homematic IP

Dual Protocol Communication

- Linux drivers for Dual Protocol support -

Version 1.0.0

eQ-3 Entwicklung GmbH
Maiburger Str. 36
26789 Leer
Phone: +49 (0)491 6008 700
Fax: +49 (0)491 6008 99 700
Internet: www.eQ-3.de

Table of content

List of figures

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

List of tables

1 About

1.1 Scope

The wireless communication protocols of HomeMatic and Homematic IP are not compatible to each other; so that the devices of the different systems are not able to communicate directly. The connection between the systems shall be made by integrating both protocols in one central control unit. Due to resource limitations in the coprocessor the implementation of the protocols has to be split into two parts: coprocessor and multimacd (which is running on the main processor on a Linux system).

For seamless integration with existing software and for low-latency communication between host CPU and coprocessor, two custom drivers are needed.

In order to meet timing requirements a low-latency UART communication channel is necessary between multimacd and the coprocessor. The standard Linux serial drivers induce latency of up to several hundred milliseconds and thus cannot be used for this purpose. To overcome this limitation a custom “low latency serial” driver is needed.

A second “character loopback” driver is needed to allow the creation of virtual serial devices by multimacd in order to be used by the application layer (rfd and friends). Using this approach, the communication with multimacd doesn’t differ from the communication with a “real” coprocessor from the point of view of the application layer.

This document contains detailed information and implementation hints for both drivers.

1.2 Purpose of this document

This document is designed primarily for the use in the product design and implementation, to be used as part of building a dual protocol communication platform.

1.3 Not purpose of this document

This document doesn’t contain the descriptions of the individual firm- and software-implementation. Furthermore this document is NOT to be used in the context of marketing.

1.4 Document history

Version	Date	Editor	Comment
1.0.0	10-Aug-2015	L. Reemts	Initial release.

Table 1: Document history

1.5 Conventions in this document

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [Bradner, Scott, “Key words for use in RFCs to Indicate Requirement Levels”, BCP 14, RFC 2119, March 1997].

The word “Byte” is used to denote an 8-bit value or field.

2 Low latency serial driver

2.1 Serial communication parameters

2.1.1 Constraints

- given by existing coprocessor hardware
- given by existing coprocessor implementations
- reuse existing coprocessor bootloaders
- reuse existing coprocessor update tools
- meet latency requirements

2.1.2 Parameters

2.1.2.1 UART settings

- Baudrate: 115200 bit/s
- Databits: 8
- Parity: None
- Stopbits: 1

Baudrate	115200
Databits	8
Parity	None
Stopbits	1

Table 2: UART Settings

2.1.2.2 Character and frame timing

The following timings are guaranteed by the coprocessor in the direction from coprocessor to driver. The purpose is to enable the driver to detect the end of a serial frame using an RX timeout interrupt.

- Maximum time between consecutive bytes of the same serial frame: 30 bit times
- Minimum time between consecutive frames: 40 bit times

2.2 Driver operation

2.2.1 General

The low latency serial driver is responsible for one or more UART ports of the SoC. For each supported UART port there SHALL be a kernel configuration option for assigning the low latency driver or the standard TTY driver to the respective port.
For each UART port under control of the low latency driver one device node in the /dev directory SHALL be created.

2.2.2 Userspace connections

The driver MUST be able to handle multiple concurrent connections from user space. Each user space connection is started by a call to open().

2.2.2.1 Connection priorities

Each connection has a 32bit priority value assigned to it. Initially (right after opening) the priority is 0. It can be set to an other value via ioctl. Priority values are only important for write operations. If while a write operation is in progress a write from a higher priority connection is requested, the current write is aborted and the higher priority write is started instead. By virtue of application level framing, the partially transferred frame from the interrupted write operation is implicitly discarded by the coprocessor.

2.2.3 Open

When the first user space connection to a port is opened, the respective port is enabled. An instance of `struct per_connection_data` is created and stored in the corresponding `struct file::private_data`.
The instance of `struct per_connection_data` is used by all subsequent driver API calls for identifying the connection.

