Programming Peperoni's USB DMX512 Interfaces

Version 2.2
by Dr. Jan Menzel*

June 18, 2007

Abstract

This document describes how all USB DMX512 interfaces by Peperoni[1] and Lighting-Solutions[2] are programmed. It includes all information software engineer have to supply to the operating systems USB-stack to talk to the interfaces.

1 Identifying

Lighting-Solutions has the Vendor ID (*idVendor*) 0x0ce1 (dec. 3297). The product ID (*idProduct*) of the USB devices within the scope of this document are

Product	*idProduct*
USBDMX X-Switch	0x0001
Rodin1	0x0002
Rodin2	0x0003
RodinT	0x0008
USBDMX21	0x0004

bDeviceClass [1, p. 197] is 0xff (vendor-specific device class), *bDeviceSubClass* is 0x00 and *bDeviceProtocol* 0x01.

The manufacturer (*iManufacturer*) and product (*iProduct*) strings are set and can be read. Their values depend on the product. The serial number string *iSerialNumber* is not used.

2 Configuring

All interfaces have one configuration with *bConfigurationValue* [1, p. 202] set to 1. *bInterfaceClass* is set to 0xff (vendor-specific class), *bInferfaceSubClass* to 0x00 and *bInterfaceProtocol* = 0xff (vendor-specific protocol). To configure any device send a SetConfiguration(1) request. The power consumption for all devices is below 100 mA, making them low power devices which can be operated on bus powered hubs. Non of the devices supports the Remove Wakeup feature and String descriptors for the configuration (*iConfiguration*).

The USBDMX X-Switch is the only device, which has up to three configurations to adapt the needs to the available power. If unconfigured both transmitter and receiver are disabled. Using configuration 1, only the transmitter is active requiring current of 180 ms. With configuration 2 both transmitter and receiver are enabled requiring 220 mA, whereas configuration 3 only enables the receiver (65 mA). Configuration 3 is only available with firmware version 1.1 (see section 4 for details) or higher. In the USBDMX X-Switch each configuration has its own string descriptor (*iConfiguration*).

*menzel@peperoni-light.de

[1] Peperoni, Dr. Jan Menzel & Dirk Bertelmann, Stiefmuetterchenweg 26, 22607 Hamburg, Germany, http://www.peperoni-light.de

[2] http://www.lighting-solutions.de

3 Programming

All interfaces support information exchange via control pipe. In addition, information exchange via bulk pipes has been added to later version.

3.1 Via Control Pipe

All information exchange with all interfaces can be done using control transfer to EP0 (in and out) and vendor-specific requests [1]. As specified, the direction of data flow is determined from bit 7 in *bmRequestType* [1, p. 183]. To denote a vendor-specific request bits 5 and 6 have to be set to 0x2 [1, p. 183].

The requested function is determined from evaluating the *bRequest* [1, p. 183] field. The following 9 requests are implemented:

Name	bRequest	Description
ID_LED	0x02	Read/write led usage
DMX_TX_MEM	0x04	Read/write transmitter data memory
DMX_TX_SLOTS	0x05	Read/write transmitter slot counter
DMX_TX_STARTCODE	0x06	Read/write transmitter startcode
DMX_TX_FRAMES	0x07	Read transmitter frame counter
DMX_RX_MEM	0x08	Read/write receiver data memory
DMX_RX_SLOTS	0x09	Read/write receiver slots counter
DMX_RX_STARTCODE	0x0A	Read/write receiver startcode
DMX_RX_FRAMES	0x0B	Read receiver frame counter

3.1.1 ID_LED

Get or set the led usage of Rodin interfaces.

All Rodin interfaces have a dual color blue/red led. Using this request one can change the usage of this led. To set the led usage send a ID_LED request with *wValue* representing the new value and *wLength* set to 0 (meaning no data stage). To read the led usage a ID_LED and expect the result in 1 byte back.

The default is 0xff which signals activities on the USB. If the value is set to 0xfe the red led will blink of not dmx signal is received. For all other values the led will blink the corresponding number using a long blink for 10th and short for 1th.

