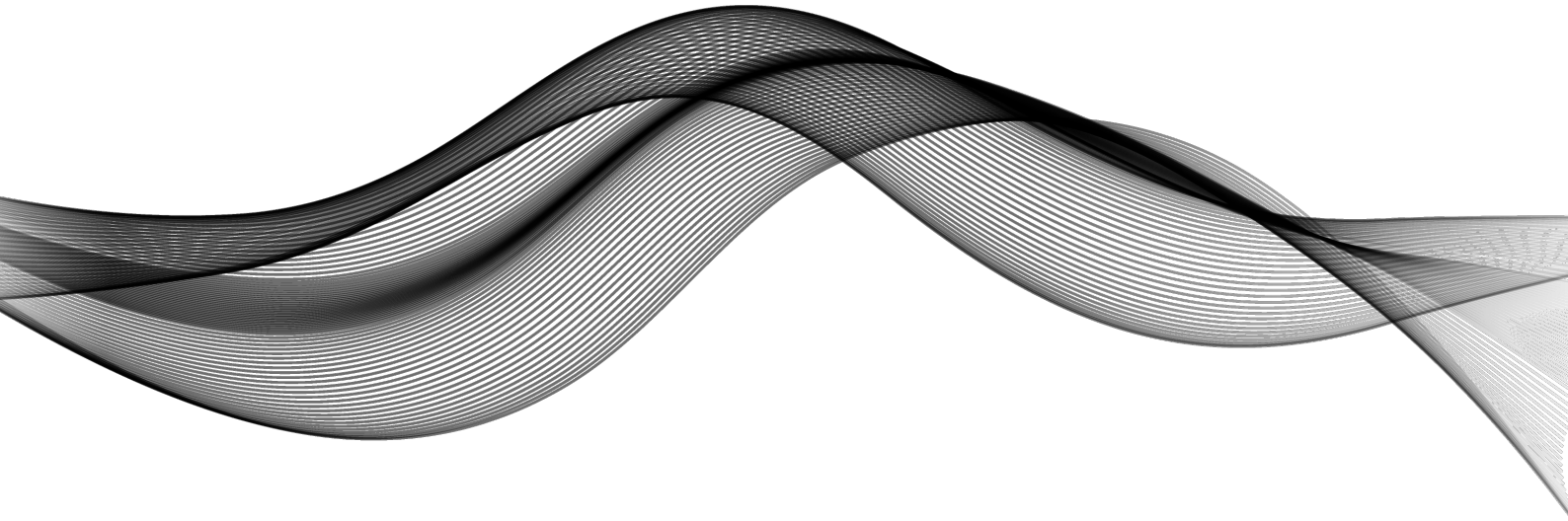


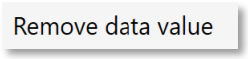
DALI Low Level Treiber
HANDBUCH

Stand 2021/02

Inhaltsverzeichnis

1 Einleitung

Verwendete Schreibweisen und Symbole

<Schaltflächen>	Für Schaltflächen die im fließenden Text genannt werden müssen, wird die Schreibweise <Schaltfläche> verwendet.
	An geeigneter Stelle werden für Schaltflächen auch grafische Symbole verwendet.
Netzwerkbefehle, Datei- und Produktnamen sowie Menüeinträge	Netzwerkbefehle wie z.B. *traceroute* oder *ping* werden kursiv geschrieben. Datei- und Produktnamen und Menüeinträge ebenfalls.
Quelltextinhalte	Im fließenden Text werden sie wie folgt dargestellt: `Quelltext`
Menübezeichnungen und -pfade	Menüfunktionen werden in der Form: MAIN MENU / SUBMENU / ... lokalisiert.
Screenshots	Verwendeten Abbildungen zeigen die Software unter einer Microsoft Windows 10 Installation.

Zielgruppe

Diese Anleitung richtet sich an Fachpersonal, welches mit der Programmierung und Netzwerkkonfiguration vertraut ist.

