NEC

Application Note

Digital Addressable Lighting Interface (DALI)

78K0/Ix2 Series

Lighting ASSP

Document No. U19884EE1V0AN00
Date published July 2009

Printed in Germany

Legal Notes

- **The information in this document is current as of July, 2008. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.**

- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.

- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.

- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.

- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.

- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".

- The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.
  "Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.
  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime

systems, safety equipment and medical equipment (not specifically designed for life support).
"Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

The quality grade of NEC Electronics products is "Standard" unless otherwise expressly specified in NEC Electronics data sheets or data books, etc. If customers wish to use NEC Electronics products in applications not intended by NEC Electronics, they must contact an NEC Electronics sales representative in advance to determine NEC Electronics' willingness to support a given application.

(Note)
(1) "NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.
(2) "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives anddistributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and otherlegal issues may also vary from country to country.

NEC Electronics Corporation
1753, Shimonumabe, Nakahara-ku,
Kawasaki, Kanagawa 211-8668, Japan
Tel: 044 4355111
http://www.necel.com/

[America]

NEC Electronics America, Inc.
2880 Scott Blvd.
Santa Clara, CA 95050-2554,
U.S.A.
Tel: 408 5886000
http://www.am.necel.com/

[Europe]

NEC Electronics (Europe) GmbH
Arcadiastrasse 10
40472 Düsseldorf, Germany
Tel: 0211 65030
http://www.eu.necel.com/

United Kingdom Branch
Cygnus House, Sunrise Parkway
Linford Wood, Milton Keynes
MK14 6NP, U.K.
Tel: 01908 691133

Succursale Française
9, rue Paul Dautier, B.P. 52
78142 Velizy-Villacoublay Cédex
France
Tel: 01 30675800

Tyskland Filial
Täby Centrum
Entrance S (7th floor)
18322 Täby, Sweden
Tel: 08 6387200

Filiale Italiana
Via Fabio Filzi, 25/A
20124 Milano, Italy
Tel: 02 667541

Branch The Netherlands
Steijgerweg 6
5616 HS Eindhoven,
The Netherlands
Tel: 040 2654010

[Asia & Oceania]

NEC Electronics (China) Co., Ltd
7th Floor, Quantum Plaza, No. 27
ZhiChunLu Haidian District,
Beijing 100083, P.R.China
Tel: 010 82351155
http://www.cn.necel.com/

NEC Electronics Shanghai Ltd.
Room 2511-2512, Bank of China
Tower,
200 Yincheng Road Central,
Pudong New Area,
Shanghai 200120, P.R. China
Tel: 021 58885400
http://www.cn.necel.com/

NEC Electronics Hong Kong Ltd.
12/F., Cityplaza 4,
12 Taikoo Wan Road, Hong Kong
Tel: 2886 9318
http://www.hk.necel.com/

NEC Electronics Taiwan Ltd.
7F, No. 363 Fu Shing North Road
Taipei, Taiwan, R.O.C.
Tel: 02 27192377

NEC Electronics Singapore Pte. Ltd.
238A Thomson Road,
#12-08 Novena Square,
Singapore 307684
Tel: 6253 8311
http://www.sg.necel.com/

NEC Electronics Korea Ltd.
11F., Samik Lavied'or Bldg., 720-2,
Yeoksam-Dong, Kangnam-Ku, Seoul,
135-080, Korea Tel: 02-558-3737
http://www.kr.necel.com/

Table of Contents

Chapter 1 DALI Overview

1.1 Introduction

DALI stands for Digital Addressable Lighting Interface and is a royalty-free, non-proprietary, two-way, open, interoperable digital protocol for controlling lighting. It is standardized in accordance with international open standards in the United States and Europe. This protocol is used to dim multi ballasts or LEDs in a system.

DALI is used in a network consisting of up to 64 short addresses and 16 group addresses that performs half-duplex command communication between one master and one or more slaves. DALI commands are used to specify the dimming level in 8-bit accuracy or 256 steps, save any dimming level as a scene and switch between different scenes, and specify other settings. The DALI protocol communicates at a rate of 1,200 Hz +/-10%.