2.2.4 Close

When the last user space connection to a port is closed, the respective port is disabled. The instance of `struct per_connection_data` identifying the connection is deleted.

2.2.5 Write

Write operations are synchronous waiting until the complete buffer is transferred. write() MUST perform the following steps

- wait until the transmitter becomes available (because a same or higher priority write might be in progress)
- start writing to the UART FIFO
- wait until the last byte has been transferred to the FIFO

If a write operation was interrupted, the number of bytes actually transferred to the TXD line must be returned. It is the responsibility of the user space application to repeat the complete write operation in this case.

2.2.6 Read operation

Read operations are not required to arbitrate between multiple connections. The driver may assume that only one call to read() is active at any given time.
The driver SHOULD implement one read buffer per UART port. Read MUST be able to handle blocking I/O as well as non blocking I/O.

2.2.7 Ioctl operation

The following ioctls must be implemented:

- `IOCSPRIORITY = _IOW('u',  1, unsigned long)`
  Sets the priority of the current connection to the value passed
- `IOCGPRIORITY = _IOR('u',  2, unsigned long)`
  Queries the priority of the current connection

The following ioctls are a subset of standard TTY ioctls and SHOULD be implemented for better compatibility to special purpose user space software, especially software written in Java and using the RXTX communication library:

- `TCSETS`
  Set the termios settings for the port. The termios setting SHOULD be stored and returned when `TCGETS` is requested.
- `TCGETS`
  Get the termios settings for the port. The stored settings from the last `TCSETS` SHOULD be returned.
- `TIOCINQ`
  Get the size of the RX queue. The driver SHALL return the number of bytes that can be read without blocking.
- `TIOCEXCL`
  Request exclusive use. The driver SHOULD report successful completion on this ioctl and is NOT REQUIRED to do anything else.
- `TCFLSH`
  Flush the output buffer. The driver SHOULD report successful completion on this ioctl and is NOT REQUIRED to do anything else.
- `TIOCMGET`
  Get the states of the modem control lines. The driver SHOULD return `TIOCM_DSR | TIOCM_CD | TIOCM_CTS`
- `TIOCMSET`
  Set the states of the modem control lines. The driver SHOULD report successful completion on this ioctl and is NOT REQUIRED to do anything else.

2.2.8 Poll operation

If either no write or a lower priority write is in progress, poll() SHOULD return `POLLOUT | POLLWRNORM` indicating writable. Alternatively, poll() MAY always return `POLLOUT | POLLWRNORM`.

If at least one character is in the RX queue, poll() SHALL return `POLLIN | POLLRDNORM` indicating readable.

Separate wait queues SHALL be used for read and write.

2.3 Timing requirements

2.3.1 Definitions

- TX latency
  Delay time from driver write operation until the start of first byte on the TXD line
- RX latency
  Delay time from end of the last frame byte on the RXD line until the user space application is notified

2.3.2 Requirements

The sum of TX latency and RX latency should be below 5ms.

2.4 Supporting another hardware platform

2.4.1 Porting the reference driver

When implementing a low latency driver for a new platform, the driver implementation used in the HomeMatic CCU2 MAY be used as a reference. The CCU2 driver targets the Freescale i.MX28 application UART (AUART). It is named `mxs_raw_auart.c`
The reference driver directly controls the AUART registers and FIFOs. No DMA is used because further latencies would be incurred.
As first step in porting “mxs_raw_auart” SHOULD be searched and replaced by something better describing the new target hardware.
The following functions are hardware dependent and MUST be adapted:

- `mxs_raw_auart_stop_txie()`
  Masks the TX interrupt. After calling this function, no TX interrupt will occur.
- `mxs_raw_auart_tx_chars()`
  Fills the TX hardware FIFO. The driver uses the define `TX_CHUNK_SIZE` for the maximum number of bytes put into the TX FIFO at once. Increasing `TX_CHUNK_SIZE` will also increase the TX latency for a high priority frame interrupting a low priority frame. Decreasing `TX_CHUNK_SIZE` will increase CPU overhead. For the i.MX28 hardware there is also a dependency between `TX_CHUNK_SIZE` and the TX FIFO interrupt threshold. `TX_CHUNK_SIZE` must be big enough to make sure that the TX FIFO is always filled beyond the threshold, otherwise the TX interrupt might not be triggered.
- `mxs_raw_auart_rx_chars()` and `mxs_raw_auart_rx_char()`
  These functions empty the RX FIFO.
- `mxs_raw_auart_irq_handle()`
  Interrupt dispatcher. Checks if RX or TX interrupt was triggered and calls `mxs_raw_auart_rx_chars()` and `mxs_raw_auart_rx_chars()` as needed.
- `mxs_raw_auart_reset()`
  Resets the UART controller. Called on initialization of the driver.
- `mxs_raw_auart_startup()`
  Enables a UART port. Called from `mxs_raw_auart_open()` for the first user space connection.
- `mxs_raw_auart_shutdown()`
  Disables a UART port. Called from `mxs_raw_auart_close()` when the last user space connection is closed.
- `mxs_raw_auart_start_tx()`
  Enabled the transmitter, enables the TX FIFO interrupt and calls `mxs_raw_auart_tx_chars()` in order to transfer the first chunk of TX bytes to the FIFO.

- `mxs_raw_auart_read_procmem()`
  Optional function exporting debug information via the proc filesystem. In this function the first block outputting the AUART registers MUST be changed.

2.4.2 Build and runtime integration

After porting the driver source code, it must be made sure that the driver is compiled and called. This involves the following steps:

- Add configuration options for the new driver to `drivers/char/Kconfig`.
- Add the new driver to `drivers/char/Makefile`.
- Create a platform device for every supported port depending on the new configuration options. Make sure that the new driver doesn't possibly conflict with the standard UART driver.
  Depending on kernel version and platform the platform device has to be created in the board ".c" file or via device tree.

2.4.2.1 Kconfig sample

For the i.MX28 driver on the CCU2, the following options were added to `drivers/char/Kconfig`

```
config MXS_RAW_AUART
       tristate "iMX28 AUART raw driver"
       depends on ARCH_MXS
       help
           This driver supports the MXS Application UART (AUART) port as
raw character device.

config MXS_RAW_AUART_PORT0
       bool "Use iMX28 AUART raw driver for AUART0"
       depends on MXS_RAW_AUART
        default n
       help
           Attach the MXS Application UART raw character device to AUART0

config MXS_RAW_AUART_PORT1
       bool "Use iMX28 AUART raw driver for AUART1"
       depends on MXS_RAW_AUART
        default y
       help
           Attach the MXS Application UART raw character device to AUART1
```

2.4.2.2 Makefile sample

For the i.MX28 driver on the CCU2, the following line was added to `drivers/char/Makefile`

```
obj-$(CONFIG_MXS_RAW_AUART)      += mxs_raw_auart.o
```

2.4.2.3 Board file sample

For the i.MX28 driver on the CCU2, the following code was added to `arch/arm/mach-mxs/eq3-ccu2/ccu2.c`

2.4.2.3.1 Binding of hardware resources

```
static struct resource        mxs_raw_auart0_resources[] = {
      {
            .start            =  MX28_AUART0_BASE_ADDR,
            .end        =  MX28_AUART0_BASE_ADDR + 0x100 - 1,
            .flags            = IORESOURCE_MEM,
      }, {
            .start            = MX28_INT_AUART0,
            .end        = MX28_INT_AUART0,
            .flags            = IORESOURCE_IRQ,
      }
};

static struct resource        mxs_raw_auart1_resources[] = {
      {
            .start            =  MX28_AUART1_BASE_ADDR,
            .end        =  MX28_AUART1_BASE_ADDR + 0x100 - 1,
            .flags            = IORESOURCE_MEM,
      }, {
            .start            = MX28_INT_AUART1,
            .end        = MX28_INT_AUART1,
            .flags            = IORESOURCE_IRQ,
      }
};
```

