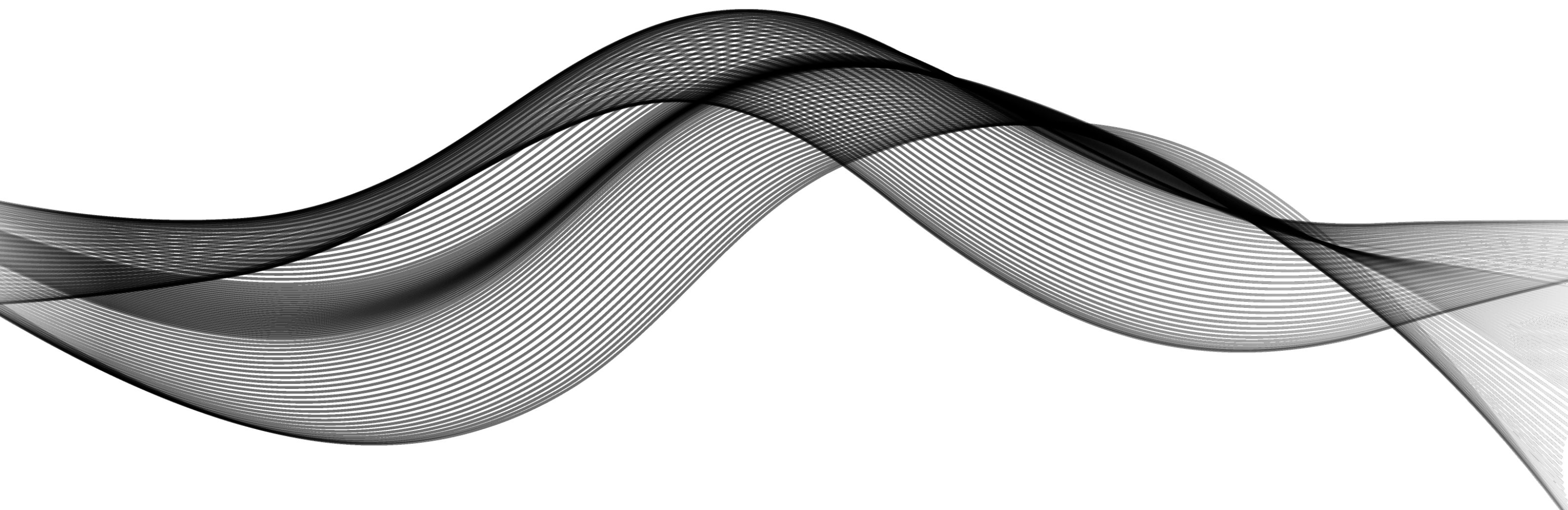


DALI Low Level Driver

MANUAL

Stand 2021/02

MBS

Table of contents

1 Introduction ... 3

2 Safety ... 4

3 Classification and purpose ... 5

4 Structure and features ... 6

5 Hardware requirements ... 7

6 Signalling ... 8

7 DALI Low Level Driver - Main program ... 9
Example ... 9
PCIO-Interrupt (STM32xx), example ... 10

8 Modul *dali_ll_hal.c* ... 13

9 Function *dali_init* ... 18

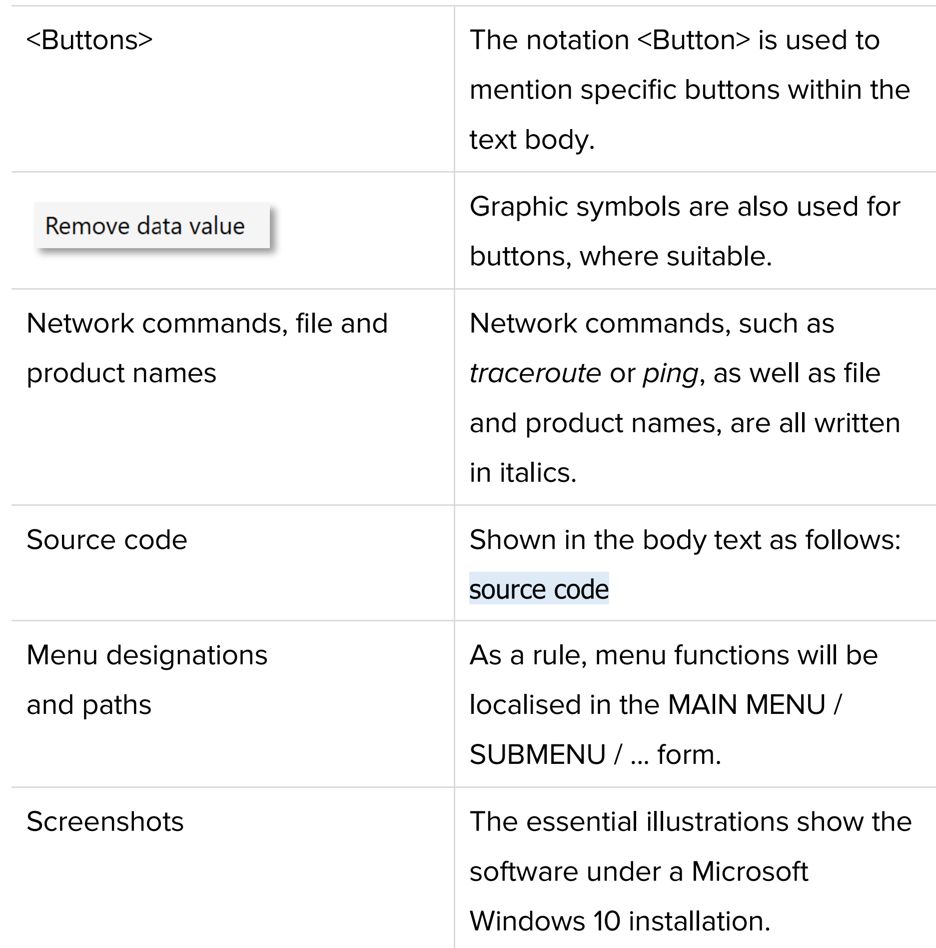
10 Data exchange – DALI low-level driver ⇔ API ... 24

11 General ... 25

12 Product support ... 26

1 Introduction

Notation and symbols used



<Buttons>	The notation <Button> is used to mention specific buttons within the text body.
Remove data value	Graphic symbols are also used for buttons, where suitable.
Network commands, file and product names	Network commands, such as *traceroute* or *ping*, as well as file and product names, are all written in italics.
Source code	Shown in the body text as follows: `source code`
Menu designations and paths	As a rule, menu functions will be localised in the MAIN MENU / SUBMENU / ... form.
Screenshots	The essential illustrations show the software under a Microsoft Windows 10 installation.

Target group

These instructions are intended for specialist personnel, who are familiar with programming and network configuration.

2 Safety

The software present no direct hazards. However, in their function as a gateway between networks in building infrastructures, they are able to seriously disrupt the interaction of network components.

Warning

Misconfiguration of hardware and software!

Faulty configuration of hardware and software can cause malfunctions in the building infrastructure on network components, sensors or actuators, **for example**:

- Monitoring devices, such as fire alarm or intrusion detection systems, are deactivated.
- Machines and fans start up unexpectedly.
- Gate valves and other valves open or close unintentionally.