1.2 Data Structure

1.2.1 Bit Definitions

DALI communication uses Manchester code. Therefore, bits are defined as 0 at falling edges and as 1 at rising edges and fixed to high level when no communication is performed.

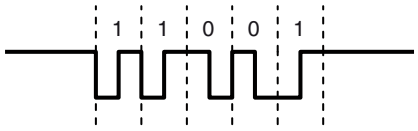


Figure 1-1 Bit Definitions

1.2.2 Frames

1.2.2.1 Forward Frame

A forward frame is a frame transmitted from the master to a slave. Such a frame has 19 bits.

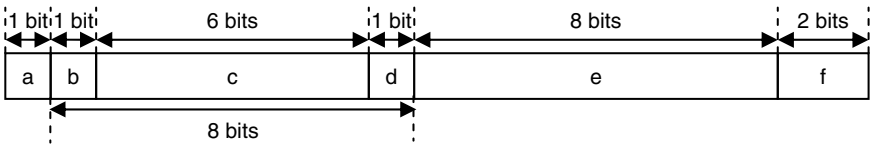


Figure 1-2 Structure of a Forward Frame

a: Start bit
This bit indicates the start of the frame.
The waveform of this bit is always equal to 1.
b to d: Address byte
This byte specifies where to transmit the frame.
e: Data byte
This byte specifies a command.
f: Stop bits
These bits indicate the end of the frame. These bits are fixed to high level.

1.2.2.2 Backward Frame

A backward frame is a frame transmitted from a slave to the master. Such a frame has 11 bits.

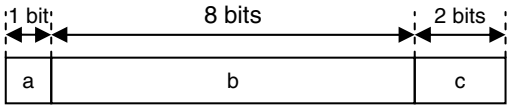


Figure 1-3 Structure of a Backward Frame

a: Start bit
This bit indicates the start of the frame. The waveform of this bit is always equal to 1.
b: Data byte
This byte replies to the master.
c: Stop bits
These bits indicate the end of the frame. These bits are fixed to high level.

1.3 Specifying the Transmission and Reception Timing

1.3.1 Timing Within a Frame

1 bit width in DALI is 833.3 us +/-10% for both forward and backward frames.

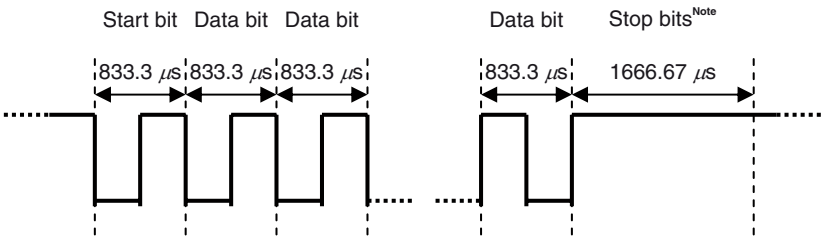


Figure 1-4 Timing within a Frame

Note Because there are two stop bits, their timing is 1666.67 us.

1.3.2 Timing Between Frames

The following timing control is required in frame units in DALI:

- Forward frame width: 15.83 ms +/-10%
- Backward frame width: 9.17 ms +/-10%
- Communication interval between a forward and backward frame: 2.92 to 9.17 ms
- Interval between a forward frame and the next forward frame: At least 9.17 ms
- Interval between a backward frame and the next forward frame: At least 9.17 ms

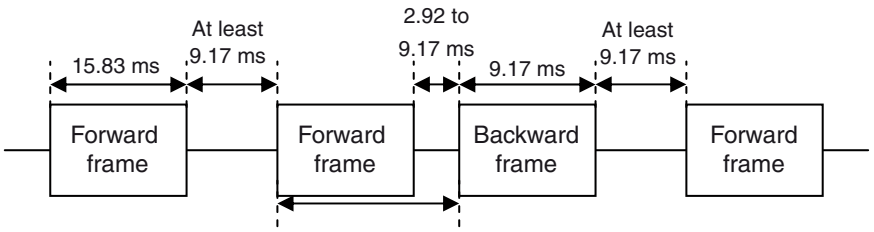


Figure 1-5 Timing Between Frames

Chapter 2 DALI Implementation Using 78K0/Ix2

2.1 DALI protocol communication control interface

The 78K0/IA2 and 78K0/IB2 microcontrollers are provided with the serial interface UART6/DALI to transmit and receive data as a slave of DALI communication by using hardware. Therefore, no CPU load due to software processing occurs when data is transmitted or received via DALI.

