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

(Version 0.03a0)

Source Documentation

Reinhold Buchinger

Matrikelnr: 0125124

Oct. 2004

1 Overview over the driver

The driver implements the standard Unix interface for device drivers. It allows of reading/writing telegrams from/to the EIB bus via standard `read(2)`/`write(2)` commands. Additionally a busmonitor mode is supported where every telegram on the bus is passed to a reading process (not only for our device addressed telegrams). The implementation of the `poll(2)` command allows the process to determine whether it can read from or write to the bus without blocking. By means of the `fasync` function a asynchronous notification is possible. In this case a registered process will receive a SIGIO signal when data arrives. Via `ioctl(2)` commands you can set/unset multiple physical and/or group addresses and query if a special address is assigned to the device. Furthermore you can enter the busmonitor mode and leave it which is equivalent to reset the TPUART. You can query the state of the TPUART and the state of the last write request (important for non-blocking writes). The README document describes in detail how to use the driver.
The driver always allows only one process to open the device. There can never be two processes communicating with the same TPUART at the same time.

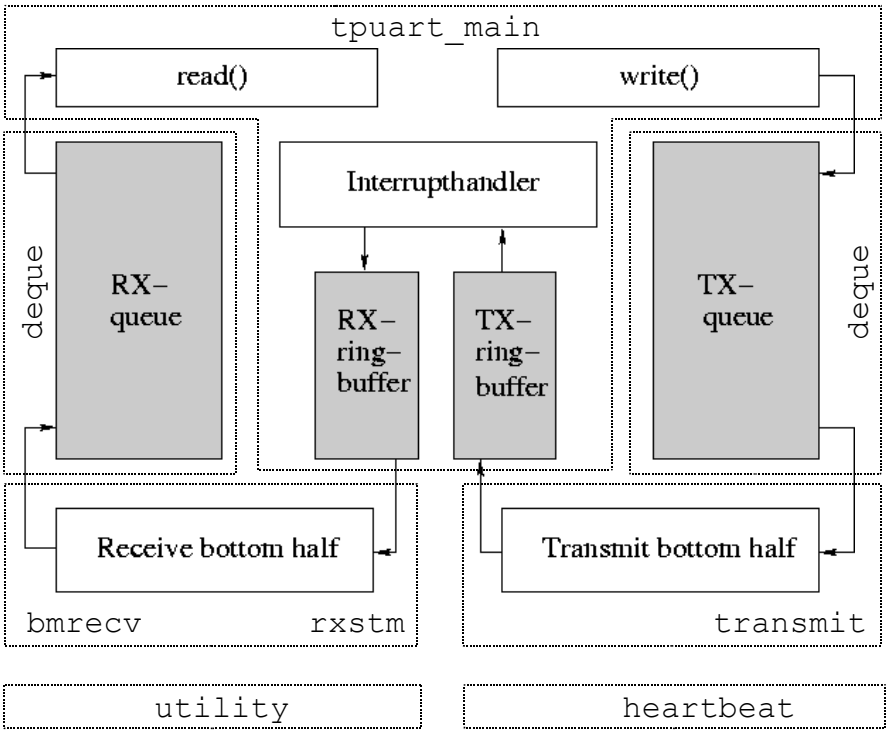

Fig. 1.1: A schematic view of the driver

1.1 How everything works together

Fig. 1.1 shows in a schematic view how a read/write is executed. Additionally the squares with dashed lines indicate the different modules and their responsibility.
The driver is made up of 7 modules:

- tpuart_main: implements the Unix interface for device drivers and the interrupt service routine.
- deque: implements the RX- and TX-queue which buffer the telegrams.
- transmit: implements the transmit bottom half. It prepares the telegrams for transmission to the TPUART.
- rxstm: implements the receive bottom half. It assembles the received telegram.
- bmrecv: implements the receive bottom half in busmonitor mode.
- heartbeat: checks the TPUART in periodic intervals.
- utility: useful utility functions (e.g. addressing).