3.1.2 DMX_TX_MEM, DMX_RX_MEM

Access transmitter (DMX_TX_MEM) or receiver (DMX_RX_MEM) memory.

The interface sends back or expects *wLength* bytes[3]. Data is read from or written to memory with offset *wIndex*. *wIndex* = 0 is the first slot of the DMX512 frame, but not the startcode. Boundaries are not checked by any interface. Writing slots above 512 ought to be avoided. Reading slots larger then 512 will return in unknown data.

wValue has to have the value 0x0000 by default. If *wValue* is set to 0x0001, reading or writing is blocked until the current frame has been transmitted or received completely.

Blocking of this read/write feature is only available with firmware version 1.1 (see section 4 for details) or higher.

[3] one byte is treated as one DMX512 slot

Recommendations

- Due to the USB protocol overhead transferring one large block should be favoured over many small ones.
- Blocking should be used when updating the transmitter respectively reading the receiver solely.
- To update the transmitter memory and read the receiver memory in parallel, blocking should be used for writing the transmitter memory while the receiver memory should be read non-blocking. Only for applications mostly retransmitting received data and relying on low latency this schema should be inverted. Blocking should be used for reading the receiver memory while the transmitter memory is written non-blocking. This allows to exactly follow the received data while the other schema has the advantage of fastest data transmission.

3.1.3 DMX_TX_SLOTS, DMX_RX_SLOTS

Get or set the number of slots to be transmit (DMX_TX_SLOTS) or read the number of slots received in the last DMX512 frame (DMX_RX_SLOTS).

To set the number of slots to be transmitted send a DMX_TX_SLOTS request with *wValue* representing the new value and *wLength* set to 0 (meaning no data stage). Setting the number of slots of the received will result in an error.

To read the number of slots transmitted or last received send a DMX_TX_SLOTS or DMX_RX_SLOTS request and expect the result in 2 bytes back. The lower byte is transmitted first.

The default is to transmit 512 slots per frame.

3.1.4 DMX_TX_STARTCODE, DMX_RX_STARTCODE

Get or set the transmitter's or receiver's start-code.

To set the transmitter or receiver start-code send a DMX_TX_STARTCODE or DMX_RX_STARTCODE request with the new value in *wValue* and *wLength* set to 0. Note, that the receiver will only read frames with the start-code set.

To read the current start-code send a DMX_TX_STARTCODE or DMX_RX_STARTCODE request and expect the result as 1 byte back.

The default start-code is 0x00 for transmitter and receiver.

3.1.5 DMX_TX_FRAMES, DMX_RX_FRAMES

Get the number of frames transmitted or received.

Transmitter and receiver have individual 32 bit counters for counting the number of frames transmitted or received. This counters are cleared on power up and incremented on any successful transmitted or received DMX512 frames. Note, that receiver only count frames with their start-code equal to the one set (see section 3.1.4).

The frame counter can only be read by sending a DMX_TX_FRAMES or DMX_RX_FRAMES request and expecting the result as 4 bytes back. The lowest byte is again transmitted first.

Trying to set the frame counter will result in an error.

These counters are designed to check e.g. whether the receiver is active and/or if a new DMX512 frame has been successfully received. Since the counters are 32 bits in size the numbers will be almost unique, but software engineers should be aware of overflows.

3.2 Via Bulk Pipe - old version

Starting from firmware version 4.0 (bcdDevice $\geq$ 0x0400) a new and fast protocol for data exchange via a bidirectional bulk pipe was added. Starting with this firmware two bulk endpoints with *bEndpointAddress* of 0x02 (direction out) and 0x82 (direction in) are available.

Data exchange is always done by sending a command structure to the device followed by a data transmission either from host to device or from device to host. This transmission only reads or writes the content of transmitter or receiver memory. It does change anything else. Changing the startcode or reading frame counters still has to be done via control transactions. Also reading or writing transmitter or receiver memory blocking is not supported by this protocol.

