RS422 and RS485 Application Note

For

BETA
TM1 / TM3 / XE1 / DB1

Document Reference: RS485_BETA_APP_NOTE

Document Issue: 1.2

Author: D Robinson

1 Introduction

RS422 and RS485 are popular bus architectures used in embedded / industrial systems.

Unlike RS232, both RS422 and RS485 use differential signalling which increases noise immunity and supports much longer cable lengths between nodes in a system.

RS422 networks are typically full duplex, point to point implementations in the same way that RS232 networks are. RS485 networks tend to be half duplex, multi node implementations. As an RS485 network is half duplex, with multiple nodes capable of transmitting data, it is important to avoid bus contention. Only one node should transmit data onto the bus at a time, with all other nodes configured in receive only mode.

The BCT range of BETA HMIs and embedded computer modules provide support for RS232, RS422, and RS485. This document focuses on how the RS422, and RS485 ports are implemented from a hardware perspective, and how they can be controlled from software.

2 Hardware Implementation

The RS422/RS485 transceiver implemented on generic BCT hardware is a MaxLinear SP3076E and is full duplex. This is ideal for an RS422 bus, however to use the transceiver on an RS485 network, the hardware must be configured to be half duplex. This can be achieved by externally connecting RX_P to TX_P, and RX_N to TX_N.

2.1 TMx Hardware implementation

The RS422 and RS485 implementation on host boards featuring a TM1, TM2, TM3 and future modules can be seen in figure 1.

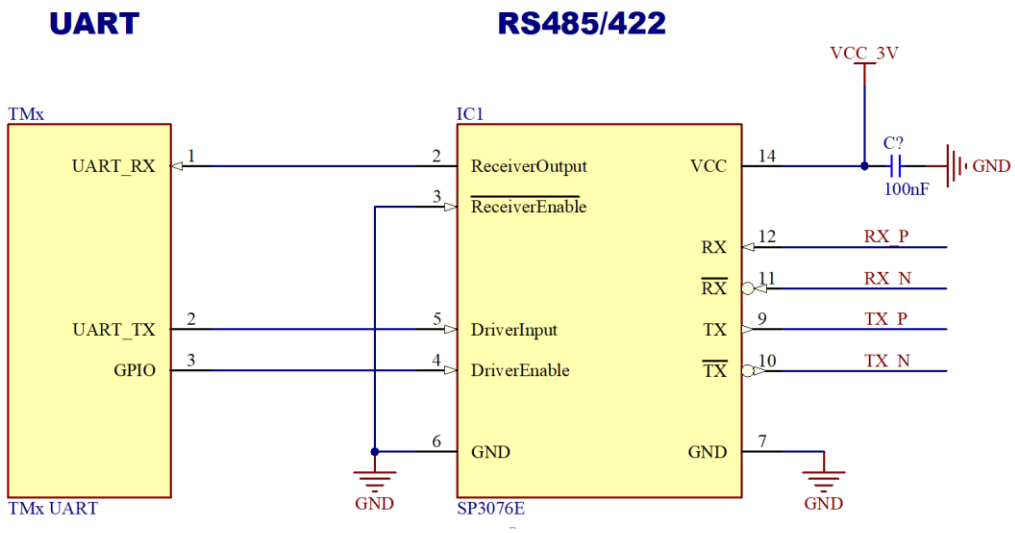

Figure 1 – RS422/RS485 implementation with hardware featuring a TMx module

Note that the ReceiverEnable# signal is hard wired to ground meaning the receiver is permanently enabled. For RS422 this is acceptable as the interface is operating full duplex. However in the RS485 use case this means that all data transmitted will be received back. This is often referred to as local loopback. See the software section for details on how this can be handled.

The DriverEnable signal can be controlled from software using a GPIO pin from the TMx module. For RS422 communication it is okay to permanently enable the transmitter as RS422 is operating full duplex. When operating on an RS485 network it is important that the DriverEnable signal is only set high while the UART is transmitting data to avoid contention, as the RS485 specification only allows one transmitter active on the bus at any one time.

2.2 XE1 Hardware implementation

The RS422 and RS485 implementation on XE1 can be seen in figure 2.