Every module is described more precisely in Section 2.
The tpuart driver is a typical example for an interrupt driven I/0. Therefore buffers are necessary. The TX-queue buffers whole telegrams received by the `write` function from a user process until they are processed by the transmit bottom half. In almost the same manner the RX-queue buffers telegrams until a user process reads them. The functionality of these buffers is implemented in the module deque.
The TX-ringbuffer is necessary to buffer the prepared telegrams, processed by the transmit bottom half, until they are transmitted to the TPUART by the interrupthandler. The RX-ringbuffer stores the (by the ISR) received parts of a telegram which is reassembled by the receive bottom half.
The pointer `u8 *buffer` points to the beginning of a ringbuffer and the buffer itself is controlled by 4 more pointers (`u8 *read, u8 *write, u8 *first, u8* end`). These pointers are part of the `xmit_t` structure which is defined in tpuart_main.h. This header file also defines the type `tpuart_t`. The `tpuart_t` structure

contains all the information which is specific for every (opened) TPUART device and is stored in the private data of the file pointer[1] which is an argument to the driver functions. In the `tpuart_t` structure you can find 2 variables of type `xmit_t`, `rx` and `tx`, one for receiving and one for transmission matters. Beside the ringbuffer and some other information the `xmit_t` structure also contains the variable `deque_t tel_buffer` – the RX- respectively TX-queue as described above.
To make things more clear we now describe the process sequence of a read and a write.
Let's first take a look at the write process. The starting point is `write` in tpuart_main. Writing to TPUART is only possible if the device is not in the busmonitor mode. If there is space left in the TX-queue (`tel_buffer`) the data is copied to the buffer and the transmit bottom half[2] is queued on the transmit work queue. This means that the bottom half is executed when is it scheduled to run (some undefined time in the future)[3]. If the queue is full the process sleeps until some space is available (blocking case). Next the process sleeps until the data is transmitted to the TPUART. In the non-blocking case we return immediately if no space is available in the TX-queue or after we have put the telegram into the TX-queue.
The bottom half for writing is the function `transmit` in transmit.c. The message stored in the TX-queue is taken and extended with the control bytes needed by the TPUART. This extended message which is now ready to send is stored in the TX-ringbuffer (part of the `xmit_t` structure). The first control byte and the first data byte are transmitted to the serial port to provoke a transmitter interrupt. The remaining bytes are transmitted via the interrupt handler (part of tpuart_main.c). When the last byte was transmitted and we received a response from the TPUART (conf, no_conf, transmitter or receive error) we can wake up the process sleeping in `write`.
The read process is similar, it only works the other way around. The starting point is now the interrupt service routine in tpuart_main.c. The ISR is responsible for recognizing the start of a new telegram. It puts the received bytes and necessary flags (e.g. FOR_US, RCV_BUSY, ...) for reassembling the message in the RX-ringbuffer. Therefore every received byte allocates two bytes in the RX-ringbuffer. Besides the interrupt handler sends the acknowledge if our device was addressed and detects timeouts.[4] In the same manner as `write` the ISR has to queue a bottom half which is responsible for the further processing of the data. It depends on the actual mode which bottom half the interrupt handler queues on the receiving work queue. If the driver is in busmonitor mode `bm_recv` of bmrecv.c is called. This is the simpler bottom half. All received data is propagated to the RX-queue and a sleeping reading process waiting for data is woken up.
If the driver is in normal mode `rxstm` of rxstm.c is queued by the interrupt handler. In rxstm.c a state machine is realized to reassemble the message. The state machine is responsible for verifying the checksum and if the received message wasn't send by ourselves (all transmitted telegrams are written back by the TPUART). After reassembling the message and no error has occurred the message is stored in the RX-queue and any sleeping reading process is woken up.
When the `read()` command is executed it simply returns a telegram from the RX-queue. If no data is present the process sleeps (or returns immediately in the non-blocking case).