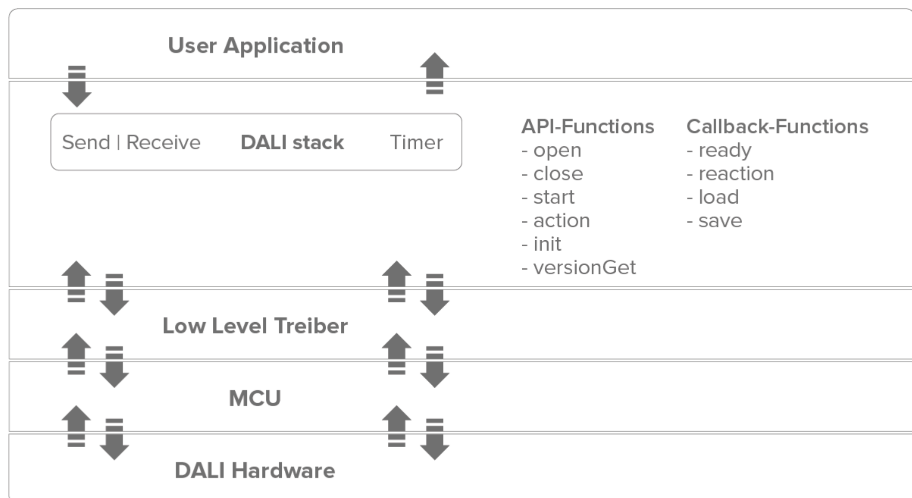
Under certain circumstances, this can lead to serious injuries or death.
The configuration of the software should therefore only be carried out by specialist personnel who are familiar with the network and driver configuration!

3 Classification and purpose

Classification

Driver software to enable the communication of microcontrollers with DALI hardware. Hereinafter referred to as the *Low Level Driver* (LLT).

DALI Architecture



Purpose

In DALI architecture, the *Low Llevel Driver* enables *DALI-APIs*, and therefore *DALI-Applications*, to communicate with the current hardware and thus the *DALI bus*.

Note

Future versions of the driver will be able to support several buses, which explains the parameters already present in the module *dali_ll_hal.c* in low-level routines.

4 Structure and features

General

The module is implemented in such a way that no hardware or operating system-related functions are used.

Such functions are transferred to a module to be created by the user dali_ll_hal.c as *callbacks*.

Although the examples given here are programmed for *STM32 processors.* It should also be possible to implement them on other hardware.

Communication

Communication to and from the driver is also established by means of *callbacks*, as described in detail in the API documentation.

All messages to and from the driver are processed via queues to stop interrupt actions lasting too long. The processing of queues is initiated in the main program or in a *DALI thread* in multi-tasking environments.

As the driver is able to support several instances of *DALI-API* on one hardware, several DALI devices (e.g. one LED and one application controller) can exist on a single hardware. These communicate with one another via the driver and with external devices via the DALI bus. For this purpose, the driver has a *loop layer* which, after a message is sent via the hardware, delivers this to all other instances on the same hardware.

5 Hardware requirements

GPIO (general-purpose input/output)	The *Low Level Driver* needs two GPIOs for writing and reading on the bus. The read GPIO must provide an interrupt in the event of a level change.
Hardware timer	Minimum resolution 10 µs, an interrupt must be available.
CPU	Bus width 32 Bit, minimum frequency 32 MHz.

6 Signalling

In multi-tasking environments such as RTOS, signalling callback is provided, by means of which the *Low Level Driver* can notify the application where data is queued for processing.

The signaling does not have to contain a queue, just signal that something is to be done. When a message is received, all *DaliQueues* are processed in full.

Note
If this mechanism is not used, the callback must be initialized with NULL.

7 DALI Low Level Driver - Main program

Example

Explanation

The dalill_inithardware() function contains everything needed to initialize the target hardware. It may be necessary to access an automatically generated code.

Here the *PCIO*s and the *timer* are initialized. The *timer* is not yet started!

In this function, the two interrupt routines of the *Low Level Driver* are also set on the corresponding vectors.