2.4.2.3.2 Definition of the platform device

```
static struct platform_device       ccu2_devices[] = {
[…]
#if defined(CONFIG_MXS_RAW_AUART) || defined(CONFIG_MXS_RAW_AUART_MODULE)
    #if defined(CONFIG_MXS_RAW_AUART_PORT0)
      {
            .name       = "mxs-raw-auart",
            .id         = 0,
        .resource = mxs_raw_auart0_resources,
            .num_resources    = ARRAY_SIZE(mxs_raw_auart0_resources),
      },
    #endif
    #if defined(CONFIG_MXS_RAW_AUART_PORT1)
      {
            .name       = "mxs-raw-auart",
            .id         = 1,
        .resource = mxs_raw_auart1_resources,
            .num_resources    = ARRAY_SIZE(mxs_raw_auart1_resources),
      },
    #endif
#endif
```

```
};
```

2.4.2.3.3 Disabling of the standard UART driver depending on new configuration options

```
#if (defined(CONFIG_MXS_RAW_AUART) || defined(CONFIG_MXS_RAW_AUART_MODULE))
    #if !defined(CONFIG_MXS_RAW_AUART_PORT0)
        mx28_add_auart0();
    #endif
    #if !defined(CONFIG_MXS_RAW_AUART_PORT1)
        mx28_add_auart1();
    #endif
#else
    mx28_add_auart0();
    mx28_add_auart1();
#endif
```

3 Character loopback driver

3.1 Supported platforms

The character loopback driver is hardware independent. Therefore the reference implementation `eq3_char_loop.c` can be used on any platform with no porting required. It SHOULD be copied to `drivers/char/ eq3_char_loop.c`.

3.2 Operation overview

When loaded, the loopback driver creates a master device node named `/dev/eq3loop`. A user space master application such as multimacd can use this master device for creating slave devices. With the current setting (`EQ3LOOP_NUMBER_OF_CHANNELS`) up to 4 slave devices are supported. When a slave device is created by the master application, a new device node with a name supplied by the application shows up in the `/dev` directory. A slave application (e.g. rfd) can open the slave device and communicate with the master application in the same way it would communicate through a serial device. For supporting serial communication libraries expecting a real serial port, a subset of standard TTY iocls is implemented by the loopback driver.
Using the loopback driver from the master application typically involves the following steps for each slave device:

- Open the master device
- Call ioctl `EQ3LOOP_IOCSCREATESLAVE` with the name of the slave device as argument
- Enter a `select()` or `poll()` loop. Within the loop:
  - If select/poll indicates readable, the slave device is open and the slave has sent data. Read the data and process it.
  - If select/poll indicates writeable, the slave device is open and buffer space is available for writing. Write pending data to the slave.
  - If select/poll indicates an exception, the slave device has been opened or closed. Call ioctl `EQ3LOOP_IOCGEVENTS` to query the open state.

3.3 Build and runtime integration

After copying the driver source code, it must be made sure that the driver is compiled and called. This involves the following steps:

- Add configuration options for the loopback driver to `drivers/char/Kconfig`.
- Add the loopback driver to `drivers/char/Makefile`.

3.3.1 Kconfig sample

For the loopback driver on the CCU2, the following options were added to `drivers/char/Kconfig`

```
config EQ3_CHAR_LOOPBACK
       tristate "eQ3 char loopback device"
       help
```

```
            This driver provides a char loopback device used by eQ-3
daemons.
```

3.3.2 Makefile sample

For the loopback driver on the CCU2, the following line was added to `drivers/char/Makefile`

```
obj-$(CONFIG_EQ3_CHAR_LOOPBACK)     += eq3_char_loop.o
```