1.2 Detection of a frame end

As explained in README, Section 6, the only reliable method to detect the end of a frame is the detection of a timeout. But this cannot be done easily in a non-real-time OS.
The driver implements two possibilities to detect the end of a frame. Either it uses the length information of the received frame or it tries to detect the timeout. Which possibility is chosen depends on a define in tpuart_main.h. If TIMEOUT_DETECTION is defined the solution with timeouts is taken else the length information is evaluated.
The evaluation of the length information should be the standard selection. The ISR treats received bytes as parts of the current telegram as long as the assumed length isn't reached[5]. This approach works fine as long as the length information is (received) correct. If the length information is less than the real length we complete the frame too early and the next received byte is interpreted in a wrong way. The control and status bytes respectively the start and end of a frame cannot be distinguished from normal frame data bytes. By chance the next received data byte may look like a start byte or whatever. Therefore our driver can get completely out of sync and we are not able to receive a telegram anymore. On the other hand if the supposed length information is too large we interpret every received byte as frame byte as long as the supposed length is reached. The checksum will prevent that this wrong frame is delivered. However the information of the received bytes is lost, we complete the frame in a wrong moment and get out of sync again. There also exists a timeout in this mode which maybe can help us to resynchronize the driver. If we are trying to assemble a frame and we receive the current byte more than TEL_TIMEOUT usec after the last byte we try to reset the TPUART, reset the receiving state machine to its initial state[6] and try to find the next start byte of a frame. TEL_TIMEOUT is

1 Note that a FILE defined in the C libary has nothing to do with the struct file in kernel code. You can get more information in [RC01], pp. 66
2 The variable `worker` in the `xmit_t` structure points to the worker function.
3 You can find more information on workqueues in [LWN]
4 Detecting timeout has a bug. Read README Section 6 for details.
5 At least we receive 8 bytes because we have to receive the length information.
6 This only works if the bottom half of the last received byte is already processed. Hopefully this is true which a

defined in tpuart_main.h.
The second option is to use the timeout to detect the end of a frame. This should be seen as experimental modus and cannot be used reasonable in the way it is implemented in this version. This end detection is chosen if TIMEOUT_DETECTION is defined in tpuart_main.h. If the current byte is received TEL_TIMEOUT usec after the last byte we assume that we have detected a correct timeout and complete the frame. As described in README, Section 6, the ISR can be delayed and we have no guarantee that we didn't detect a timeout too early. Beside the existing timing problems we only measure the time between the arrival of 2 bytes. The outcome of this is that we have to receive an additional byte (which can take a while) after the last byte of a frame to complete the frame[7]. Therfor we have to call the bottom half `rxstm` once again but without having received new data. To minimise the changes of source code two dummy bytes (all bits zero) are put into the RX-queue and the TIMEOUT flag is set before the bottom half is queued. This provokes the bottom half to complete the frame.

2 Modules

This section provides a short survey of the modules. For a detailed description you should read the source code.

2.1 tpuart_main

This module implements the Unix interface for device drivers and the interrupt service routine which is responsible for receiving and transmitting bytes from/to the serial port. It follows a list of all functions with a short description:

- *static int __init tpuart_init (void)*
  This is executed when the driver is loaded. The major number is assigned to the driver and a message is printed to the kernel log file.

- *static void __exit tpuart_cleanup (void)*
  This is executed when the driver is unloaded. It releases the major number.

- *static irqreturn_t int_handler (int irq, void *filp, struct pt_regs *regs)*
  This is our interrupt service routine. In a big switch statement we distinguish the different interrupts. If we receive a receiver line status interrupt (RLSI) we just do some debug messages if an error is indicated but generally we ignore this kind of interrupt. We do the same with Modem status interrupt and time out interrupt. These interrupts are not important for our application.
  Our interest is directed towards the receiver interrupt (RHI) and the transmitter holding register empty interrupt (THI). The part for the receiver interrupt reads all bytes it gets in a big loop and can be separated in two main tasks. The first task is the reception of all parts of a frame. If we identify the start of a new frame (L_DATA_CTRL or L_LONG_DATA_CTRL) we start to collect the individual parts of the frame. Every byte of the frame is written into the RX-ringbuffer together with the necessary tel_flags and the receive bottom half (`rxstm`) is queued. This means that every second byte in the RX-ringbuffer is a flag which indicates different properties important for reassembling the message (if our device was addressed, if we have sent a busy ack, ...). We assume the end of a message if the supposed length is reached or a timeout is detected (which could cause some problems, read README Section 6 for details). After having received the address and we are addressed an acknowledge or busy (if the RX-queue is full) is sent immediately.
  The other main task of the receiver interrupt is to receive state messages, reset indications, error indications or data confirms. The last two indicate the end of a write cycle. In this case we wake up a sleeping write and a new telegram can be written. If we are in the busmonitor mode simply every received byte is written in the buffer and bmrecv is queued as bottom half.
  The transmit interrupt is simpler. If we transmit a telegram to the TPUART the first two bytes are written to the serial port by the transmit bottom half (transmit). This triggers a transmit interrupt and the remaining bytes are written to the serial port in the interrupt service routine. We always write two bytes (control + data byte) at once. After the last byte the SEND_IN_PROGRESS flag is cleared. If the heartbeat couldn't send the last state message it's now time to do it (indicated by the STATE_MESS_QUEUED flag).