In the case of the *LLT*, this means the functions dalill_interruptExt for the DALI read pin and dalill_timerinterrupt2 for the timer. Both functions require a parameter of the type dalill_bus_t*. As outlined above, this is envisaged for the future multi-bus operation.

```
#include "dali_ll_hal.h"
#include "dali_ll.h"
#include "dali.h"

dalill_bus_t* pDalill_bus_0;
void DALI_ThreadFunc(void *argument) // oder main()
{
  // do hardware initialization
  dalill_initHardware();
  // init dalistack
  // init dali_ll
  pDalill_bus_0 = dalill_createBusLine(&dalill_getBusState,
                                       &dalill_setBusStateHigh,
                                       &dalill_setBusStateLow);
  // init dalilib (API)
  dali_init(pDalill_bus_0,NULL);
  while (1)
  {
    // Dieses if nur in Multitaskingumgebungen mit Signallingcallback
    // Beispielhaft für Rtos
    if (DALI_FLAG  == osThreadFlagsWait(DALI_FLAG,osFlagsWaitAny,15))
    {
      // something to do ?, not necessary if nothing else should be done in
main
```

```
    while (dalill_isBusy())
    {
      dalill_processQueues();
    }
   }
  }
}
```

PCIO-Interrupt (STM32xx), example

```
extern dalill_bus_t* pDalill_bus_0;
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if(GPIO_Pin == DALI_IN_Pin)
  {
    dalill_interruptExt(pDalill_bus_0);
  } ...
```

For the *TimerInterrupt* in this example, a preliminary routine was used which then supplied the parameter to the real *interrupt routine*:

```
// timerinterrupt needs parameter, so we add it here
void dalill_timerInterrupt()
{
  dalill_timerInterrupt2(pDalill_bus_0);
}
```

Followed by:

```
HAL_TIM_RegisterCallback(&htim16, HAL_TIM_PERIOD_ELAPSED_CB_ID,
dalill_timerInterrupt);
```

Initialization

Then dalill_createBusLine is called. As a parameter, this has three callback functions that must be defined in dali_ll_hal.c. The callbacks are used to test, set and reset the DALI bus lines.

The level HIGH or LOW relates to the DALI bus and not the value of the PCIO. When the PCIO drives the DALI bus in an inverted way, therefore, the PCIO is set to LOW and the bus level is set to HIGH through dalill_setBusStateHigh.

The callbacks are set so that the bus can be set to IDLE as soon as the hardware is activated; this prevents interruptions to the bus.

If dalill_createBusLine returns a value != NULL, pDalill_bus_0 is a pointer to this instance of the hardware driver.

This parameter is now used to call the function dali_init() in the application module.

Dali_init is also described in the DALI-API documentation.

Main loop

Once initializations have been successfully executed, the main loop can be performed.

dalill_isbusy can be used to test whether data requiring processing is present in the read and/or write queue.

Calling dalill_processQueues then prompts all data present in both queues to be processed. This means the data is both passed on to the DALI-API and sent via the bus. The data is also transferred to any other instances of DALI-API via the loop interface.

If nothing else needs to be done in the main loop (apart from operating the DALI stack), only the dalill_processQueues function needs to be permanently called.

Note

Where other actions are performed here, make sure these never block and never last more than a few milliseconds; this will guarantee smooth operation of the *DALI stack*.

In multi-tasking environments such as RTOS, *signalling* can be used as outlined above.

Note

As the low-level driver generates a 10 ms *heartbeat* for every instance of the API and thereby calls up the *timing helper* function of the API, the function dalill_isbusy will report data after 10 ms at the latest in IDLE state.

This or the *timing helper* are suitable for making an LED flash (for example), thereby signalling the readiness of the driver.

8 Modul *dali_ll_hal.c*

This module contains all necessary hardware-related functions. The example code is intended for an *STM32 processor* and the *signalling function* is realized using the *RTOS-APIs*.

