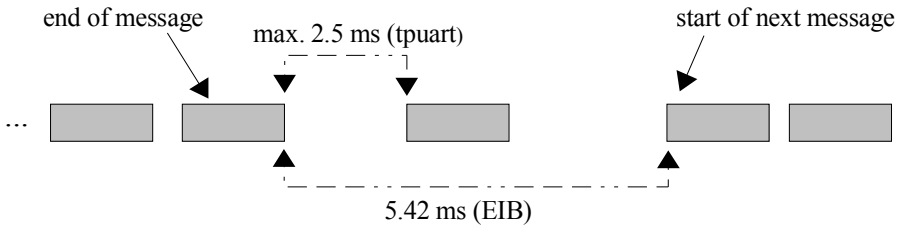

Fig. 6.1: timing constraints

That this is the only possibility to detect the end of a message is no problem for a microcontroller since they can easily keep up with timing requirements. For a normal PC this leads to serious problems. If we miss a timeout wrong bytes would be added to our message (probably this will be detected by the checksum procedure but we lose the information of these bytes). If we detect the end of a frame too early the next received byte(s) will be interpreted incorrectly. The problem is that the start and the end of a frame is not especially escaped. The best solution to this problem would be to add begin and end octets to the EIB messages (or, at least, to add them between the tpuart to PC communication).
In Version 3.0 detecting the end of a frame via timeout is only an experimental adjustment for the driver. It timestamps every received byte in the interrupt service routine and compares this timestamp with the timestamp of the last received byte. But this approach cannot be reliable in a non-real-time OS. The interrupt service routine could be delayed which leads to a wrong timeout.
The driver bypasses the problem of detecting a timeout by analyzing the length information of a frame. It presumes the end of a frame when the necessary number of bytes were received. This could led to a bug if the length information of the frame is wrong. If the length information is different from the length of the received telegram we abort the receiving mechanism too early (late). This leads to a misinterpretation of the next received byte(s) and our receiving machinery gets out of sync.
You can find more information about the detection of a frame end in Source Documentation , Section 1.2.

The driver treats repeated telegrams the same as non repeated ones, i. e. it doesn't care about the repeat-flag in the control byte. This also means if someone else except us sends a busy acknowledge we will receive the telegram twice. If we ourselves send a busy we only will treat the repeated telegram.

7 Thanks et al.

7.1 V0.02

This work was done as a project at our University together with Siemens A&D, Regensburg. From there, we wish to thank Georg Luber, Marc Wucherer, Andre Hänel and Josef Hofmeister. They supported us with information and hardware during the development.

The Author can be reached at <raffael.stocker@stud.fh-deggendorf.de>.

There is a Webpage about the driver and other EIB-related stuff at http://os-projects.fh-deggendorf.de/eib/

Raffael Stocker (06-Jun-2001)

7.2 V0.03

The Author of the port can be reached at <e0125124@student.tuwien.ac.at>

I wish to thank Wolfgang Kastner and Georg Neugschwandtner for helping me a lot during this project.

Reinhold Buchinger (08-Oct-2004)