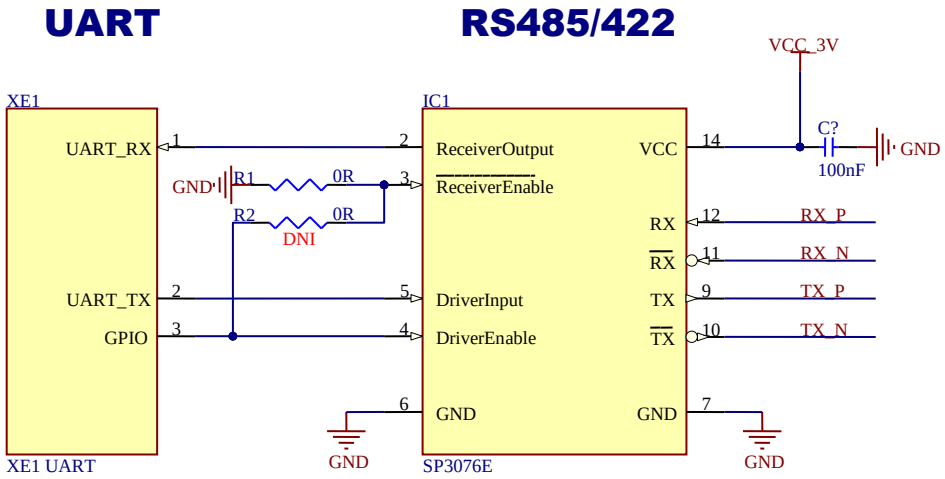

Figure 2 - RS422/RS485 implementation with hardware featuring XE1

Note that the ReceiverEnable# signal is hard wired to ground by default meaning the receiver is permanently enabled. For RS422 this is acceptable as the interface is operating in full duplex. However in the RS485 use case this means that all data transmitted will be received back. This is often referred to as local loopback. See the software section for details on how this can be handled.

XE1 also has a hardware population option which allows the ReceiverEnable# signal to be attached to the DriverEnable signal. The XE1 can be populated with R2 fitted instead of R1 which effectively disables the local loopback feature when operating on an RS485 bus.

The DriverEnable signal can be controlled from software using the DTR modem control line of the UART. For RS422 communication it is acceptable to permanently enable the transmitter as RS422 is operating full duplex. When operating on an RS485 network it is important that the DriverEnable signal is only set high while the UART is transmitting data to avoid contention, as the RS485 specification only allows one transmitter active on the bus at any one time.

2.3 DB1 Hardware implementation

The RS422 and RS485 implementation on DB1 can be seen in figure 3.

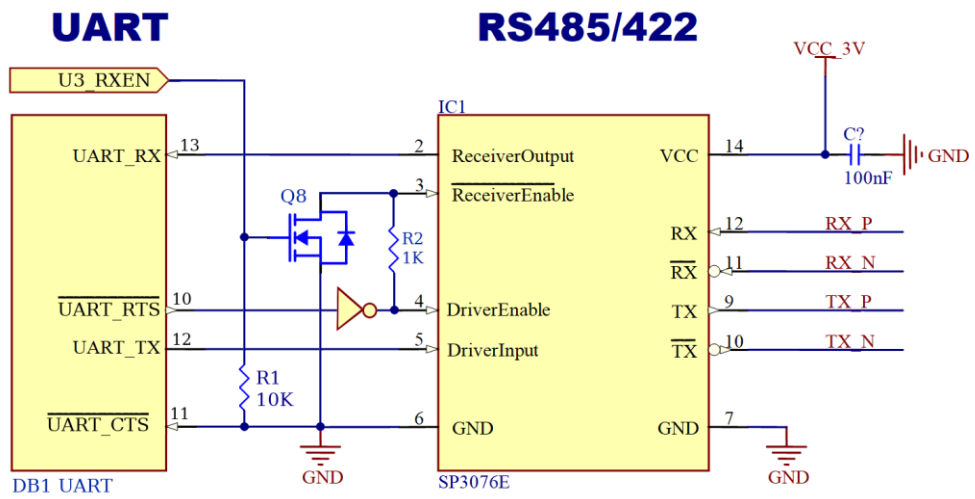

Figure 3 - RS422/RS485 implementation with hardware featuring DB1

ReceiverEnable# is driven via hardware flow control. Data transmission from DB1 starts by UART_RTS# (Request to Send) asserting the line which enables DriverEnable and disables ReceiverEnable. UART_CTS# (Clear to Send) is tied to ground therefore the UART IC can transmit data immediately over TX. During the transmission the UART_RTS# stays active (low) blocking the reception - ReceiverEnable# is pulled high by the inverter. After the transmission the UART_RTS# goes inactive (high) disabling the DriverEnable and enabling the ReceiverEnable#. That's suitable for RS485 half duplex communication.

To enable full duplex RS422 communication the ReceiverEnable# must be held low. That is achieved by Q8 Mosfet which connects ReceiverEnable# to Ground when the Gate of the Mosfet is driven high by U3_RXEN GPIO.

3 Software Control

3.1 RS422/RS485 Protocol

The RS422 and RS485 specifications do not define a protocol. They are purely an electrical specification of the physical layer. Data rates, data format, handshaking, commands, responses, CRC, turnaround times are all software defined. A successful RS422/RS485 network requires all nodes to implement the same software protocol.