The initializations and work routines for the timer are in the upper part.

```
/*M>-------------------------------------------------------------------------
 * Project:       DALI-Stack HAL
 * Description:   Abstraction Layer between DALI-low-level driver and uC
Timer, Interrupts, etc.
 *
 * Copyright (c) by mbs GmbH, Krefeld, info@mbs-software.de
 * All rights reserved.
 -------------------------------------------------------------------------<M*/

#include "app_common.h"
#include "system.h"
#include "dali_ll_hal.h"

TIM_HandleTypeDef htim16;
void Error_Handler(void);
void Tim16BaseMspInitCb(TIM_HandleTypeDef *htim);
void Tim16BaseMspDeInitCb(TIM_HandleTypeDef *htim);
void Tim16init(void);
void dalill_timerInterrupt();
void Tim16init(void)
{
  htim16.Instance = TIM16;
  htim16.Init.Prescaler = 31;
  htim16.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim16.Init.Period = 10000;
  htim16.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim16.Init.RepetitionCounter = 0;
  htim16.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim16) != HAL_OK)
  {
    Error_Handler();
  }
}

/*--------------------------------------------------------------*/
```

```
void Tim16BaseMspInitCb(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN TIM16_MspInit 0 */
  /* USER CODE END TIM16_MspInit 0 */
    /* Peripheral clock enable */
    __HAL_RCC_TIM16_CLK_ENABLE();
    /* TIM16 interrupt Init */
    HAL_NVIC_SetPriority(TIM1_UP_TIM16_IRQn, 5, 0);
    HAL_NVIC_EnableIRQ(TIM1_UP_TIM16_IRQn);
}

/*-------------------------------------------------------------*/

void Tim16BaseMspDeInitCb(TIM_HandleTypeDef *htim)
{
  __HAL_RCC_TIM16_CLK_DISABLE();
}

/*-------------------------------------------------------------*/

void dalill_initHardware()
{
  HAL_TIM_RegisterCallback(&htim16, HAL_TIM_BASE_MSPINIT_CB_ID,
Tim16BaseMspInitCb);
  HAL_TIM_RegisterCallback(&htim16, HAL_TIM_BASE_MSPDEINIT_CB_ID,
Tim16BaseMspDeInitCb);
  Tim16init();
  HAL_TIM_RegisterCallback(&htim16, HAL_TIM_PERIOD_ELAPSED_CB_ID,
dalill_timerInterrupt);
}

/*-------------------------------------------------------------*/

// extern dalill_bus_t dalill_bus_lines_g[]; Only used for Multibus !

uint32_t dalill_getCurrentTimerVal(dalill_bus_t* pDalill_bus)
{
  return htim16.Instance->CNT;
}
uint32_t dalill_getTimerPeriod(dalill_bus_t* pDalill_bus)
{
  return htim16.Init.Period+1;
}

void dalill_startTimer()
{
  HAL_TIM_Base_Start_IT(&htim16);
}
```

```
void dalill_setTimerPeriod(uint16_t uPeriod,dalill_bus_t* pDalill_bus)
{
 __HAL_TIM_SET_AUTORELOAD(&htim16, uPeriod +
dalill_getCurrentTimerVal(pDalill_bus) - 1);
}

/*!  Set/Read GPIOs of Bus 0 */

uint8_t dalill_getBusState()
{
  return HAL_GPIO_ReadPin(DALI_IN_GPIO_Port, DALI_IN_Pin)?0:1;
}

void dalill_setBusStateHigh()
{
  HAL_GPIO_WritePin(DALI_OUT_GPIO_Port, DALI_OUT_Pin,
GPIO_PIN_RESET);
}

void dalill_setBusStateLow()
{
  HAL_GPIO_WritePin(DALI_OUT_GPIO_Port, DALI_OUT_Pin,
GPIO_PIN_SET);
}

// Signalling

extern osThreadId_t DALI_ThreadId;
void dalill_signalToThread()
{
  osThreadFlagsSet (DALI_ThreadId, DALI_FLAG);
}

// block and release interrupts

void enableIRQ()
{
  __enable_irq();
}

void disableIRQ()
{
  __disable_irq();
}
```

Target hardware

If there is no direct possibility for the target hardware to read and/or change the current timer value or set the timer trigger value, the functions dalill_getCurrentTimerVal, dalill_getTimerPeriod and dalill_setTimerPeriod can implement themselves.

TimerInterrupt routine

For this, the timer interrupt routine must be called at least every 10 µs and, depending on a counter, the timer routine of the *Low Level Driver*.

The variables required for this are already defined in the structure dalill_bus_t in dali_ll.h. These are the member variables of type uint32_t, tick_cnt and tim_period.

