Embedding it better...

µTasker Document

µTasker – DMX512 and RDM

Table of Contents

1. Introduction

This document discusses the DMX512 (Digital Multiplex) implementation in the µTasker project.

DMX512 is used typically for controlling stage lighting and effects but also finds use in various non-theatrical/lighting environments. It usually makes use of RS485 electrical signalling at the electrical level together with asynchronous UART bit level protocol at 250kBaud, one start bit, 8 bits, no parity and 2 stop bits (11 bits per character).

DMX512 is unidirectional from a master to a slave but can have a feedback path if its RDM (Remote Device Management) option is used.

A DMX512 frame consists of a break condition signifying its start, a MAB (Mark-After-Break) and a slot 0 data byte followed by 512 channel data bytes. However, shorter frames than 512 are allowed (as log as not shorter than 1.204ms in duration) and the frame rate can be as low as once every 1s. Timing is fairly flexible as long as the minimum break times of 92us and minimum MAB of 12us are respected, which give a maximum refresh rate of about 44Hz (22.676ms frame time).

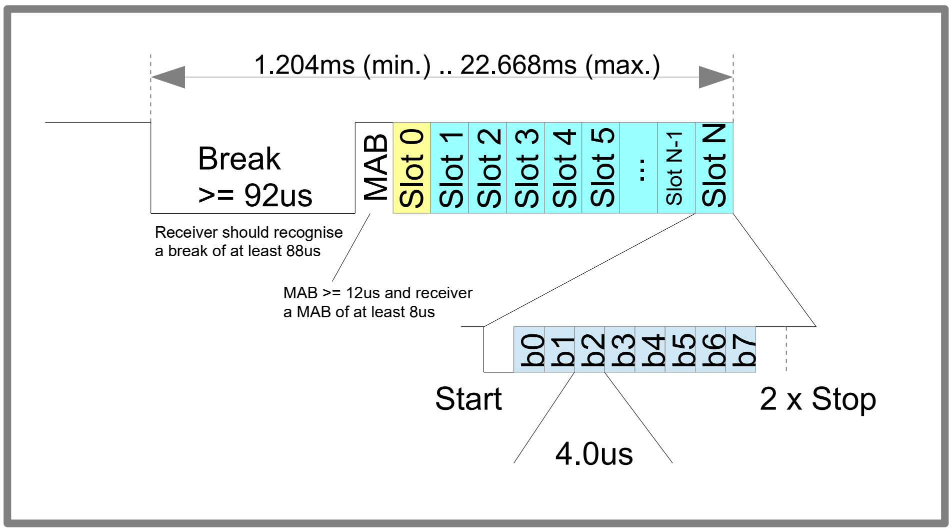
The µTasker DMX512 transmitter allows configurable lengths and timing with accurate frame timing, together with low overhead; DMA is used where possible so that the application only needs to supply the next transmit frame's content on time.

The µTasker DMX512 receiver is designed to be tolerant to frame timing deviations and also has low overhead with DMA being used wherever possible. The application is informed of new reception frame availability in its input buffer so that the content can be handled without needing high priority operations to be in place.

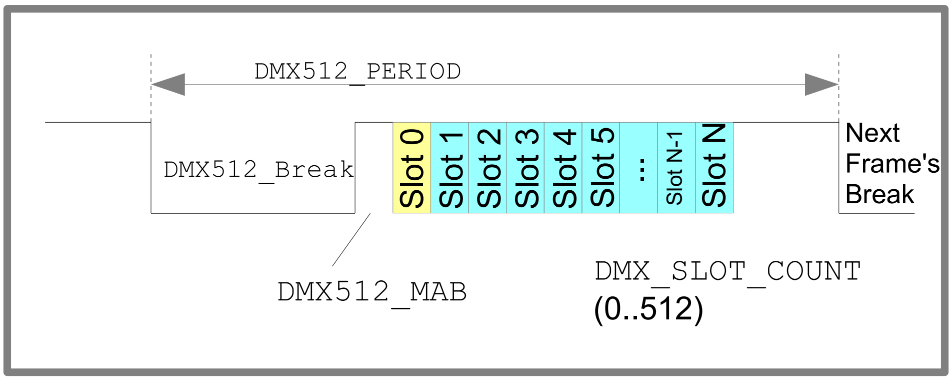
Optional RDM transmission and reception is supported which takes over as much work from the application as possible.

2. DMX512 Frame

The following figure shows the basic DMX512 frame and its timing values and ranges where Slot N is max. Slot 512:

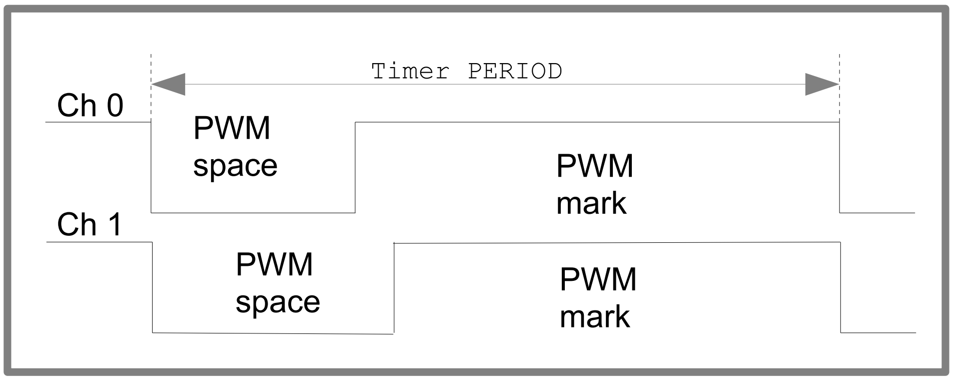

1.204ms (min.) .. 22.668ms (max.)
Break
>= 92us
MAB
Slot 0
Slot 1
Slot 2
Slot 3
Slot 4
Slot 5
...
Slot N-1
Slot N
Receiver should recognise a break of at least 88us
MAB >= 12us and receiver a MAB of at least 8us
b0
b1
b2
b3
b4
b5
b6
b7
Start
2 x Stop
4.0us