Some protocols that employ RS485 signalling include Modbus RTU, Modbus ASCII, PROFIBUS, DMX, ADAM.

3.1.1 Simple Protocol Overview

Many protocols define a system where a single master node and multiple slave nodes exist. To communicate with slaves the master node sends a command containing an 8 bit slave address. Only the slave node that matches the unique address will respond to the request.

Using Modbus RTU protocol as an example, the command to request 2 holding register from slave address 1 would be:

Data stream request from master:

Byte	0	1	2	3	4	5	6	7
Data (Hex)	01	03	00	00	00	02	C4	0B
Description	Address	Command	Register Address		Register Count		Checksum	

Data stream response from slave:

Byte	0	1	2	3	4	5	6	7	8
Data (Hex)	01	03	04	00	06	00	05	DA	31
Description	Address	Command	Count	Register Data 0		Register Data 1		Checksum	

3.1.2 High level protocol implementation

Given the example in the previous section from the perspective of the master node, the software flow would look similar to the following

- Open the UART
- Configure UART E.g. 9600,8,n,1
- Set transmit enable high
- Transmit message {01,03,00,00,00,02,C4,0B}
- Wait for transmission complete
- Set transmit enable low
- Turnaround time delay
- Read response
- Discard what was transmitted if local loopback enabled
- Interpret response

3.2 Software implementation with TM1, TM3, XE1 and DB1 hardware

3.2.1 UART

Transmitting and receiving data via the RS422/RS485 transceiver is the same on all TMx, DB1 and XE1 hardware. The application software simply opens a UART (COM port), and then uses standard operating system functions to configure the port and read/write data.

The UART port mappings for TM1, TM3, DB1 and XE1 for the supported operating systems can be found in the following table.

Operating System	Windows	Linux / Android
TM1	-	/dev/ttymxc2

TM3	-	/dev/ttyS3
XE1	COM3	/dev/ttyS2
DB1	-	/dev/ttySC0

3.2.2 Transmit Control

Note that the method of controlling the transmit enable pin varies between the hardware type and the operating system.

3.2.2.1 TM1/HB5 Transmit Control

For TMx modules the transmit enable signal is controlled by a GPIO . To control the GPIO from within Linux the GPIO sysfs system can be used. An example for TM1 follows:

```
echo 67 >> /sys/class/gpio/export
echo out >> /sys/class/gpio/gpio67/direction
echo 1 >> /sys/class/gpio/gpio67/value
echo 0 >> /sys/class/gpio/gpio67/value
```

https://www.kernel.org/doc/Documentation/gpio/sysfs.txt

See appendix A for a sample Linux application that controls GPIO 67 as a transmit enable pin.

TM3 module uses GPIO 386 for the same purpose.

To control the transmit enable signal in Android see the TM1HB5-AC1-RS485Demo sample application within the Android SDK for TM1.

3.2.2.2 XE1 Transmit Control

For XE1 the transmit enable signal is controlled by the DTR modem control line of the UART.

To control the signal from within Windows the EscapeCommFunction can be used.

https://docs.microsoft.com/en-us/windows/desktop/api/winbase/nf-winbase-escapecommfunction

To control the signal from within Linux the ioctl functions TIOCMGET and TIOCMSET can be used.

http://man7.org/linux/man-pages/man4/tty_ioctl.4.html

3.2.2.3 DB1 Transmit Control

On DB1 board the transmit enable signal is fully controlled by the RTS# UART control line and it can't be affected by software intervention. Therefore, DB1's RS485 UART is always ready to transmit. DB1 controls the ReceiverEnable# via U3_RXEN GPIO which can enable full duplex mode (see section 2.3 for more information). This GPIO is driven by the Linux kernel UART driver and is enabled when the user space application configures the serial port with SER_RS485_RX_DURING_TX flag. See Appendix A – Linux sample application for more information how to apply the flag.

3.2.3Transmit control timing and slave turnaround time

For point to point RS422 communication the timing of the transmit enable is not important. The transmit enable merely needs to be enabled before transmission begins.

For RS485 communication however the timing of the transmit enable is very important. If the transmit enable is not enabled for long enough the message may not get fully transmitted. If the transmit enable is enabled for too long the slave may not be able to send a response due to bus contention (i.e. two nodes in transmit at any time).

The slave node turnaround time is a key factor that determines how precise the timing of the transmit enable signal needs to be. The turnaround time is protocol and node specific, and isn't always documented. Typical Modbus turnaround times are between 100ms and 200ms which normally provides ample time to disable the transmitter before a slave node responds. Other protocols may have a turnaround time in the order of a few milliseconds or even microseconds. In these cases the timing of the transmit enable needs to be more precise.