GPIO manipulation

In this example, the routines for GPIO manipulation are intended for hardware with an inverting bus interface.

Timer functions

The timer functions assume a timer that counts up and whose period can be extended with alill_setTimerPeriod, even while it is already running.

The function dalill_getCurrentTimerVal assumes, that the timer counter continues to count even if the period is extended. So, no timer interrupt is triggered during the extension.

"getTimerPeriod" returns the current maximum value of the timer.

> **Note**
>
> If this behavior cannot be achieved by the timer of the target hardware, it must be simulated in order for the *low level driver* to function properly.

MBS

IRQ functions

The callbacks "enableIRQ" and "disableIRQ" are used to block and enable the processor interrupts. This is necessary so that the write and read operations to the I/O queues in the interrupt are atomic.

Because this action is different on each target hardware, it was moved to callbacks.

Note
If this mechanism is not used, the callback must be initialized with NULL.

9 Function *dali_init*

A dali_init function is described here for two instances of the *DALI stack*.

Note

Most parts are also described in the API documentation.

The instances of the stack are initially created at the start of dali_init. Further callbacks for the *low-level driver* are initialized and the timer is then started. The driver is thereby ready to use.

```
/***************************************************************/
/* Variables for two instances of the stack
***************************************************************/

// application controller

dalilib_action_t    action;
uint8_t             bDaliStackReady;
dalilib_instance_t  pDaliStackInstance;
dalilib_cfg_t       ctrl_device_config;

// LED DT6-Device

dalilib_action_t    actionLED;
uint8_t             bDaliStackReadyLED;     //  for LED
dalilib_instance_t  pDaliStackInstanceLED;  //  for LED
dalilib_cfg_t       ctrl_gear_config;


/***************************************************************/
/* initialize the HAL driver and the DALI stack
***************************************************************/

void dali_init(dalill_bus_t* pDalill_bus,dalill_bus_t *pDalill_bus2)
{

  dalill_base_t dalill_base;

  // For the application controller

  pDaliStackInstance = NULL;
  bDaliStackReady = 0;
  // create new DALI stack instance
  pDaliStackInstance = dalilib_createinstance();
  if (NULL == pDaliStackInstance)
  {       // error
      return;
  }
```

```
  // Tell lowleveldriver from this instance of the stack
  addInstance(pDalill_bus,pDaliStackInstance);
  // create and configure DALI stack as single application controller
  dali_create_application_controller_config();
  // add Low Level structure to DALI stack instance and vice versa
  ctrl_device_config.context = pDalill_bus;

  // Only one initial value. This will be overwritten by the loop layer of the
ll-Driver
  pDalill_bus->context = pDaliStackInstance;
  // initialize DALI stack instance
  if (R_DALILIB_OK != dalilib_init(pDaliStackInstance ,
&ctrl_device_config))
  {
     // error
     return;
  }
  // start DALI stack instance
  if (R_DALILIB_OK != dalilib_start(pDaliStackInstance))
  {
     // error
     return;
  }

  // For the ledDevice

  pDaliStackInstanceLED = NULL;
  bDaliStackReadyLED = 0;

  // create new DALI stack instance

  pDaliStackInstanceLED = dalilib_createinstance();
  if (NULL == pDaliStackInstanceLED)
  {        // error
      return;
  }

  // Tell lowleveldriver from this instance of the stack
  addInstance(pDalill_bus,pDaliStackInstanceLED);

  // create and configure DALI stack as a LED (DT6-Device)

  dali_create_gear_configLED();

  // add Low Level structure to DALI stack instance and vice versa

  ctrl_gear_config.context =pDalill_bus;

  // initialize DALI stack instance
  if (R_DALILIB_OK != dalilib_init(pDaliStackInstanceLED ,
&ctrl_gear_config))
  {
```

```
    // error
    return;
  }

  // start DALI stack instance

  if (R_DALILIB_OK != dalilib_start(pDaliStackInstanceLED))
  {
    // error
    return;
  }

  // initialize callback functions for DALI Low Level Driver

  dalill_base.debug_mode          = 0;
  dalill_base.max_frame_length    = 32;
  dalill_base.rx_high_offset      = 40;   // 60 DALI 2 click
  dalill_base.rx_low_offset       = 40;   // 60 DALI 2 click
  dalill_base.tx_high_offset      = 20;   // 35 DALI 2 click
  dalill_base.tx_low_offset       = 20;   // 35 DALI 2 click
  // Api functions
  dalill_base.dalilltimingHelper = &dalill_timingHelper;
  dalill_base.dalilltoDalilib     = &dalill_toDalilib;
  // funtions in dali_ll_hal.c
  dalill_base.getCurrentTimerVal = &dalill_getCurrentTimerVal;
  dalill_base.getTimerPeriod      = &dalill_getTimerPeriod;
  dalill_base.setTimerPeriod      = &dalill_setTimerPeriod;
  dalill_base.startTimer          = &dalill_startTimer;

  // It is important to set this to NULL if no signalling is used !!!
  // or implement a dummy function in dali_ll_hal.c

  dalill_base.signalToThread      = &dalill_signalToThread;

  // functions to block and release interrupts
  // It is important to set this pointer to NULL if not used

  dalill_base.enableIRQ           = enableIRQ;
  dalill_base.disableIRQ          = disableIRQ;

  // create all data for LL-Driver and start the timer
  dalill_createBase(&dalill_base);
}
```