- *int tpuart_open (struct inode *inode, struct file *filp)*
  When a program opens the driver the minor number determines which serial port is used(0-4). It is warranted that always only one process can open one device. In this function the necessary memory is allocated and all the initialization stuff is done (e.g for the `tpuart_t` structure). At last the TPUART is resetted and the heartbeat is started.

correspondingly great timeout value.

7 Unintentionally the heartbeat can provide assistance because the status bytes arrive in regular intervals.

- *int tpuart_release (struct inode *inode, struct file *filp)*
  This is the counterpart to `open`. The heartbeat is stopped and all memory is released.

- *ssize_t tpuart_read (struct file *filp, char *buffer, size_t count, loff_t *ppos)*
  The read call blocks until a telegram is detected or the heartbeat fails. If no failure occurs the last received (and not already consumed) telegram is copied to user space. If the read is non-blocking it returns immediately if no telegram exists in the buffer.

- *ssize_t tpuart_write (struct file *filp, const char *buffer, size_t count, loff_t *ppos)*
  The write call can sleep twice. First to wait until memory is available in the TX-queue and second to wait for a confirm or error code from the TPUART. If the call is non-blocking it returns immediately if the queue is full or after the message was queued. If in the blocking case a receive or transmit error was received the TPUART is resetted. Additionally it returns if the heartbeat detects a failure. No write call is possible in busmonitor mode.

- *int tpuart_fasync (int fd, struct file *filp, int mode)*
  This function allows asynchronous notification. You can find more information in [RC01], pp. 161. How to use this feature is desriped in README Section 5.6.

- *unsigned int tpuart_poll (struct file *filp, poll_table *wait)*
  The poll method is the back end of the system calls poll and select, both used to inquire if a device is readable or writable. You can find more information in [RC01], pp. 154.

- *int tpuart_ioctl (struct inode *inode, struct file *filp, unsigned int command, unsigned long arg)*
  The driver implements 21 different device-specific. You find a table with all commands in REAMDE, Section 5.2.

The telegram and device `tpuart_t` structure as well as all flags, TPUART service code definitions, serial port register definitions, EIB telegram fields and ioctl commands are defined in tpuart_main.h.

2.2 transmit

Transmit represents the bottom half of the transmission. It's queued in the „default“ work queue[8] of the kernel in the write function of tpuart_main. To ensure that all messages are sent in a row first the SEND_AND_WAIT flag is tested. It's set in transmit and cleared when the write cycle is finished (i. e. a confirm or error message was received by the ISR). If we are sending already the SEND_REQU flag becomes set. This indicates the ISR to queue the bottom half again after this write cycle. The DID_SEND flag allows the read bottom half to determine if a received message could have been sent by ourselves. The main task of transmit is to prepare the message for the TPUART. The TPUART awaits a control field followed by one byte of the real frame. This structure is generated here and stored in the TX-ringbuffer. The checksum is calculated and appended to the frame as last byte. The SEND_IN_PROGESS flag is set as long as we really send data (in contrast to SEND_AND_WAIT which is set until a response from the TPUART was received). Then we send the first two bytes to the serial port and provoke a transmitter interrupt with it. The ISR is responsible for sending the rest of the message.