3.2.3.1 Transmit control timing using tcdrain

The Linux tcdrain function waits for the UART FIFOs to be empty. By waiting for the tcdrain function to return we can be confident that the message was fully transmitted, before disabling the transmit enable signal.

https://linux.die.net/man/3/tcdrain

Using tcdrain the software flow would be:

- Set transmit enable high
- Transmit message {01,03,00,00,00,02,C4,0B}
- Call tcdrain
- Set transmit enable low
- Delay for slave node turnaround
- Read response.

During testing we found that although this method was reliable at determining when the FIFOs were empty, the tcdrain function added a lot of latency when transfer sizes were small. For example, with a transfer that takes less that 2ms the tcdrain function would not return for 18.2ms. See Figure 3.

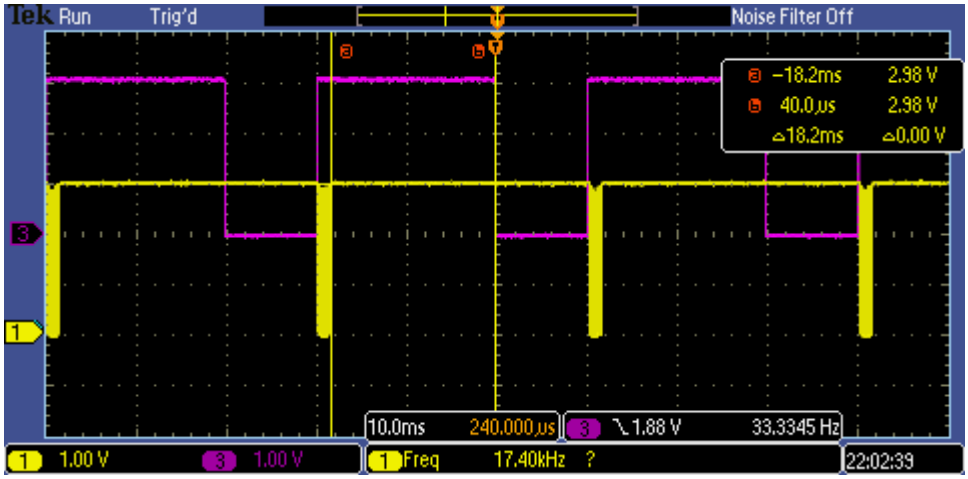

Figure 3 - Transmit enable timing with tcdrain (18.2ms)

Our conclusion from this test is that the tcdrain method is fine for transmit control timing when the slave node turnaround time is greater than 20ms.

3.2.3.2 Transmit control timing using calculation

By calculating how long a message should take to transfer out of the UART FIFO we can more accurately control the transmit enable signal.

The following calculation can be used as guidance to determine the message transfer time.

CharactersPersecond = BaudRate / SymbolBitLength
E.g. 9600 / 10 (8 data bits, 1 start bit, 1 stop bit, no parity) = 960 characters per second

CharactersPerMilisecond = CharactersPersecond / 1000
960 / 1000 = 0.96 characters per milisecond

msPerSymbol = 1 / CharactersPerMilisecond;
1 / 0.96 = 1.04 ms per symbol

MessageTransferTime = (1.04 * MessageLength) + 1

Using calculated transfer time the software flow would be:

- Set transmit enable high
- Transmit message {01,03,00,00,00,02,C4,0B}
- usleep(MessageTransferTime)
- Delay for slave node turnaround
- Set transmit enable low
- Read response.

During testing we found that this method was reliable at communicating with slave nodes having a turnaround time greater than 2ms. See figure 5. With this method we measured 432 microseconds of jitter in the transmit enable signal transitions. See figure 4.