From this general frame the configuration settings for the transmitted frame in the µTasker project configuration can be matched to:

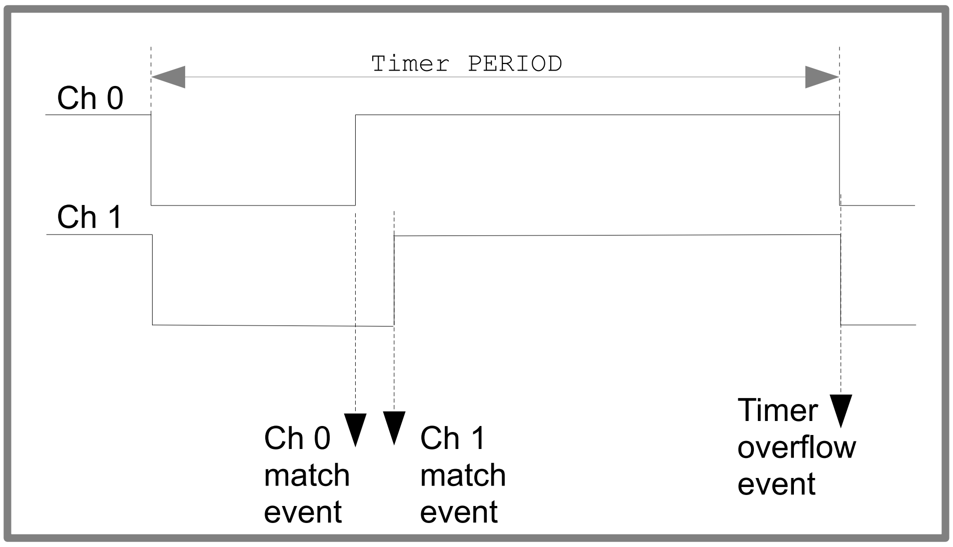


3. DMX512 Transmission

The FlexTimer (*or Timer PWM Module [TPM] in some Kinetis parts*) is an excellent time base for DMX512 frame timing and so is used as generic method on all Kinetis parts. As the following figure shows, two channels of a timer (which share the same time base and are synchronised) can be used to derive the MAB start and MAB stop points in time. Their cycle time (frequency) defines the DMX512 frame period.



The channel 0 and channel 1 signals are generated by setting them up to generate PWM outputs starting with a space and changing to a mark at the appropriate times. Channel 0 represents thus the break condition that needs to be generated on the transmitter output and channel 1 output going high represents the point in time when the DMX512 data transmission starts. These outputs are useful for checking the timing involved but don't actually need to be used due to the fact that their interrupts (or DMA triggers) are used to control the process – *the related timer and channel events are shown in the following image*:



It is clear to see from these events (interrupt or DMA triggers) that the control of the frame output is very straight forward indeed:
1. On each *timer overflow event* a break condition ('0' on transmitter output) is started at the transmitter
2. On each *channel 0 match event* the break condition is terminated so that the transmitter output becomes '1' again
3. On each *channel 1 match event* frame data transmission is started

Further to this, each timer overflow event (1) can also be used to inform the application layer that the frame will start. In the µTasker implementation one or more frame buffers can be prepared for transmission before the frame begins and so the application can use this to prepare the subsequent frame data so that it is ready when its transmit time slot arrives (2 in a following frame transmission).

It is worth pointing out that some of the Kinetis devices (such as KL27 or K66) allows outputs from certain flex timer (or TPM) channels to modulate certain UART transmit outputs. This is intended for infra-red transmission, for example, but can also be used to directly modulate the break condition (see channel 0 output) on to the transmitter. This is however not used as "generic" method since it is restricted to certain parts with this capability and is only available on certain UART/timer combinations.

The following extracts from the DMX512 code explain important details of the implementation.

3.1 UART initialisation

```
unsigned char   ucDMX512_tx_buffer[DMX_SLOT_COUNT + 1 + 2];
tInterfaceParameters.ucSpeed = SERIAL_BAUD_250K;                    // fixed DMX512 baud rate
tInterfaceParameters.Config = (CHAR_8 | NO_PARITY | TWO_STOPS | CHAR_MODE |
UART_HW_TRIGGERED_TX_MODE);                                         // fixed DMX512 settings
tInterfaceParameters.Rx_tx_sizes.TxQueueSize = (sizeof(ucDMX512_tx_buffer) * DMX512_TX_BUFFER_COUNT);
                                                                    // output buffer size
tInterfaceParameters.ucDMAConfig = (UART_TX_DMA);                   // transmit using DMA
```

In the UART configuration these parameters set the DMX512 mode of operation (8 bits, two stop bits at 250kBaud) whereby the transmitter is operated in DMA mode whenever possible.

The UART driver mode `UART_HW_TRIGGERED_TX_MODE` is specified which allows queuing frames in the output buffer for later release under hardware control.

The output buffer size is a multiple of the frame data length so that up to `DMX512_TX_BUFFER_COUNT` frames can be queued.

Note that when `DMX_SLOT_COUNT` is 512 the buffer size is a multiple of 515 bytes. This is due to slot 0 in addition to the 512 data bytes and also two bytes in the queue in this mode that specify each queued frame's individual length.