The variables

int16_t dalill_base.rx_high_offset,

int16_t dalill_base.rx_low_offset,

int16_t dalill_base.tx_high_offset and

int16_t dalill_base.tx_low_offset are used, to compensate the switching times of the interface between *PCIO* and the *DALI bus*.

This is necessary to ensure the driver does not interpret its own signals as a collision when sending and thereby block itself.

The variables

rx_high_offset and

rx_low_offset are used to control the receiving process.

tx_high_offset and

tx_low_offset are used to control the sending process.

The unit of this variables is microseconds.

Note

If needed, negative values are also accepted here!

Procedure for setting

Note

The variables for the transmitting direction can only be set with the support of a *Dali Tracer* or an oscilloscope.

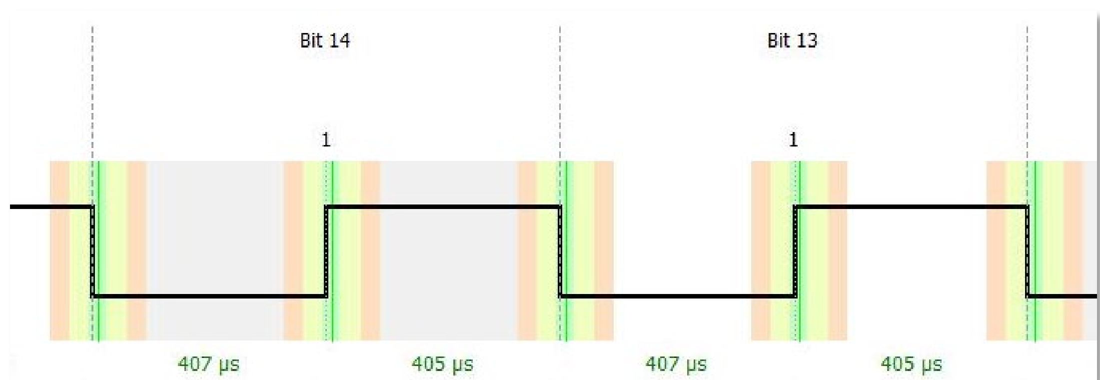
Sending offsets

With tx_high_offset the time can be changed in which the level of a „*half-bit*" should remain at LOW. If this time is too short, the parameter must be **reduced** by the corresponding number of microseconds.