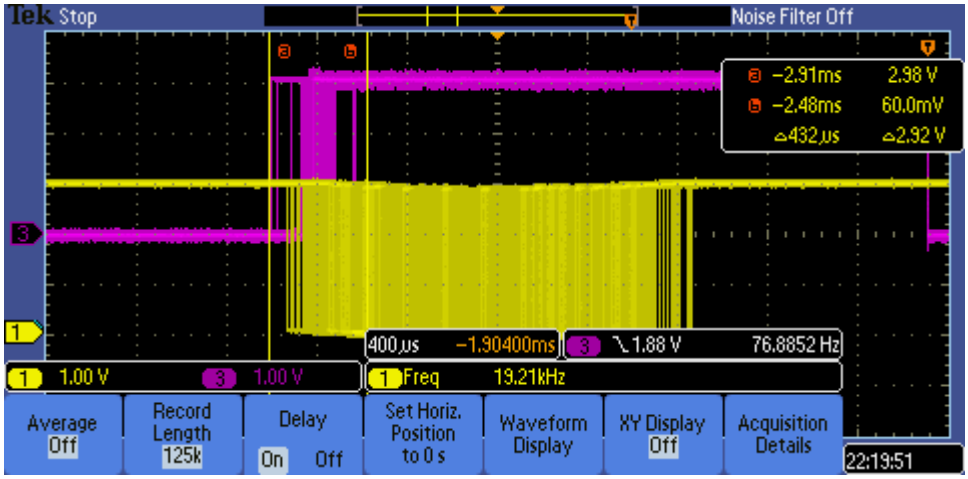

Figure 4 - RS485 transmit enable control within 1.27ms of message transmission

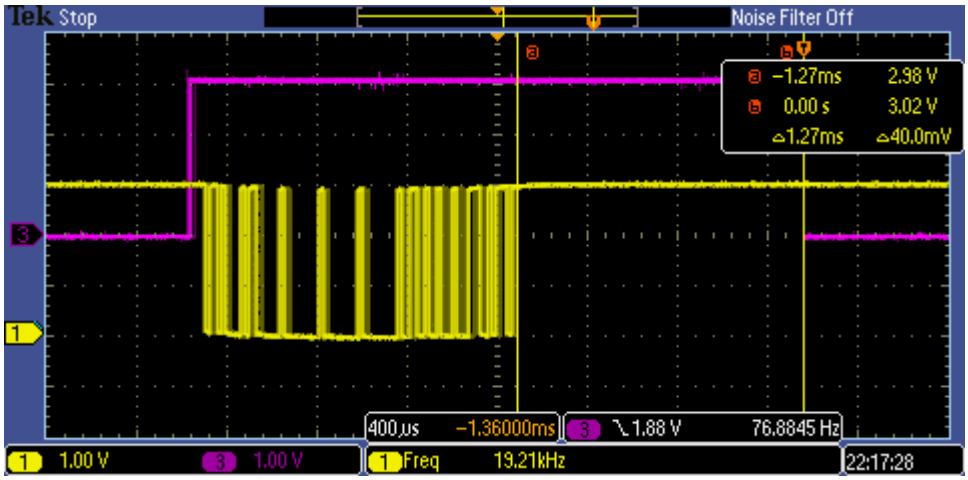

Figure 5 - 432us jitter when controlling the transmit enable signal using calculated timings

3.2.3.3 Automatic transmit control within the Linux kernel

We have updated our TM1, TM3 and DB1 UART driver for both Linux and Android to support the Linux serial-rs485 API.

https://www.kernel.org/doc/Documentation/serial/serial-rs485.txt

This API allows the UART to be put into an RS485 mode where the transmit control signal is automatically enabled and disabled. The benefits of this are:

- Control of the transmit enable signal is automatic from application software perspective.
- The timing of the transmit enable signal is far more accurate as it is based on hardware interrupts rather than calculated delays.

Using automatic transmit control the software flow would be:

- Enable RS485 UART mode

- Transmit message {01,03,00,00,00,02,C4,0B}
- Delay for slave node turnaround
- Read response.

During testing we found that this method of transmit control was reliable at communicating with slave nodes having a turnaround time greater than 59us. This was calculated with the traces captured in figures 6 and 7.

Figure 6 shows the transmit enable going low 35.2us after the final rising edge of a transmitted message (beginning of the final stop bit). As the baud rate of the captured transmission was 115200, the actual latency from the end of transmission to the transmit enable going low is 26.5us.

Figure 7 shows 32us of jitter in the transmit enable signal transitions.