Figure 2-1 shows a schematic example of a DALI communication circuit. Pins for DALI reception input (RxD6 pin) and DALI transmission output (TxD6 pin) are required for performing DALI communication.

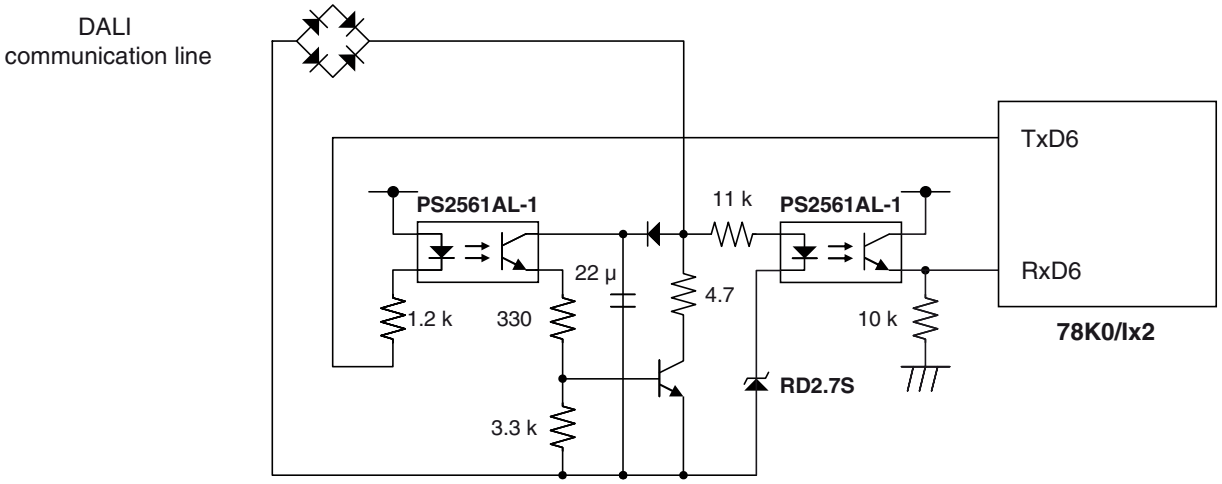


Figure 2-1 Schematic Example of a DALI Communication Circuit

Remarks:
1. PS2561AL-1 is a photocoupler made by NEC Electronics.
2. RD2.7S is a Zener diode made by NEC Electronics.

2.1.1 Initialization

An example of initializing the serial interface UART6/DALI when performing DALI communication is shown below.

```
UADLSEL = 1;    /* Specifies DALI mode. */
ASICL6.1 = 0;   /* Specifies that the MSB is first. */
CKSR6 = 5;      /* Sets the count clock to 625 kHzNote 1. */
BRGC6 = 130;    /* Sets the baud rate to 1,200 bpsNote1 */
ASIM6.0 = 0;    /* Specifies INTSRE6 Note2 as the interrupt
                   generated when a reception error occurs.*/
```

Notes
1. This applies if PLL clock mode is used and 20 MHz is specified for the clock supplied to the peripheral hardware units.
2. The interrupt generated when data is received successfully (INTSR6) can also be specified.

2.1.2 Timing Chart When Receiving Data

Figure 2-2 shows an example of the timing chart when data is received from the master during DALI communication.