The command structure is defined as follows.

Offset	Field	Size	Description
0	protocol	1	Protocol identifier, has to be 1
1	request	1	request type
2	slots	2	size of the data stage, max. value: 512

The total size of the command structure is 4 bytes.

Slots has to be transmitted as little endian with LSB first.

The requests supported are

Request	Value	Description
TX_SET	0x00	Write the transmitter memory
TX_GET	0x01	Read the transmitter memory
RX_SET	0x02	Write the receiver memory
RX_GET	0x03	Read the receiver memory
TX2_SET	0x04	Write the second universes transmitter memory
TX2_GET	0x05	Read the second universes transmitter memory

Using this protocol one sends a command structure with the intended request followed by sending or reading *slots* bytes of data. In case of any <X>_SET request the command structure and data should be placed back-to-back in one USB transaction.

3.3 Via Bulk Pipe - new version

Starting from firmware version 5.0 (bcdDevice $\geq$ 0x0500) a highly sophisticated protocol, optimized for RDM, via a bidirectional bulk pipe was added. This new version exchanges all relevant parameters in the device at once. Hence it allows to send and receive frames with individual parameters. It even allows to switch between transmission and reception on a frame by frame base.

Compared with the old version, this new version uses a three stage strategy for data exchange: first a command structure is send from host to device. Then data with a previously negotiated amount and direction is exchanged. Finally a status structure is send from device to host.

3.3.1 The Command Structure

The command structures is composed of a general structure follows by an individual structures depending on the request.

Offset	Field	Size	Description
0	version	4	Protocol Version: 0x326b4d02
4	request	1	request type
5	universe	1	universe number
6	length	2	length of the data stage

All multi byte values are little endian and have to be send LSB first.
request can have the following values:

Request	Value	Description
REQUEST_TYPE_SET	0x00	Transmit a DMX512 frame
REQUEST_TYPE_GET	0x10	Receive a DMX512 frame

The *length*-field specifies the total length of the data stage. The maximum value is 519.

For data transmission (*request* = *REQUEST_TYPE_SET*) the command structure is completed by the following structure:

Offset	Field	Size	Description
8	config	1	configuration for the transmission
9	time	2	time in units of ms, meaning depends on *config*
11	time_break	1	length of Break, see text
12	time_mab	1	length of Mark-after-Break, see text

config controls the transmission and can have a or'ed combination of the following values:

Configuration	Value	Description
CONFIG_DELAY	0x01	Delay this frame with respect to the previous
CONFIG_BLOCK	0x02	block USB transaction until transmission starts
CONFIG_SWITCH_RX	0x04	switch to receive mode immediately after this frame
CONFIG_DONT_RETRANSMIT	0x08	do not retransmit this frame

If CONFIG_DELAY is set the start of this frame will be delayed by the given value of *time* (units ms) with respect to the previous frame. The actual value used as reference from previous frame is the one returned as *timestamp* in the status stage. Using this feature one can precisely send e.g. 20 frames per second if the time difference between the last frame transmitted and the exact time the next frame has to be transmitted is calculated and set as *time* value. Usually this value will be just the time between two frames.

Setting CONFIG_BLOCK will block the current USB transaction until either *time* has elapsed or the DMX universe transitions back to idle state. The former will be reported as *STATUS_TIMEOUT*, whereas the later means that the previous transmission or reception has finished.

The values of *time_break* and *time_mab* are in internal units to be calculated using

$$time(t) = 256 - (t - t_{offset})/t_{units} \quad (1)$$

with the parameters

Variable	t_{units}	t_{offset}	Default
time_break	$2.67\,\mu s$	$1\,\mu s$	181
time_mab	$2.67\,\mu s$	$5\,\mu s$	250

Both, *time_break* and *time_mab*, can have any value between 0 and 255. If set to 255 no break and/or no Mark-after-Break is generated.