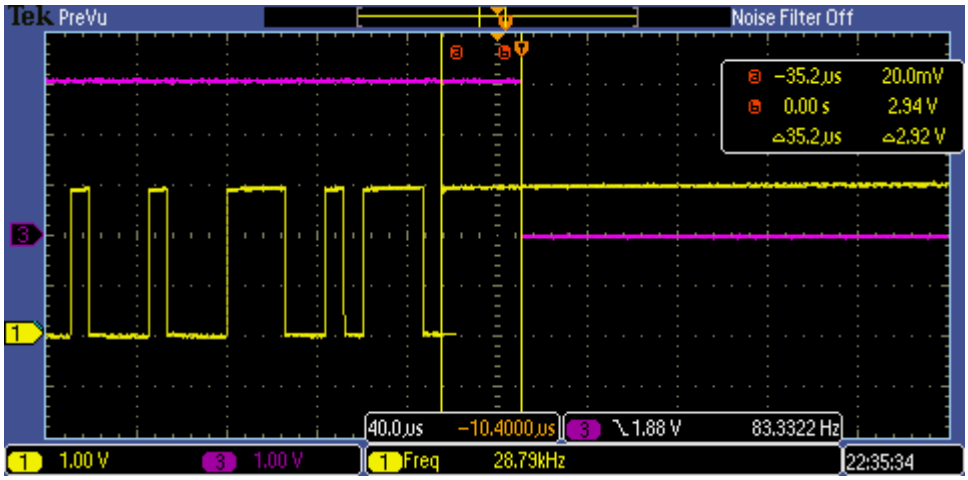

Figure 6 - RS485 transmit enable control within 35.2us of message transmission

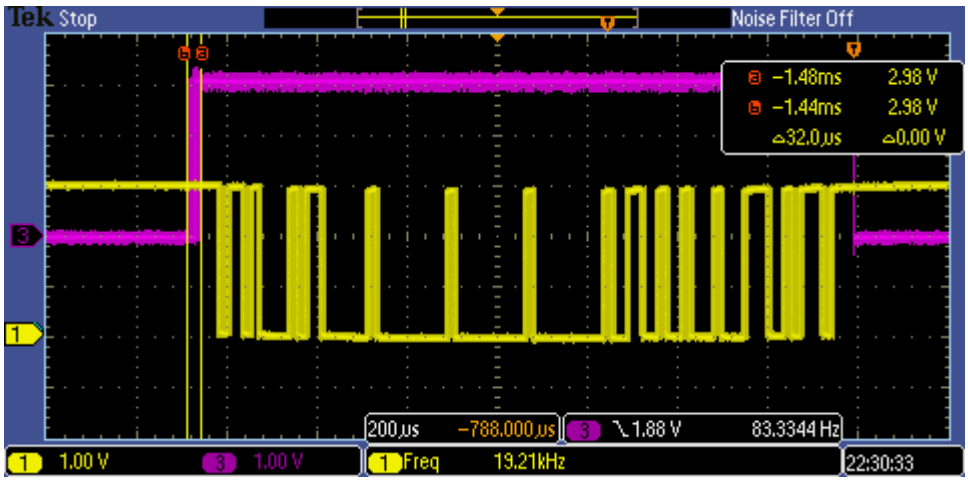

Figure 7 - 32us jitter when automatic transmit enable signal used

This feature is compatible with the following Linux kernel versions and Android Version:

Linux 3.14.28 built later than Thu Apr 11 12:21:33 2019

Linux 4.9.88

Android 4.4.3 builds V1.21

3.2.3.3 Transmit control by product

The following table identifies which transmit control methods are supported by each product

Product / Control Type	**Automatic control**	**tcdrain**	**Calculated**
TM1	Yes	Yes	Yes
TM3	Yes	Yes	Yes
XE1	No	Yes	Yes
DB1	Yes, always on	N/A	N/A

3.2.4 Local Loopback

As identified in section 2, our hardware solutions are designed to be flexible and support both RS422 and RS485 operation. The transceiver is full duplex to support RS422, and an externall loop back is required to support RS485. If ReceiverEnable# signal on RS485 chip is tied to ground (as implemented on TM1, TM3 and XE1, see section 2 Hardware Implementation) the side effect of externally looping-back transmitter and receiver is that any data transmitted is also received back. This section discusses the options for handling this.

Note that DB1 by default does not receive transmitted data due to different hardware configuration. If looping-back data feature is required - for example for software compatibility reasons - the serial port on DB1 needs to be configured with SER_RS485_RX_DURING_TX RS485 flag enabled (see Appendix A – Linux sample application).

3.2.4.1 Software ignore or discard

With a default hardware configuration and manual transmit control any data transmitted will be received back (applies to TMx and XE1). Using the same Modbus example as section 3.1.1 the transmit and receive buffers of the UART would read as follows.

UART Transmit buffer:

Byte	0	1	2	3	4	5	6	7
Data (Hex)	01	03	00	00	00	02	C4	0B
Description	Address	Command	Register Address		Register Count		Checksum	

UART Receive buffer:

Byte	0	1	2	3	4	5	6	7
Data (Hex)	01	03	00	00	00	02	C4	0B
Description	Copy of what we transmitted							

Byte	8	9	10	11	12	13	14	15	16
Data (Hex)	01	03	04	00	06	00	05	DA	31
Description	Address	Command	Count	Register Data 0		Register Data 1		Checksum	

As we know that local loopback is configured it is expected that the transmitted data is received back. Software can easily be coded to either discard the transmitted data or ignore it.

3.2.4.1 Hardware disable of local loopback (XE1 only)

As identified in section 2.2, the XE1 hardware design provisions for disabling local loopback in hardware (See Figure 2). The default configuration permanently enables the receiver however with a population option the receiver can be disabled while transmitting data. Contact your sales representative if you would like to discuss this option.

With local loopback disabled the transmit and receive buffers of the UART would read as follows.