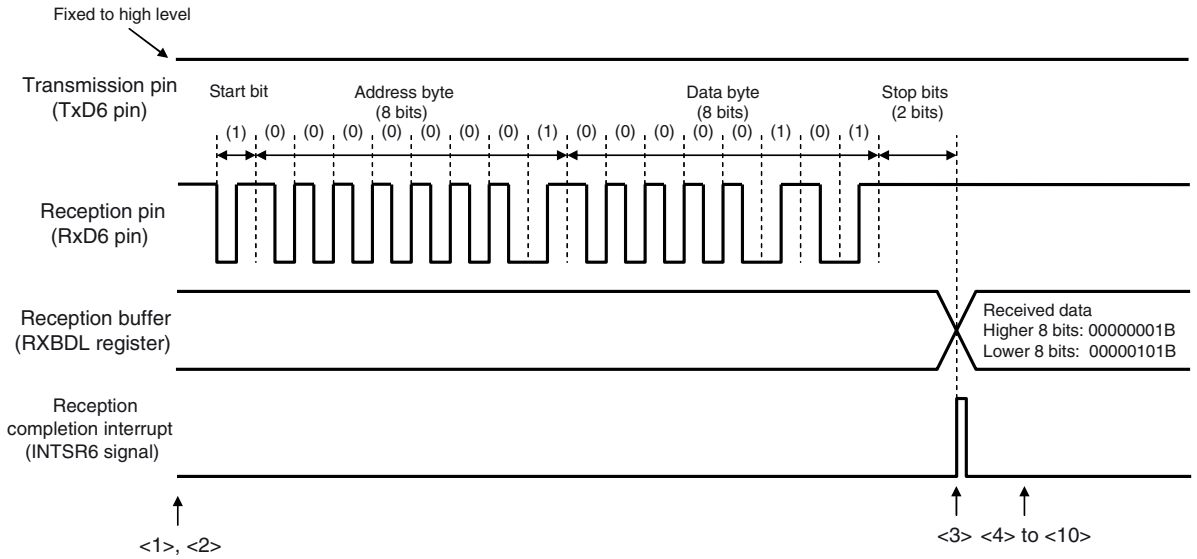


Figure 2-2 Timing Chart When Receiving Data During DALI Communication

The operations are summarized below:

<Enabling reception>

1. The internal operation clock of the serial interface UART6/DALI is enabled.
2. Reception is enabled and a status in which reception can be performed is specified.
3. If data is received successfully from the master during DALI communication, the interrupt INTSR6 is generated and received data is stored into the RXBDL register. If a reception error occurs, an INTSRE6 interrupt is generated and the reception error status is stored into the ASIS6 register. (Figure 7 shows a timing chart for when data was received successfully.)

<Processing after an INTSR6 interrupt is generated when data was received successfully>

4. The received data (16 bits) is saved.
5. Reception is disabled.
6. The internal operation clock of the serial interface UART6/DALI is disabled.

<Processing after an INTSRE6 interrupt is generated when a reception error occurred>

7. The reception error status is read.
8. The received data (16 bits) is discarded.
9. Reception is disabled.
10. The internal operation clock of the serial interface UART6/DALI is disabled.

Sample code for operations <1>, <2>, and <4> to <10> is shown below:

<Enabling reception>

```
POWER6 = 1;   /* <1> Enables the internal operation clock. */
RXE6 = 1;     /* <2> Enables reception. */
```

<Processing after an INTSR6 interrupt is generated when data was received successfully>

```
ushRxCode = RXBDL;  /* <4> Saves the received dataNote 1. */
RXE6 = 0;           /* <5> Disables reception. */
POWER6 = 0;         /* <6> Disables the internal operation clock. */
```

<Processing after an INTSRE6 interrupt is generated when a reception error occurred>

```
ucRxeCode = ASIS6; /* <7> Reads the reception error statusNote 2. */
ushTemp = RXBDL;   /* <8> Discards the received data Note1 */
RXE6 = 0;          /* <9> Disables reception. */
POWER6 = 0;        /* <10> Disables the internal operation clock. */
```

Notes
1. ushRxCode and ushTemp are 16-bit variables.
2. ucRxeCode is an 8-bit variable.

2.1.3 Timing Chart When Transmitting Data

Figure 2-3 shows an example of the timing chart when data is transmitted to the master during DALI communication.