For data reception (*request* = *REQUEST_TYPE_GET*) the command structure is completed by the following structure:

Offset	Field	Size	Description
8	slots	2	number of slots to receive, incl. startcode
10	timeout	2	timeout to wait for the frame to be receive, units ms
12	timeout_rx	1	timeout for premature end of frame, see text

timeout defines the time to wait longest for a complete frame (*slots* slots, incl. start-code) to be received. If that expires *STATUS_TIMEOUT* will be reported.

timeout_rx defines a time to wait longest for the next slot within the current frame to be receive. It can be used to quickly detect a frame shorter as expected (less than *slots* slots, incl. start-code). If this time expires a premature end of frame error (*STATUS_RX_TIMEOUT*) will be reported. The value to *timeout_rx* has to be calculated using equation 1 with $t_{offset} = 0$ and $t_{units} = 42.67\,\mu$s giving timeout of up to 10.9 ms at a value of 0. Setting *timeout_rx* to 0xff will disable this feature. In this case caution should be taken, since transmission can get impossible. [4]

Taking the above said together a frame ends normally if *slots* slots (including start-code) have been received or if the time between two consecutive slots was larger then *timeout_rx*.

Transceiver supporting this protocol will receive frames without startcode and report them as such.

3.3.2 The Data Structure

The data phases uses a rather simple structure:

Offset	Field	Size	Description
0	version	4	Protocol Version: 0x326b4d02
4	slots	2	number of slots to transmit/received, incl. startcode
6	data	513	data to transmit or received incl. startcode

The length of the data structure is always given by the *length*-field of the command structure. The frame to be transmitted or received can be shorter, meaning that *slots* does not have and will not necessarily be *length* - 6. This should be kept in mind when receiving data.

3.3.3 The Status Structure

The structure used as status stage is given as follows:

Offset	Field	Size	Description
0	version	4	Protocol Version: 0x326b4d02
4	timestamp	2	timestamp of the frame transmitted or received, units ms
6	status	1	status of the last DMX frame
7	spare	1	for future use, to be ignored

timestamp is the actual value of a 16bit millisecond counter taken at the time the first slot was transmitted or received. It can be used to precisely calculate the speed of reception or to precisely record and/or retransmit DMX512 frames.

status can have the following values:s

[4] To understand that, one has to take into account that only on an idle line a transmission can be started safely. That means that any previous reception has to have definitely ended. And detecting that can only be done by comparing expected and current number of slots received and using this timeout.

Status	Value	Description
STATUS_OK	0x00	no errors
STATUS_TIMEOUT	0x01	request timed out
STATUS_TX_START_FAILED	0x02	delayed start of transmission failed
STATUS_UNIVERSE_WRONG	0x03	wrong universe addressed
STATUS_RX_OLD_FRAME	0x10	old frame not read
STATUS_RX_TIMEOUT	0x20	reception finished with timeout
STATUS_RX_NO_BREAK	0x40	frame without break received
STATUS_RX_FRAMEERROR	0x80	reception finished with frame error

All STATUS_RX_<X> values can be or'ed together.

STATUS_RX_TIMEOUT means that the reception finished because no slots was received within the expected time given by the value of *timeout_rx*.

STATUS_RX_FRAMEERROR denotes that the frames finished with the last slot being not correctly received: the level of one of the two stop bits was not HIGH.

4 Changes

4.1 Firmware

v1.0 (*bcdDevice* = 0x0100) initial version

v1.1 (*bcdDevice* = 0x0101) configuration 3 (read only), blocking read/write and the serial number were added.

v4.0 (*bcdDevice* = 0x0400) data exchange on bulk pipe using old protocol added.

v5.0 (*bcdDevice* = 0x0500) data exchange on bulk pipe using new protocol added.

4.2 Programming Specifications

v1.0 initial release

v1.1 changes related to firmware v1.1 added

v2.0 added documentation for bulk pipe data exchange

v2.1 section 3.3: command structure and status codes updated

v2.2 ID_LED added

References

[1] Universal Serial Bus Specification, Revision 1.1, 23.09.1998, http://www.usb.org