UART Transmit buffer:

Byte	0	1	2	3	4	5	6	7
Data (Hex)	01	03	00	00	00	02	C4	0B
Description	Address	Command	Register Address		Register Count		Checksum	

UART Receive buffer:

Byte	0	1	2	3	4	5	6	7	8
Data (Hex)	01	03	04	00	06	00	05	DA	31
Description	Address	Command	Count	Register Data 0		Register Data 1		Checksum	

3.2.4.1 Automatic disable of local loopback within the Linux kernel (TM1 and TM3 only)

As documented in section 3.2.3.3 the TM1 and TM3 Linux kernels and Android were updated to support the Linux serial-rs485 API.

https://www.kernel.org/doc/Documentation/serial/serial-rs485.txt

As well as the API supporting automatic transmit control it also supports disabling local loopback. This is achieved by disabling the UART receive buffer during transmissions. The SER_RS485_RX_DURING_TX flag should be cleared in the serial_rs485 structure to disable local loopback.

With local loopback disabled the transmit and receive buffers of the UART would read as follows.

UART Transmit buffer:

Byte	0	1	2	3	4	5	6	7
Data (Hex)	01	03	00	00	00	02	C4	0B
Description	Address	Command	Register Address		Register Count		Checksum	

UART Receive buffer:

Byte	0	1	2	3	4	5	6	7	8
Data (Hex)	01	03	04	00	06	00	05	DA	31
Description	Address	Command	Count	Register Data 0		Register Data 1		Checksum	

Appendix A – Linux sample application demonstrating auto rs485 transmit control, and tcdrain based transmit control

```
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <asm/types.h>

//#define MANUAL_RS485 1

struct serial_rs485
{
        __u32 flags; /* RS485 feature flags */
#define SER_RS485_ENABLED (1 << 0) /* If enabled */
#define SER_RS485_RTS_ON_SEND (1 << 1) /* Logical level for RTS pin when sending */
#define SER_RS485_RTS_AFTER_SEND        (1 << 2)        /* Logical level for RTS pin after sent*/
#define SER_RS485_RX_DURING_TX (1 << 4)
        __u32           delay_rts_before_send;     /* Delay before send (milliseconds) */
        __u32           delay_rts_after_send;      /* Delay after send (milliseconds) */
        __u32 padding[5]; /* Memory is cheap, new structs are a royal PITA .. */
};

static struct termios cfg;
static struct serial_rs485 rs485conf;

void closeserial(int fd)
{
   tcsetattr(fd, TCSANOW, &cfg);
   if (close(fd) < 0)
      perror("closeserial()");
}

int enablers485(int fd)
{
        int ret;
        /* Enable RS485 mode: */
        rs485conf.flags |= SER_RS485_ENABLED;

#ifdef MANUAL_RS485
        //disable RS485 mode if we want to run in manual mode
        rs485conf.flags &= ~SER_RS485_ENABLED;
#endif

        /* Set logical level for RTS pin equal to 1 when sending: */
        rs485conf.flags |= SER_RS485_RTS_ON_SEND;

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

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

        /*Clear this flag if you do not want to receive data while sending data */
        rs485conf.flags &= ~(SER_RS485_RX_DURING_TX);

        ret = ioctl (fd, TIOCSRS485, &rs485conf);
        if (ret < 0) {
                /* Error handling. See errno. */
                printf("Failed to enable RS485 mode: %d\r\n", ret);
        }
        return ret;
}

int openserial(char *devicename)
{
   int fd;
   struct termios attr;

   if ((fd = open(devicename, O_RDWR)) == -1) {
      perror("openserial(): open()");
      return 0;
   }

   if (tcgetattr(fd, &cfg) == -1) {
      perror("openserial(): tcgetattr()");
```

```
        return 0;
    }


    cfmakeraw(&cfg);

    cfsetspeed(&cfg, B38400);

    cfg.c_cflag &= ~CS5;
    cfg.c_cflag &= ~CS6;
    cfg.c_cflag &= ~CS7;
    cfg.c_cflag &= ~CS8;
    cfg.c_cflag |= CS8;

    cfg.c_cflag &= ~CSTOPB;

    cfg.c_cflag &= ~PARENB; //clear parity enable
    cfg.c_cflag &= ~PARODD; //clear parity type bit

    attr = cfg;

    if (tcsetattr(fd, TCSANOW, &cfg) == -1) {
        perror("initserial(): tcsetattr()");
        return 0;
    }

    if (tcflush(fd, TCIOFLUSH) == -1) {
        perror("openserial(): tcflush()");
        return 0;
    }

    return fd;
}


int main()
{
    int ret;
    int uartfd;
    int txengpio;

    char *serialdev = "/dev/ttymxc2";
    char *txendev = "/sys/class/gpio/gpio67/value";

    uartfd = openserial(serialdev);
    if (!uartfd) {
        fprintf(stderr, "Error while initializing %s.\n", serialdev);
        return 1;
    }

    //open gpio value file for manual RS485 transmit control
    txengpio = open(txendev, O_WRONLY);
    if (!txendev) {
        fprintf(stderr, "Error while opening %s.\n", txendev);
        return 1;
    }

    //Set TxEn GPIO low
    write(txengpio, "0", 1);

    fcntl(uartfd, F_SETFL, O_NONBLOCK);

    ret = enablers485(uartfd);
    if(ret < 0)
    {
        return 1;
    }

    tcflush(uartfd, TCIFLUSH);   /* Discards old data in the rx buffer            */

    char write_buffer_modbus_rtu[8] = {0x01, 0x03, 0x00, 0x00, 0x00, 002, 0xC4, 0x0B};           /* Buffer containing characters to write into port */
    int  bytes_written  = 0;    /* Value for storing the number of bytes written to the port */

                #ifdef MANUAL_RS485
                write(txengpio, "1", 1); //manually enable transmit control
                #endif

                bytes_written = write(uartfd, write_buffer_modbus_rtu, sizeof(write_buffer_modbus_rtu));

                printf("\n  %d Bytes written to ttymxc2", bytes_written);

                #ifdef MANUAL_RS485
                tcdrain(uartfd); //wait for TX Complete
                write(txengpio, "0", 1); //manually disable transmit control
                #endif

                usleep(20000); //Turnaround time

                char read_buffer[128];   /* Buffer to store the data received              */
                int  bytes_read = 0;    /* Number of bytes read by the read() system call */
                int i = 0;

                bytes_read = read(uartfd, &read_buffer, 128); /* Read the data                  */

                printf("\n\n  Bytes Received: %d", bytes_read); /* Print the number of bytes read */
                printf("\n\n  ");



                for (i = 0; i < bytes_read; i++)                /*printing only the received characters*/
                {
```

```
                        printf("%.2x ", read_buffer[i]);
                }

                printf("\r\n");

    closeserial(uartfd);
    close(txengpio);
    return 0;
}
```