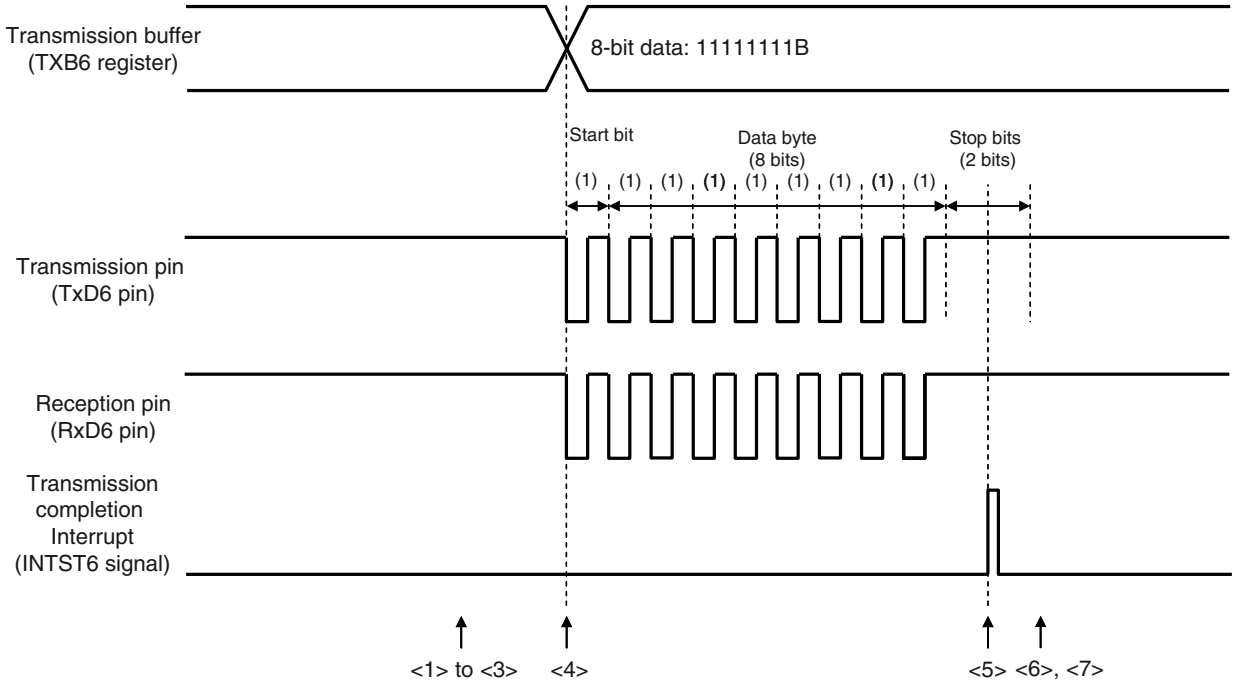


Figure 2-3 Chart When Transmitting Data during DALI Communication

The operations are summarized below:

<Starting transmission>

1. The internal operation clock of the serial interface UART6/DALI is enabled.
2. Transmission is enabled and a status in which reception can be performed is specified.
3. An NOP instruction (which consumes two clocks) is executed so that the system waits at least one base clock cycle.
4. The data to transmit is stored into the TXB6 register and transmission starts Note.
5. An INTST6 interrupt is generated when transmission ends.

<Processing after an INTST6 interrupt is generated when transmission ends>

6. Transmission is disabled.
7. The internal operation clock of the serial interface UART6/DALI is disabled.

Note As shown in *Figure 2-3* , the transmitted data is directly input to the reception pin during DALI communication. Therefore, preventing the generation of reception error interrupts is recommended by disabling reception (RXE6 = 0) before starting transmission.