See the UART user's guide for full details of using the driver: http://www.utasker.com/docs/uTasker/uTaskerUART.PDF

3.2 Preparing and queueing a DMX512 frame

```
int I = 3;
fnDriver(DMX512_PortID[0], (TX_OFF), MODIFY_TX);
    // disable the transmitter since we will be preparing output and then starting by hardware trigger
ucDMX512_tx_buffer[0] = (unsigned char)((DMX_SLOT_COUNT + 1) >> 8);
                                                                  // individual message content length
ucDMX512_tx_buffer[1] = (unsigned char)(DMX_SLOT_COUNT + 1);
ucDMX512_tx_buffer[2] = 0;                                                            // slot 0 value
while (i < sizeof(ucDMX512_tx_buffer)) {
   ucDMX512_tx_buffer[i] = (unsigned char)(i - 3);                       // fill frame with data values
   i++;
}
fnWrite(DMX512_PortID[0], ucDMX512_tx_buffer, (DMX_SLOT_COUNT + 1 + 2)); // prepare first DMX512 frame
```

The transmitter is initially disabled since it will later be enabled to release frames and a frame is prepared in a temporary buffer. We note that in this mode the first two bytes of the content represents the length of the individual frame that we will be queuing in the output buffer. The frame is written to the UART output where it remains pending due to the fact that the transmitter is not yet active.

3.3 Configuring and starting timer control

```
PWM_INTERRUPT_SETUP pwm_setup;
pwm_setup.int_type = PWM_INTERRUPT;
pwm_setup.pwm_mode = (PWM_SYS_CLK | PWM_PRESCALER_16 | PWM_EDGE_ALIGNED | PWM_POLARITY |
PWM_NO_OUTPUT);                         // clock PWM timer from the system clock with /16 pre-scaler
pwm_setup.int_handler = 0;
pwm_setup.pwm_reference = (_TPM_TIMER_1 | 0);                               // timer module 1, channel 0
pwm_setup.pwm_frequency = (unsigned short)PWM_TPM_CLOCK_US_DELAY(DMX512_PERIOD, 16);
                                                       // generate frame rate frequency on PWM output
pwm_setup.pwm_value = ((pwm_setup.pwm_frequency * DMX512_BREAK)/DMX512_PERIOD);
                          // output starts low (inverted polarity) and goes high after the break time
pwm_setup.int_priority = PRIORITY_HW_TIMER;                 // interrupt priority of cycle interrupt
pwm_setup.pwm_mode |= PWM_CHANNEL_INTERRUPT;                // use channel interrupt to stop the break
pwm_setup.channel_int_handler = _mab_start_interrupt_0;     // interrupt call-back on channel match
START_DMX512_BREAK_0();                                     // generate the first break immediately
fnConfigureInterrupt((void *)&pwm_setup);                   // configure and start

pwm_setup.pwm_reference = (_TPM_TIMER_1 | 1);               // timer module 1, channel 1
pwm_setup.pwm_value = ((pwm_setup.pwm_frequency * (DMX512_BREAK + DMX512_MAB - 44))/DMX512_PERIOD);
            // second channel goes high after the MAB period to control the timing for the start of the
               frame transmission
pwm_setup.int_handler = _frame_interrupt_0;                 // interrupt call-back on PWM cycle
pwm_setup.channel_int_handler = _mab_stop_interrupt_0;      // interrupt call-back on channel match
fnConfigureInterrupt((void *)&pwm_setup);                   // configure and start
```

TPM1 channels 0 and 1 are configured (identical setup can be used for flex timers too) where the frame period and MAB times are expressed in us. Interrupts are entered on the timer overflow and on each channel match.

Note that there is a reduction in the time for channel 1's delay of 44us. This is used to compensate a 44us delay between enabling the Kinetis UART and it sending the first character, which is a characteristic of the UART in these devices; it sends 11 x '1' before shifting out the first data byte. This compensation may not be required for other processor types.

If the PWM outs are not required `PWM_NO_OUTPUT` is used. Without this, the signal appears at its PWM pin.

In the case of the Kinetis flex timers `PWM_POLARITY` is used to control a space followed by a mark (inverting the normal output polarity). If only interrupt (or DMA) generation is relevant this setting has no consequence.

See the hardware timer guide for full details of using the driver: http://www.utasker.com/docs/uTasker/uTaskerHWTimers.PDF

3.4 Interrupt call-backs

```
static void _mab_start_interrupt_0(void)
{
    END_DMX512_BREAK_0();        // remove the break condition (this is the start of the MAB period)
}
```

```
static void _mab_stop_interrupt_0(void)
{
    START_DMX512_TX_0();        // initiate frame transmission (this is the end of the MAB period)
}
```

```
// DMX512 transmit frame period interrupt
//
static void _frame_interrupt_0(void)
{
    START_DMX512_BREAK_0();
    fnInterruptMessage(OWN_TASK, (unsigned char)(DMX512_TX_NEXT_TRANSMISSION_0));
                             // inform the application that a DMX512 frame is just starting and it
                                should prepare the following frame content
}
```

The interrupts are used to control the hardware to generate break/idle signals and to start the frame transmission. The control is implemented as a macro that matches the hardware used.

In the case of the Kinetis UART the break control is performed by setting the Tx output temporarily to GPIO output function driving a '0' before reprogramming the pin back to its UART function. The reason for using this method rather than queuing break characters is that it allows more accuracy and is easier to control; the break character that the Kinetis UART sends is always 10 (or 11) bits in length (40 or 44us) and it is necessary to add additional break characters at appropriate times to extend this. Using GPIO the resolution is not restricted to a multiple of the break character length.