2 Sicherheit

Von der Software an sich gehen keine direkten Gefahren aus. Allerdings ist sie als Treiber in Netzwerken von Gebäude-infrastrukturen in der Lage, das Zusammenwirken von Netzwerk-komponenten empfindlich zu stören.

Warnung

Fehlkonfiguration von Hard- und Software!

Durch fehlerhafte Konfiguration von Hard- und Software können an Netzwerkkomponenten, Sensoren oder Aktoren Fehlfunktionen in der Gebäudeinfrastruktur auftreten, wie **zum Beispiel**:

- Überwachungseinrichtungen werden deaktiviert,
- Maschinen und Lüfter laufen unerwartet an,
- Schieber und Ventile öffnen oder schließen unbeabsichtigt.

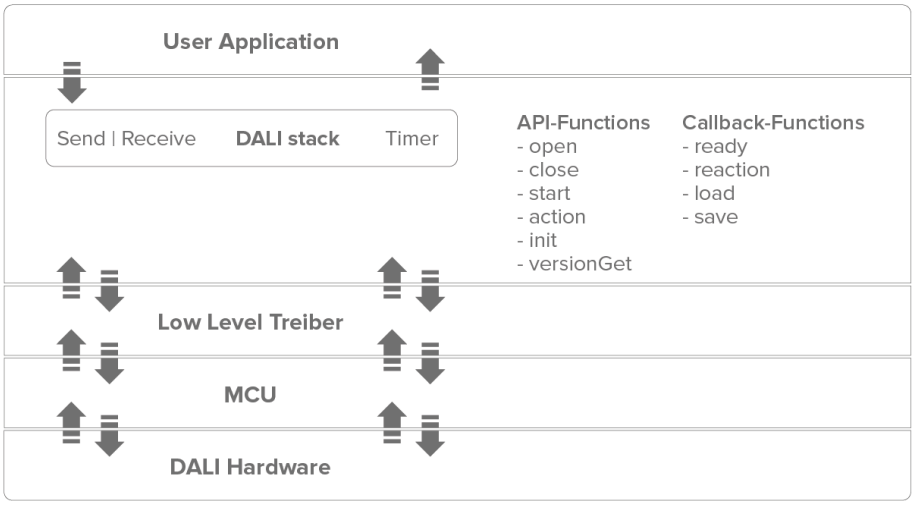
Das kann unter Umständen zu schweren Verletzungen oder zum Tod von Personen führen.
Die Konfiguration der Software sollte deshalb nur von Fach-personal, welches mit der Netzwerk- und Treiberkonfiguration vertraut ist, vorgenommen werden!

3 Einordnung und Einsatzzweck

Einordnung

Treibersoftware für die Kommunikation von Microcontrollern mit DALI-Hardware. Im Folgenden als *Low Level Treiber* (LLT) genannt.

DALI Architektur



Einsatzzweck

Der *Low Level Treiber* dient in der DALI-Architektur zur Kommunikation der *DALI-API*s und damit der *DALI-Applikationen* mit der aktuellen Hardware und damit mit dem *DALI-Bus*.

> **Hinweis**
>
> In künftigen Versionen soll der Treiber auch mehrere Busse unterstützen können, was den im Modul *dali_ll_hal.c* bereits vorkommenden Parameter in den Low Level Routinen erklärt.

4 Aufbau und Leistungsmerkmale

Allgemein

Das Modul ist so implementiert, dass es keine hardware- oder betriebssystemnahen Funktionen benutzt.

Solche Funktionen sind in ein vom Anwender zu erstellendem Modul dali_ll_hal.c als *Callbacks* ausgelagert.

Alle hier aufgezeigten Beispiele sind für *STM32-Prozessoren* programmiert. Sie sollten aber auch auf anderer Hardware zu implementieren sein.

Kommunikation

Die Kommunikation von und zum Treiber findet auch über *Callbacks* statt. Diese sind ausführlich in der API-Dokumentation beschrieben.