2.3 rxstm

Rxstm is the bottom half of the read process. It's queued in the ISR every time we receive a part of a frame. Rxstm implements a state machine. It assembles the frame and analyzes the rx flags and tel_flags set by the ISR. It seems that the author of version 0.02 had more in mind originally with this two kind of flags. In fact we only need the rx flags because all tel_flags are mapped to rx flags for further usage. Among other things it controls the received checksum and if the message was sent by us. Only if no error occurred, no busy was sent and the message isn't from us the message is copied to the RX queue (as usual via the `deque_put_irq` function). We wake up a sleeping read process and the `read` function in tpuart_main can return the message to the user program. It's the best to read the source code for details of this state machine.

2.4 bmrecv

Bmrecv is the receive bottom half if we are in busmonitor mode. In busmonitor mode every received byte is put into the RX-ringbuffer and bmrecv is queued by the ISR. If we didn't receive the start of a message it puts a telegram of a single byte in the RX queue. Otherwise it assembles a whole telegram and put it in the queue. A process sleeping in read is woken up and the telegram can be returned.

8 You can find more information about work queues in [LWN], The Workqueue Interface

2.5 deque

This module represents the implementation of the RX and TX queue. The queue is controlled by a head and a tail pointer which are always in- or decremented by `elem_size` (which is set to the size of a telegram). There are methods for putting or getting a telegram in/from the queue (`deque_put` and `deque_get`) and methods for testing if the queue is full or empty (`deque_is_full` and `deque_is_empty`). Every method exists in two more variants: *`_irq` and *`_irqsave`. The methods ending with `_irq` disable interrupts using `spin_lock_irq`. It should only be used in situations in which you know that interrupts will not have already been disabled. Methods ending with `_irqsave` use `spin_lock_irqsave` which not only disables interrupts on the local processor but also stores the current interrupt state. The functions without these additional endings only use `spin_lock`. This could lead to a deadlock in co-operation with the ISR. You can find more information on spinlocks in [RC01], pp. 281.

2.6 heartbeat

This module implements a heartbeat to avoid infinite blocking if the TPUART is dead. It uses a kernel timer[9] to create a periodic interrupt (the period is determined by `heartbeat_interval`). During normal operation the heartbeat function is executed when the timer elapses. The TPUART is tested by sending a state message to it. The flag HB_STATE_AWAIT is set when sending the state message and cleared in the ISR when a state message has been received. The `heartbeat` function checks if the HB_STATE_AWAIT flag is cleared. If the flag is cleared everything is fine and the next state message is sent. This is done by the `sending_state_message` function which sends the state message immediately if no write process is in progress (indicated by the SEND_IN_PROGRESS transmission flag) or queues the state message to be sent in the ISR when the write process is over. If the HB_STATE_AWAIT flag is still set no state message was received. Then the DEAD flag is set and all sleeping read- or write processes are woken up. The `resetloop_request` function is now used as timer function and set to run immediately. This function sends a reset request to the TPUART and requests the timer to run in about 60ms the `resetloop_answer` function. This function checks if the reset was successful. If so the DEAD flag is cleared and the normal heartbeat (sending state messages) is used again. If not a next reset try is started in `heartbeat_interval`. A successfully reset could also be done by a reset via ioctl.
In heartbeat.h two macros are defined to stop and restart the heartbeat. They are used if a reset is done in write or via ioctl to avoid a mixture of two resets. When we enter the busmonitor mode the heartbeat is also disabled.

2.7 utility

This module is a mixture of useful utility functions. The main part of these functions is responsible for setting and clearing physical and group addresses and testing if we are addressed. The addresses are stored in a bit field where the corresponding bit is set if an address is assigned.
Additionally there are functions for detecting a timeout and initializing the serial port. The macro QUEUE_BH is defined in the header file to queue the bottom halves.

Literature

[RC01] Alessandro Rubini and Jonathan Corbet. Linux Device Drivers, 2nd Edition. O'Reilly, 2001.

[LWN] Jonathan Corbet. Porting device drivers to the 2.6 kernel.
http://lwn.net/Articles/driver-porting/

9 You can find more information about kernel timers in [RC01], pp. 200.