3.5 Application event handling

```
switch (ucInputMessage[MSG_INTERRUPT_EVENT]) {
case DMX512_TX_NEXT_TRANSMISSION_0:             // DMX512 frame is starting so we should prepare the
                                                   following frame's data content
    if (fnWrite(DMX512_PortID[ucInputMessage[MSG_INTERRUPT_EVENT] - DMX512_TX_NEXT_TRANSMISSION_0], 0,
(DMX_SLOT_COUNT + 1 + 2)) > 0) {                // if there is free buffer space
        ucDMX512_tx_buffer[2]++;
        fnWrite(DMX512_PortID[ucInputMessage[MSG_INTERRUPT_EVENT] - DMX512_TX_NEXT_TRANSMISSION_0],
ucDMX512_tx_buffer, (DMX_SLOT_COUNT + 1 + 2));  // prepare next DMX512 frame
    }
break;
}
```

The application handles the interrupt event that is posted by the frame cycle interrupt callback. It defines the next frame's content (here it is just incrementing the value sent in slot 0 for test purposes) and writes it into the UART output buffer where it remains until transmission is triggered again.

Note that in the `UART_HW_TRIGGERED_TX_MODE` the transmitter is automatically disabled after each frame transmission so that additionally queued frames will remain pending until released as required. The application can thus queue as many frames as the UART output buffer can hold. *The output buffer size must be an exact multiple of the frame size in this mode*.

The fast that the application can prepare multiple frame sin advance makes it very insensitive to its reaction time and ensure accurate timing in all circumstances.

3.6 Notes about DMA events

Interrupts have been used in the generic DMC512 transmission method since it makes it usable on all parts, even those without DMA or too limited DMA channels. By using the timer events to trigger DMA to perform the HW activity instead of the interrupts is otherwise possible but not shown further here; DMA driven control will remove any interrupt jitter in systems that require most exact timing. Generally, a frame cycle interrupt is however still needed in order to generate a suitable periodic event for the application.

3.7 Multiple DMA512 Outputs

The timer events can be used to perform the same actions on multiple UARTs if multiple DMX512 outputs are used. Each output just requires to open its own UART interface with the same settings and prepare its own frame content. The single timer can be used to control multiple synchronised transmitters.

If synchronisation is not desired (eg. a transmitter uses a different frame period and/or timing) a second timer can be configured in the same way to supply it's event timing.

4. DMX512 Reception

A DMX512 receiver needs to receive the data sent by a DMX512 master and associate it with a particular frame so that it can recognise slots 0 to slot 512 (for a full length frame). For lowest overhead, the receiver can operate in DMA mode – otherwise it operates in interrupt driven mode.

The DMX512 protocol uses a break to signify the start of a frame. This also signified the end of the previous frame, if there was one.

The minimum break duration that a receiver must be able to detect is 88us and it must be capable of receiving a data frame starting after a minimum 8us MAB.

A suitable strategy for reception (also suitable for DMA operation) is thus as follows:

1. Initially the receiver ignores or discards any data if started/connected during a period of frame data reception
2. A break is waited for and used to enable the subsequent reception
3. The receiver then collects data until a following break is detected. The data length is reported to the application so that it can read out the frame content. The receiver remains enabled so that it is ready to receive following data too, even if the application hasn't yet handled the first buffer content

The application extracts a complete frame (knowing its length) and processes the slot 0 frame type, followed by the data content. As long as the UART reception buffer is adequately large it allows multiple frames to be received before the application needs to process them, thus relaxing the application's reception timing requirements.

In a system with know, fixed length DMX512 frames the receiver could report the availability of the new frame once the approppriate amount fo data had been received. This would mean that the data could generally be processed earlier in the frame, rather than waiting for a break to be detected to signify that it has completed. The same strategy could be used to already close a reception frame after 513 bytes, knowing that this must be a compete frame. This is however not performed in the generic solution since it complicates DMA reception and is not always possible in processor DMA modes. The break detection is therefore always used to terminate a reception frame.

In the case of frame reception that is not terminated by a break after a maximum time of 1s its content is flushed and the receiver set to its inactive state, waiting for the start of new activity.

Frames that have more that 513 bytes in them are also flushed becasue they are longer than the allowed DMX512 frame length.

These reception error cases are reported to the application as error interrupt events for statistical purposes.

The basis for the UART DMA reception is a free-running DMA receiver with adequate buffer space for at least one DMX512 frame. If no DMA is available, or desired, the UART receiver can however also be used work in standard interrupt mode.

A break detection interrupt call-back is configured in order to initially enable the UART reception and handle the end of frame detection. It informs the application of the length of DMX512 frame data to be read from the UART input buffer.

The application responds to the report of the received frame being ready by reading its length from the UART's input buffer and processing the content accordingly. As soon as the application has read the data its space in the UART input buffer is again available for further reception.

DMX512 Reception in DMA mode

DMA mode is configured with the UART DMA mode:

```
tInterfaceParameters.ucDMAConfig = (UART_TX_DMA | UART_RX_DMA | UART_RX_DMA_BREAK | UART_RX_MODULO);
```

whereby `UART_RX_MODULO` is only required by devices whose DMA controller needs modulo aligned buffers for continuous operation (most Kinetis KL parts, for example).

DMX512 Reception in interrupt mode

Interrupt driven mode is configured with the UART configuratation

```
tInterfaceParameters.Config |= (MSG_BREAK_MODE);
```

which enabled interrupt framing mode based on break detection.

