RS485 Serial Communications

1. Introduction

EIA-485, also known as TIA/EIA-485 or RS-485, is a standard defining the electrical characteristics of drivers and receivers for use in balanced digital multipoint systems. This standard is widely used for communications in industrial automation because it can be used effectively over long distances and in electrically noisy environments.

2. Hardware-related Considerations

Some CPUs/UARTs (e.g., Atmel AT91 or 16C950 UART) contain a built-in half-duplex mode capable of automatically controlling line direction by toggling RTS or DTR signals. That can be used to control external half-duplex hardware like an RS485 transceiver or any RS232-connected half-duplex devices like some modems.

For these microcontrollers, the Linux driver should be made capable of working in both modes, and proper ioctls (see later) should be made available at user-level to allow switching from one mode to the other, and vice versa.

3. Data Structures Already Available in the Kernel

The Linux kernel provides the serial_rs485 structure (see [1]) to handle RS485 communications. This data structure is used to set and configure RS485 parameters in the platform data and in ioctls.

The device tree can also provide RS485 boot time parameters (see [2] for bindings). The driver is in charge of filling this data structure from the values given by the device tree.

Any driver for devices capable of working both as RS232 and RS485 should implement the rs485_config callback in the uart_port structure. The serial_core calls rs485_config to do the device specific part in response to TIOCSRS485 and TIOCGRS485 ioctls (see below). The rs485_config callback receives a pointer to struct serial_rs485.

4. Usage from user-level

From user-level, RS485 configuration can be get/set using the previous ioctls. For instance, to set RS485 you can use the following code:

```
#include <linux/serial.h>

/* Include definition for RS485 ioctls: TIOCGRS485 and TIOCSRS485 */
#include <sys/ioctl.h>

/* Open your specific device (e.g., /dev/mydevice): */
int fd = open ("/dev/mydevice", O_RDWR);
if (fd < 0) {
        /* Error handling. See errno. */
}

struct serial_rs485 rs485conf;

/* Enable RS485 mode: */
rs485conf.flags |= SER_RS485_ENABLED;

/* Set logical level for RTS pin equal to 1 when sending: */
rs485conf.flags |= SER_RS485_RTS_ON_SEND;
/* or, set logical level for RTS pin equal to 0 when sending: */
rs485conf.flags &= ~(SER_RS485_RTS_ON_SEND);

/* Set logical level for RTS pin equal to 1 after sending: */
rs485conf.flags |= SER_RS485_RTS_AFTER_SEND;
/* or, set logical level for RTS pin equal to 0 after sending: */
rs485conf.flags &= ~(SER_RS485_RTS_AFTER_SEND);

/* Set rts delay before send, if needed: */
rs485conf.delay_rts_before_send = ...;

/* Set rts delay after send, if needed: */
rs485conf.delay_rts_after_send = ...;

/* Set this flag if you want to receive data even while sending data */
rs485conf.flags |= SER_RS485_RX_DURING_TX;

if (ioctl (fd, TIOCSRS485, &rs485conf) < 0) {
        /* Error handling. See errno. */
```

```
}

/* Use read() and write() syscalls here... */

/* Close the device when finished: */
if (close (fd) < 0) {
        /* Error handling. See errno. */
}
```

5. References

[1] include/uapi/linux/serial.h

[2] Documentation/devicetree/bindings/serial/rs485.txt