Alle Nachrichten von und zum Treiber werden über Queues abgewickelt um zu verhindern, dass Interrupt-Aktionen zu lange dauern. Die Verarbeitung der Queues wird im Hauptprogramm oder bei Multitasking-Umgebungen in einem *DALI-Thread* angestoßen.

Der Treiber ist in der Lage, mehrere Instanzen der *DALI-API* auf einer Hardware zu unterstützen. Das heißt, dass auf einer Hardware mehrere DALI-Geräte, z. B. eine LED und ein Applikations-Controller existieren können. Sie kommunizieren über den Treiber sowohl untereinander, als auch über den DALI-Bus mit externen Geräten. Dafür besitzt der Treiber eine *Loop-Schicht* die, nachdem eine Nachricht über die Hardware gesendet wurde, diese auch an alle anderen Instanzen auf derselben Hardware zustellt.

5 Hardwarevoraussetzungen

GPIO (general-purpose input/output)	Der *Low Level Treiber* benötigt zwei GPIOs für das Schreiben und Lesen auf dem Bus. Der Lese-GPIO muss einen Interrupt bei Pegelwechsel zur Verfügung stellen.
Hardwaretimer	Auflösung mindesten 10 µs, ein Interrupt muss zur Verfügung stehen.
CPU	Busbreite 32 Bit, Taktfrequenz min. 32 MHz.

6 Signalling

Für Multitasking-Umgebungen wie RTOS, ist eine *Signallingcallback* vorgesehen mit dem der *Low Level Treiber* die Applikation benachrichtigen kann, wenn in den Queues Daten zur Verarbeitung anstehen.

Das *Signalling* muss keine Queue enthalten, sondern nur signalisieren das etwas zu tun ist. Bei Erhalt der Nachricht werden alle *DaliQueues* komplett abgearbeitet.

Hinweis

Wird dieser Mechanismus nicht benutzt, muss der Callback mit NULL initialisiert werden.

7 DALI Low Level Driver - Mainprogramm

Beispiel

Erläuterung

Die Funktion dalill_inithardware() enthält alles was zum Initialisieren der Zielhardware notwendig ist. Eventuell muss hier auf automatisch generierten Code zurückgegriffen werden.

Hier werden die *PCIO*s und der *Timer* initialisiert. Der *Timer* wird hier noch nicht gestartet!

In der Funktion werden auch die beiden Interrupt Routinen des *Low Level Treibers* mit auf die entsprechenden Vektoren gelegt.

Das sind im Fall des *LLT* die beiden Funktionen dalill_interruptExt für den Dali-Read-Pin und dalill_timerinterrupt2 für den Timer. Beide Funktionen benötigen einen Parameter vom Typ dalill_bus_t*. Dieser ist wie schon beschrieben, für den künftigen Multibusbestrieb vorgesehen.

```
#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), Beispiel