The UART driver reports frames identically in both cases, based on a message sent to the owner task with information about the channel that received a frame, along with its size. The task's input queue handles the message as follows:

```
        case TASK_TTY:                                          // message from the UART (pseudo) task
            fnRead(PortIDInternal, ucInputMessage, ucInputMessage[MSG_CONTENT_LENGTH]);
            switch (ucInputMessage[0]) {
            case TTY_BREAK_FRAME_RECEPTION:
                {
                    QUEUE_HANDLE Channel;
                    unsigned short usFrameLength;
                    Channel = ucInputMessage[1];   // uart channel that the frame has been received on
                    usFrameLength = ucInputMessage[2];
                    usFrameLength <<= 8;
                    usFrameLength |= ucInputMessage[3];    // the length of the frame waiting
                                                            in the UART buffer
                    fnHandleDMX512_frame(DMX512_PortID[0], usFrameLength);       // handle the frame
                    break;
                }
            }
            break;
```

The handling routine simply reads the data from the input queue and processes its content. The following shows it reading a frame (error handling has been removed for clarity) and deciding whether it is a standard DMA frame (slot 0 is 0x00) or needs special handling:

```
    unsigned char ucRxFrame[DMX_RX_MAX_SLOT_COUNT + 1];
    fnRead(uart_handle, ucRxFrame, usFrameLength);              // extract the DMX512 reception frame
    switch (ucRxFrame[0]) {                         // decide on what to do with it based on slot 0 value
    case START_CODE_DMX512:                                     // (0x00) DMX512 content
        break;
    case START_CODE_RDM:                                        // (0xcc) RDM content
        break;
    default:
        break;
    }
```

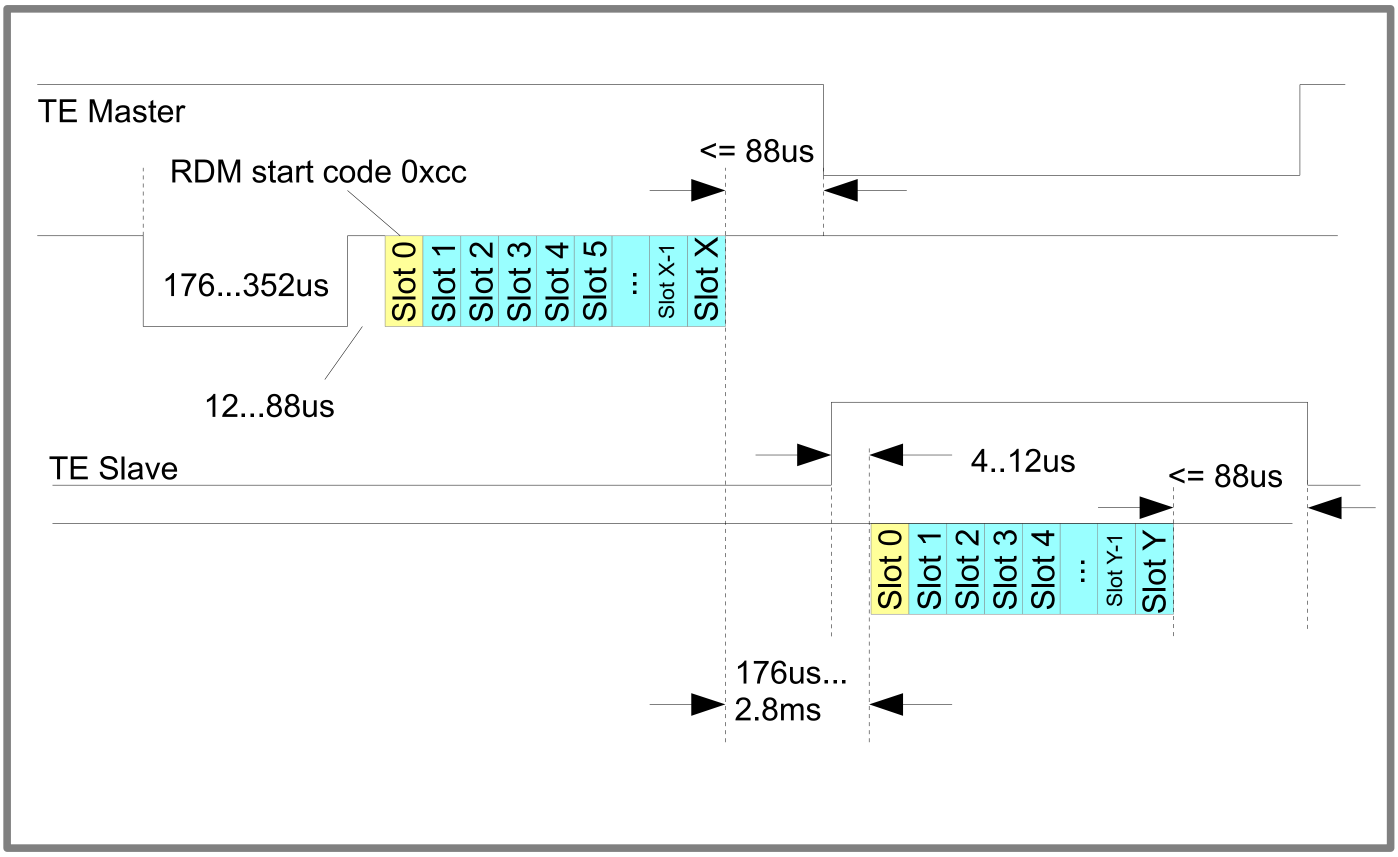
5. RDM (Remote Device Management)

RDM over DMX512 networks is specified in ANSI E1.20 – 2010.