Appendix B – TM1, TM3 specific UART optimisations

The TM1 and TM3 UART driver in the Linux kernel is designed to be efficient at high throughputs and baud rates. One technology that the driver uses is DMA (direct memory access) to provide efficient transfer of data. A second technique that the driver uses is setting a high FIFO threshold to limit the amount of interrupts requiring software servicing. While these driver optimisations give good performance and efficiency at high throughputs, this is not always the case for low baud rates and small amounts of data which tends to be the case with protocols using RS-485.

Therefore we have modified the TM1 and TM3 Linux kernel UART driver to allow DMA to be disabled and enabled at runtime and also allow the FIFO threshold to be configured at run time.

For RS485 communication it is recommended that DMA is disabled, and the FIFO threshold is set to 1.

B.1 – TM1 UART DMA

To allow the DMA function to be disabled a file called dmaenabled has been added to the sysfs for UARTS.

To disable UART DMA for the RS485 UART on **TM1** the following command can be issued at a console or through application software.

echo 0 > /sys/class/tty/ttymxc2/device/dmaenabled

To enable UART DMA for the RS485 UART on **TM3** the following command can be issued at a console or through application software.

echo 1 > /sys/class/tty/ttyS3/device/dmaenabled

Note: DMA must be disabled before application software opens the UART.

B.2 – TM1 UART FIFO Threshold

To allow the UART FIFO threshold to be configured a file called rxfifothreshold has been added to the sysfs for UARTS.

To modify the UART FIFO threshold to 1 for the RS485 UART on **TM1** the following command can be issued at a console or through application software.

echo 1 > /sys/class/tty/ttymxc2/device/rxfifothreshold

The rxfifothreshold can be set to any value between 1 and 32 on **TM1**.

To modify the UART FIFO threshold to 1 for the RS485 UART on **TM3** the following command can be issued at a console or through application software.

echo 1 > /sys/class/tty/ttyS3/device/rxfifothreshold

The rxfifothreshold can be set to any value between 1 and 256 on **TM3**, however due to hardware limitations only 4 effective value ranges are applied. These are as follows:

- Value range 1 – 31: interrupt is raised when 1 character is in RX FIFO
- Value range 32 –95: interrupt is raised when 64 characters are in RX FIFO (¼ of the FIFO buffer size)
- Value range 96-191: interrupt is raised when 128 characters are in RX FIFO (1/2 of the FIFO buffer size)
- Value range 192-256: interrupt is raised when the RX FIFO is full.

Note: The FIFO threshold must be set before application software opens the UART.

Change Log

1.2

- added information related to TM3 module

1.1

- added DB1 specific information
- added Change Log

1.0

- initial document