```
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);
  } ...
```

Für den *TimerInterrupt* wurde hier beispielhaft eine Vorschaltroutine benutzt, welche dann die echte *Interrupt Routine* mit dem Parameter versorgt:

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

Und darauffolgend:

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

Initialisierung

Anschließend wird dalill_createBusLine aufgerufen. Diese hat als Parameter drei Callbackfunktionen, die in dali_ll_hal.c definiert sein müssen. Diese Callbacks dienen zum Testen, Setzen und Zurücksetzen der DALI-Busleitungen.

Der Pegel HIGH oder LOW bezieht sich dabei auf den DALI-Bus und nicht auf den Wert des PCIO. Das heißt, wenn der PCIO den DALI-Bus invertiert treibt, so wird durch dalill_setBusStateHigh der PCIO auf LOW und damit der Buspegel auf HIGH gesetzt.

Diese Callbacks werden schon hier gesetzt, damit der Bus unmittelbar nach Einschalten der Hardware auf IDLE gesetzt werden kann und so keine Störungen auf dem Bus erzeugt werden.

Liefert dalill_createBusLine einen Wert != NULL ist pDalill_bus_0 ein Zeiger auf diese Instanz des Hardwaretreibers.

Mit diesem Parameter wird nun die Funktion dali_init() im Applikationsmodul aufgerufen.

Dali_init ist auch in der Dokumentation der DALI-API beschrieben.

Hauptschleife

Nach erfolgreich ausgeführten Initialisierungen kann die Hauptschleife ausgeführt werden.

Mit dalill_isbusy kann getestet werden, ob sich in der Read- und/oder Writequeue Daten befinden, die auf Weiterverarbeitung warten.

Der Aufruf von dalill_processQueues führt dann dazu, dass alle in beiden Queues vorhandenen Daten verarbeitetet werden. Das heißt, sie werden sowohl an die DALI-API weitergegeben, als auch über den Bus verschickt. Über das Loopinterface werden sie auch an eventuell vorhanden weitere Instanzen der DALI-API weitergegeben.

Soll in der Hauptschleife nichts anderes getan werden, als den DALI Stack zu bedienen, muss nur die Funktion dalill_processQueues permanent aufgerufen werden.

Hinweis

Werden hier noch andere Aktionen ausgeführt, so ist darauf zu achten, dass diese nie blockieren und auch nicht länger als wenige Millisekunden dauern um die reibungslose Arbeit des *DALI Stack*s zu garantieren.

In Multitasking-Umgebungen wie RTOS kann das hier beschriebene *Signalling* benutzt werden.

Hinweis

Weil der Low Level Treiber für jede Instanz der API einen 10 ms *Heartbeat* erzeugt und damit die *Timinghelper*-Funktion der API aufruft, wird die Funktion `dalill_isbusy` im IDLE-Fall spätestens nach 10 ms Daten melden.

Hier oder im *Timinghelper* ist eine gute Position, um z. B. eine LED blinken zu lassen, welche die Einsatzbereitschaft des Treibers signalisiert.

8 Modul *dali_ll_hal.c*

Dieses Modul enthält alle benötigten hardwarenahen Funktionen. Der Beispielcode ist für einen *STM32-Prozessor* vorgesehen und die *Signallingfunktion* wird mit Hilfe der *RTOS-API*s realisiert.

Im oberen Teil befinden sich die Initialisierungen und die Arbeitsroutinen für den Timer.

```
/*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();
}
```

Zielhardware

Sollte die Zielhardware keine direkte Möglichkeit zur Verfügung stellen, den aktuellen Timerwert auszulesen und/oder zu verändern oder auch den Timerauslösewert zu setzen, so können die Funktionen dalill_getCurrentTimerVal, dalill_getTimerPeriod und dalill_setTimerPeriod selbst implementiert werden.

TimerInterrupt-Routine

Hierzu muss die TimerInterupt-Routine mindestens alle 10 µs aufgerufen werden und weiterhin, abhängig von einem Zähler, auch die Timerroutine des *Low Level Treibers*.

Die dazu notwendigen Variablen sind schon in der Struktur dalill_bus_t in dali_ll.h definiert. Das sind die Membervariablen vom Typ uint32_t tick_cnt und tim_period.

GPIO-Manipulation

Die Routinen zur GPIO-Manipulation in diesem Beispiel sind für eine Hardware mit invertierendem Businterface vorgesehen.

Timerfunktionen

Die Timerfunktionen gehen von einem Timer aus, der aufwärts zählt und dessen Periode mit alill_setTimerPeriod verlängert werden kann, auch noch während sie bereits abläuft.

Die Funktion dalill_getCurrentTimerVal geht davon aus, dass der Timercounter kontinuierlich weiter zählt auch wenn die Periode verlängert wird. Bei der Verlängerung wird also kein Timer-Interrupt ausgelöst.

„getTimerPeriod“ liefert den aktuellen Maximalwert des Timers.

Hinweis

Ist dieses Verhalten durch den Timer der Zielhardware so nicht zu erreichen, muss es simuliert werden damit der *Low Level Treiber* ordnungsgemäß funktioniert.

IRQ-Funktionen

Die Callbacks „enableIRQ“ und „disableIRQ“ dienen dem Blockieren und Freigeben der Prozessorinterrupts. Das ist notwendig damit die Schreib- und Lesevorgänge auf die I/O-Queues im Interrupt atomar sind.

Weil diese Aktion auf jeder Zielhardware anders ist, wurde sie in Callbacks verlegt.

Hinweis

Wird dieser Mechanismus nicht benutzt, muss der Callback mit NULL initialisiert werden.

9 Funktion *dali_init*

Hier wird eine dali_init Funktion für zwei Instanzen des *DALI stack* beschrieben.

> **Hinweis**
>
> Die meisten Teile sind auch in der API-Dokumentation beschrieben.

Zu Beginn der dali_init werden zuerst die Instanzen des Stacks erzeugt. Dann werden weitere Callbacks für den *Low Level Treiber* initialisiert und danach der Timer gestartet. Damit ist der Treiber einsatzbereit.