It permits a master to discover, then configure, monitor and manage slave devices on the network.

A master can thus communicate with slaves when the network starts and is being configured and then intermittently during normal DMX512 operation (with some but minimum impact on the overall operation).

Since a slave often responds to a RDM transmission there is a difference to the basic DMX512 operation which is represented in the following diagram:



The first thing to notice is that the communication is bi-directional. Since it is semi-duplex on an RS485 physical layer the master requires an output to control it driving the line during normal operation and listening when it is expecting a slave response (TE Master signal). The same is true at the slave since it should only drive the line when responding to a request from the master.

During the data exchange the master's normal DMX512 cycle framing stops since it waits for a response from messages for up up 3ms (until the start) before declaring a lost response.

The master's break and MAB timing range in RDM mode is not the same as the DMX512 range but the DMX512 operation can be set to be within a range matching both specifications.

The slave responds without sending first a break when it responds to discovery message (DISC_UNIQUE_BRANCH), as shown the previous illustration. It also needs to respond to requests before the master sends a following break due to the fact that it MUST start a response within 2.8ms of the end of the master's transmission.

All other slave responses *include a break/MAB phase* before the data content as known from the master operation.

The packet format of an RDM frame is the same in both master-> slave and slave->master directions. It starts with the start code 0xcc followed by a sub-start code and a message length due to the fact that the messages and responses are not of a fixed length but can vary each time. It becomes evident from this fact and the response limitations that the slave must analyse the frame as it arrives and can't use DMA reception to handle only a complete frame; it must therefore always work in interrupt mode, check the start code and message length and then handle the frame once the advertised number of bytes have arrived.

The packet format continues with a 48 bit destination UID, a 48 bit source UID etc. as shown below:

Byte 0: START CODE
Byte 1: SUB-START CODE
Byte 2: MESSAGE LENGTH (including start code and content without check-sum – 24..255)
Bytes 3..8: DESTINATION UID (48-bits)
Bytes 9..14: SOURCE UID (48-bits)
Byte 15: TRANSACTION NUMBER (TN)
Byte 16: Port ID / RESPONSE TYPE
Byte 17: MESSAGE COUNT
Bytes 18..19: SUB-DEVICE (16 bits)
Bytes 20..X-2: MESSAGE DATA BLOCK (MDB) of variable size
Bytes X-1 and X: CHECKSUM (16 bits – sum of all previous bytes in the packet)

Multi-byte data is transmitted in big-endian (network) order.

6. RDM Discovery

RDM supports slave discovery which is usually the first thing that takes place in order to find out how many slaves are attached and to configure the complete system in order to avoid the need for manual configuration. It is based on the use of the DISC_UNIQUE_BRANCH message which allows all slaves that are within the requested branch to respond. Multiple slave responses will cause collisions (desired in this case) that allow the master to recognise that 'some' are there and then repeat the requests with ever decreasing scope in order to pin point where slaves are located (their addresses).

Before starting the discovery the master however sends a broadcast DISC_UN_MUTE message which allows the slaves to respond to such requests.

After identifying individual slaves the DISC_MUTE command, send to their uni-cast address, stops them from answering while further discovery is attempted.

The implementation of master RDM operation is based on a state event machine that temporarily stops the normal DMX512 transmission in preference of RDM transmission. Since there is often a response expected from slaves that support RDM the DMX512 framing is disabled (although the same timing is retained and used as a base for the RDM transmissions themselves).

Following the transmission of a complete RDM frame (from the second stop bit of the final character) the transmission line is no longer driven, the receiver is enabled, and a timer is started that corresponds to the maximum time that a slave response would be expected in (6ms) and the UART receiver buffer is then checked for reception. An reception data received (even corrupted) counts as a RDM receiver response, whereby no data reception is counted as no RDM response; this is used during discovery when multiple slaves may respond and result in corrupted data.

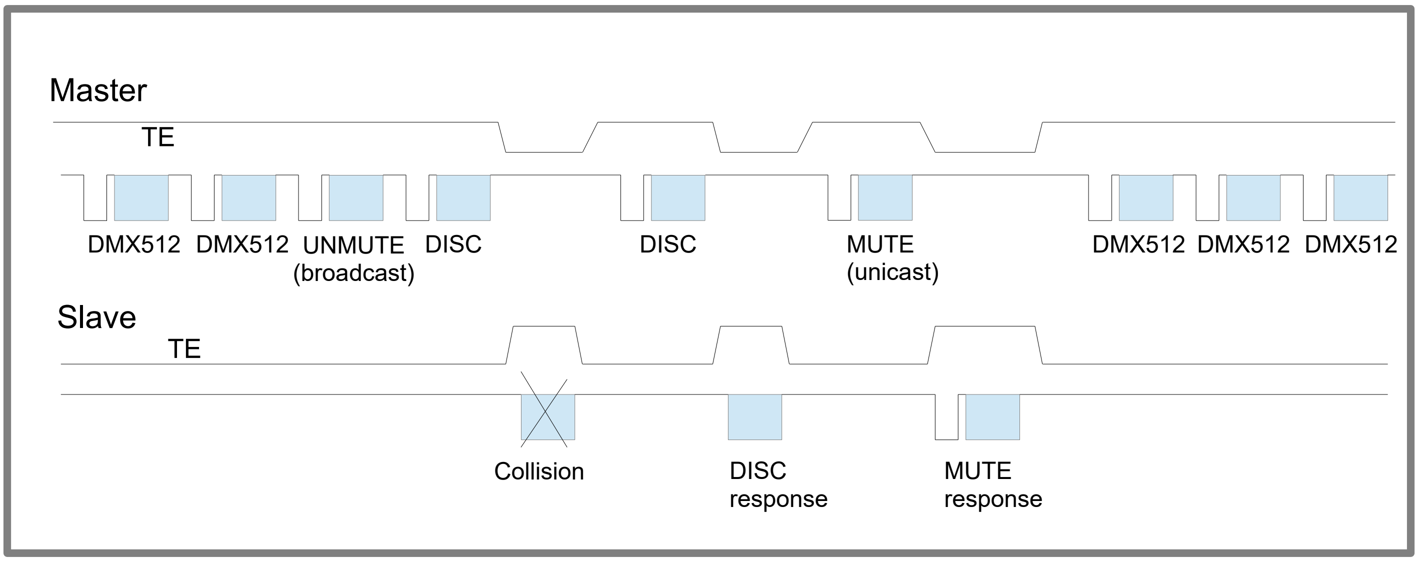
After handing potential responses the receiver is disabled, the transmitter driven again and either normal DMX512 or following RDM transmissions continue when the next frame allows it.