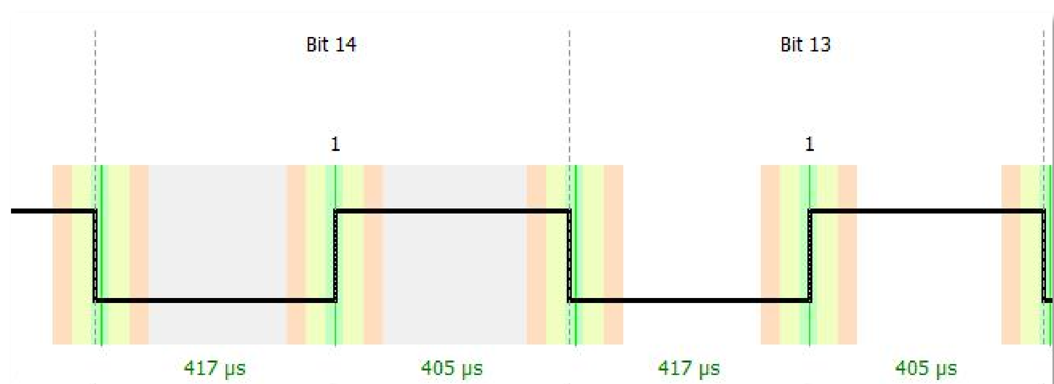
With tx_low_offset the time can be changed in which the level should remain HIGH. If this time is too short, the parameter must be **increased**.

The naming of the parameters refers to the change to the target state HIGH or LOW.

Examples In the following example, both times are too short:



After reducing tx_high_offset by 10 microseconds, the following picture is obtained:



After increasing tx_low_offset by 12 microseconds, the following picture is obtained:

This way the function is then error-free.

Read offsets

Setting the read offsets is relatively simple.

First rx_high_offset and rx_low_offset are set to value zero.

When another device transmits, collisions are most likely reported by the *low level driver*.

The offsets are then slowly increased in steps of 10.

As soon as the collisions stop, this value is to documented. The offsets are then increased further until collisions are reported again.

The optimal offset is in the middle between the lower and the upper determined value. With this, an error free reception should be possible.

rx_high_offset and rx_low_offset normally have the same values.

10 Data exchange – DALI low-level driver ⇔ API

The *Low Level Driver* sends messages to the *API* via the dalill_toDaliLib function, as in the following example:

```
/****************************************************************/
/* forward low level frames to dalilib
****************************************************************/
void dalill_toDalilib(void * p_context, dalill_frame_t* p_frame)
{
       dalilib_receive(p_context, (dalilib_frame_t*)p_frame);
}
```

The *API* sends data to the *Low Level Driver* via the dali_send_callback. This is loaded with data via the function dall_ll_SendQueue. To ensure the *loop layer* functions properly, it is important to ensure the right instance of the APIs is entered in the third parameter.

For example:

```
/****************************************************************/
/* will be called by the DALI library if it wants to send a DALI message
 * to the driver
 * result: 0: success
 ****************************************************************/
static uint8_t dali_send_callback(void * p_context, dalilib_frame_t*
p_frame)
{
  uint8_t result = 0;
  if(bDaliStackReady)
  {
    result = dalill_pushSendQueue(p_context,(dalill_frame_t* ) p_frame,
                                                pDaliStackInstance);
  }
  return result;
}
```

The last major callback is the *timing helper*, which makes it possible to check and comply with API times.

For example:

```
/****************************************************************/
/* called every 10 ms
****************************************************************/
void dalill_timingHelper(void *pInstance,uint32_t dali_time_ticker)
{
  dalilib_timingHelper(pInstance, dali_time_ticker);
}
```

11 General

The DALI low-level stack is usually supplied with a sample application to demonstrate the functioning of the processes described here.

12 Product support

Manufacturer	MBS GmbH Römerstraße 15 47809 Krefeld
Telephone	+49 21 51 72 94-0
Fax	+49 21 51 72 94-50
E-Mail	support@mbs-solutions.de
Internet	www.mbs-solutions.de
	wiki.mbs-software.info
Service times	Monday to Friday: 8:30 to 12:00 13:00 to 17:00