```
/*************************************************************/
/* 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);
}
```

Die Variablen
int16_t dalill_base.rx_high_offset,
int16_t dalill_base.rx_low_offset,
int16_t dalill_base.tx_high_offset und
int16_t dalill_base.tx_low_offset werden genutzt, um die Schaltzeiten des Interfaces zwischen *PCIO* und dem *DALI-Bus* zu kompensieren.

Das ist notwendig, damit der Treiber beim Senden nicht seine eigenen Signale als Kollision interpretiert und sich damit selbst blockiert.

Die Variablen
rx_high_offset und
rx_low_offset dienen zur Steuerung des Empfangsprozesses.

tx_high_offset und
tx_low_offset dienen zur Steuerung des Sendeprozesses.

Die Einheit dieser Variablen ist Mikrosekunden.

Hinweis

Falls erforderlich sind hier auch negative Werte möglich!

Vorgehen zur Einstellung

Hinweis

Die Variablen für die Senderichtung können nur mit Unterstützung eines *Dali Tracers* oder eines Oszilloskops einstellen werden.

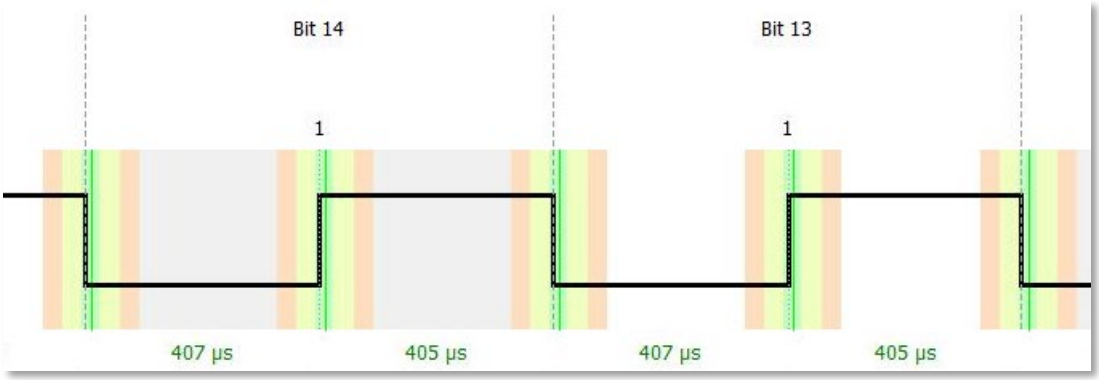
Sendeoffsets

Mit tx_high_offset kann die Zeit verändert werden, in der der Pegel eines Halb-Bits auf LOW bleiben soll. Ist diese Zeit zu kurz, muss der Parameter um die entsprechende Zahl vom Mikrosekunden **verkleinert** werden.

Mit tx_low_offset kann die Zeit verändert werden, in der der Pegel auf HIGH bleiben soll. Ist diese Zeit zu kurz, muss der Parameter **vergrößert** werden.

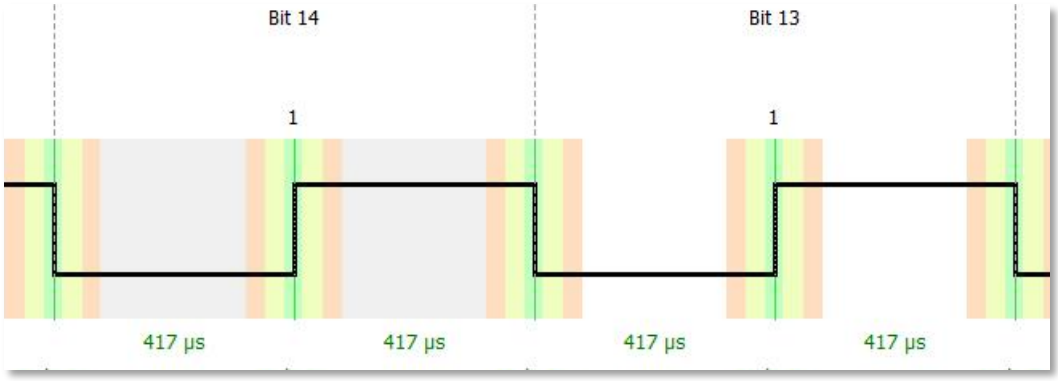
Die Namensgebung der Parameter bezieht sich dabei auf den Wechsel in den Zielzustand HIGH beziehungsweise LOW.

Beispiele Im folgenden Beispiel sind beide Zeiten zu kurz:



Nach Verkleinerung von tx_high_offset um 10 Mikrosekunden erhält man folgendes Bild:

Nach Vergrößerung von tx_low_offset um 12 Mikrosekunden erhält man folgendes Bild:



Damit ist die Funktion dann fehlerfrei.

Leseoffsets

Die Einstellung der Leseoffsets ist relativ einfach.

Zuerst werden rx_high_offset und rx_low_offset auf den Wert Null gesetzt.