7. RDM Reception

RDM reception imposes additional constrains on a slave due to the fact that the slave needs to recognise the frame type and its length at the start of a reception and respond to RDM data within a strict time window (176us to 2.0ms from the end of the master's RDM transmission to returning a response – which is either the start bit of a data character or the start of a break as received by the master). This means that the break following the data reception will not be available to terminate the frame. Due to this reasons, slaves that need to support RDM and not operated in DMA reception mode but instead in receiver interrupt mode, handling each received data byte immediately in order to be able to achieve accurate response timing.

The following diagram shows a typical discovery sequence whereby the DMX512 framing is temporarily interrupted in order to send first a broadcast discovery un-mute command and then a number of discovery unique branch messages. The first response from the slave shows a collision resulting (corruption) due to more than one slave responding but the second (where the master has changed the discovery range that excludes the colliding slave) is responded to successfully. The master subsequently mutes the responding slave so that it can continue discovering further devices.

After the mute command has been sent the normal DMX512 framing continues, although normally the master will probably perform further discovery sequences to find and mute all slaves that exist.



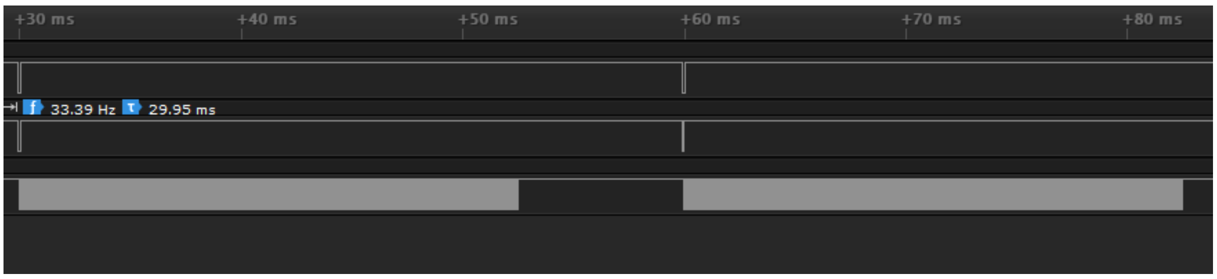
8. Conclusion

Work in progress.

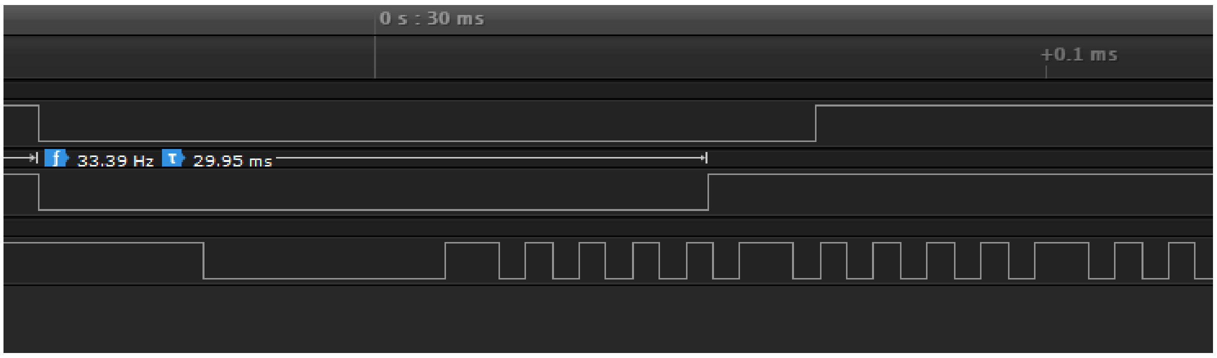
Modifications:

V1.00 3.05.2018: In progress

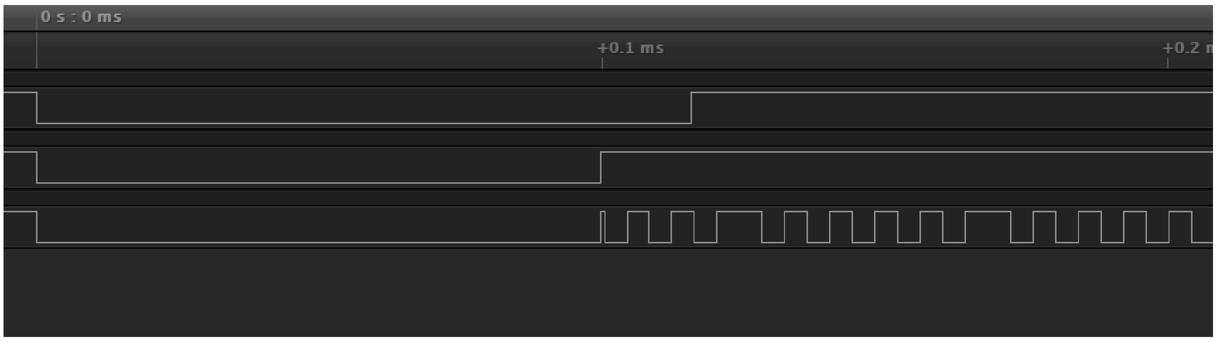
TEMPORARY recordings.