Wenn ein anderes Device sendet, werden mit hoher Wahrscheinlichkeit vom *Low Level Treiber* Kollisionen gemeldet.

Die Offsets werden dann in 10er Schritten langsam erhöht.

Sobald die Kollisionen aufhören, wird dieser Wert festgehalten. Die Offsets werden dann weiter erhöht bis erneut Kollisionen gemeldet werden.

Der optimale Offset liegt in der Mitte zwischen dem unteren und dem oberen ermitteltem Wert. Damit sollte ein fehlerfreier Empfang möglich sein.

rx_high_offset und rx_low_offset haben normalerweise die gleiche Werte.

10 Datenaustausch - DALI Low Level Driver ⇔ API

Der *Low Level Treiber* schickt seine Nachrichten über die Funktion dalill_toDaliLib zur *API*, wie im folgenden Beispiel:

```
/*************************************************************/
/* 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);
}
```

Die *API* schickt ihre Daten über den dali_send_callback an den *Low Level Treiber*. Dieser wird über die Funktion dall_ll_SendQueue mit den Daten beschickt. Wichtig ist, dass im dritten Parameter die richtige Instanz der APIs eingetragen wird, damit der *Loop-Layer* richtig arbeitet.

Zum Beispiel:

```
/*************************************************************/
/* 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;
}
```

Der letzte wichtige Callback ist der *Timinghelper*. Dieser ermöglicht es der API-Zeiten zu kontrollieren und einzuhalten.

Zum Beispiel:

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

11 Allgemeines

Der DALI Low Level Stack wird im Allgemeinen mit einer Beispielapplikation geliefert, die alle hier beschriebenen Vorgänge funktional zeigt.

12 Produktunterstützung

Hersteller	MBS GmbH Römerstraße 15 47809 Krefeld
Telefon	+49 21 51 72 94-0
Telefax	+49 21 51 72 94-50
E-Mail	support@mbs-solutions.de
Internet	www.mbs-solutions.de
	wiki.mbs-software.info
Servicezeiten	Montag bis Freitag: 8:30 bis 12:00 Uhr 13:00 bis 17:00 Uhr