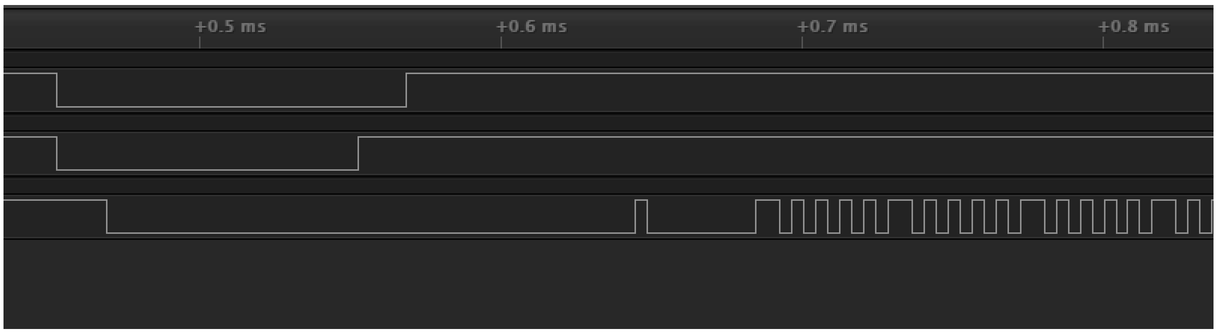
Frame, using PWM outputs to mark timing



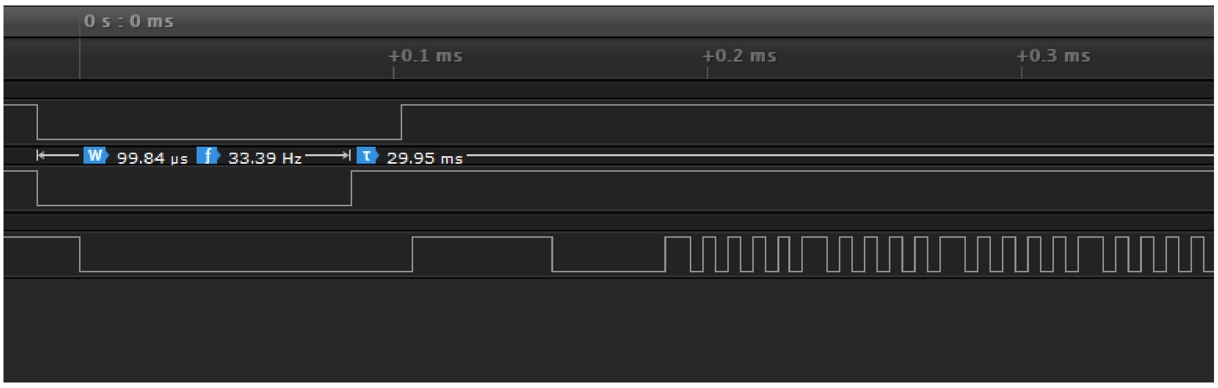
Close up of start (transmit starts too early in this test)



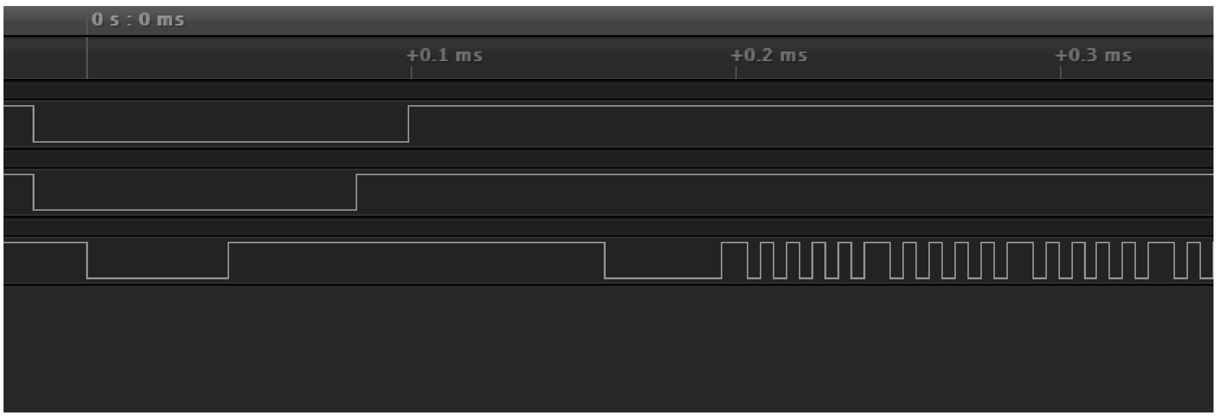
Using the first PWM to modulate the Tx shows that it is possible to generate a break on the TX line.



Using PIT to control delays and the LPUART's break generation.



Compared to same but directly controlling the output:



Thi shows a break set/clear generates 43us (10 bits), which is the only possibility in LPUART (43, 86, 129us etc would be possible by queuing multiple – but timing critical...)