Sample code for operations <1> to <4>, <6>, and <7> is shown below:

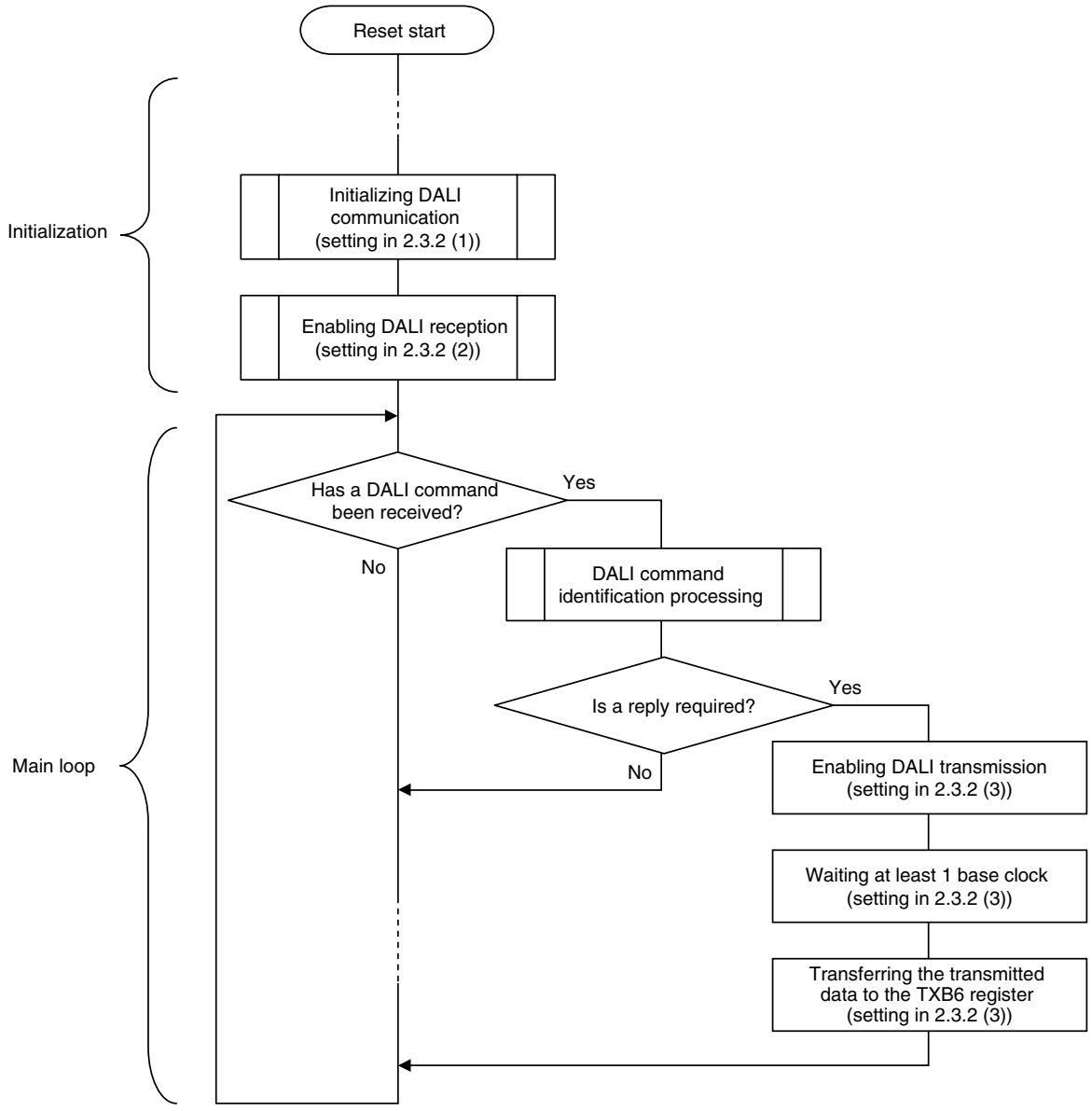
<Starting transmission>

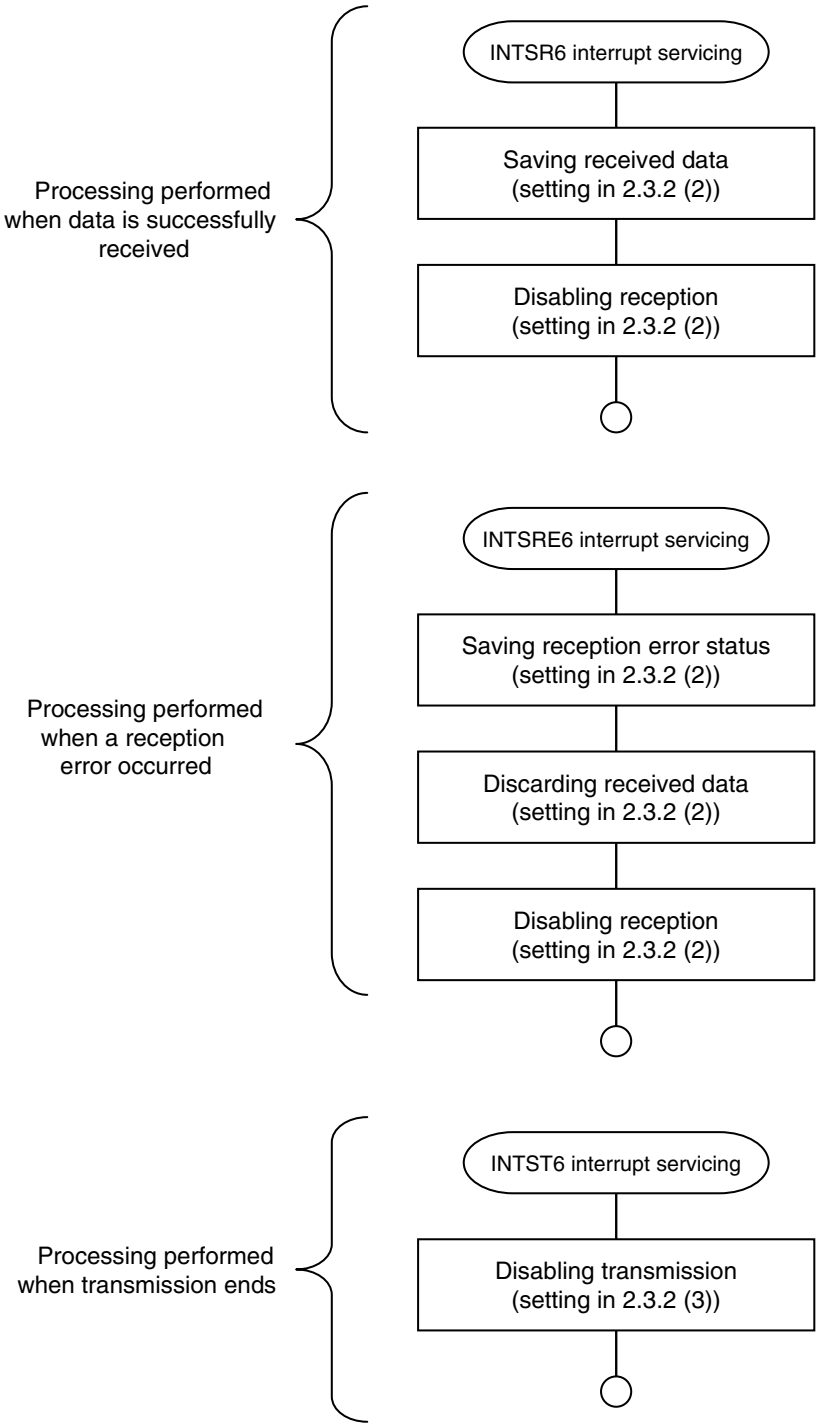
```
POWER6 = 1;          /* <1> Enables the internal operation clock. */
TXE6 = 1;            /* <2> Enables transmission. */
NOP();               /* <3> Waits at least 1 base clock. */
TXB6 = 0b11111111;   /* <4> Starts transmitting the data 11111111. */
```

<Processing after an INTST6 interrupt is generated when transmission ends>

```
TXE6 = 0;         /* <6> Disables transmission. */
POWER6 = 0;       /* <7> Disables the internal operation clock. */
```

2.2 Software Flowcharts for DALI Communication




Processing performed when data is successfully received
INTSR6 interrupt servicing
Saving received data (setting in 2.3.2 (2))
Disabling reception (setting in 2.3.2 (2))
Processing performed when a reception error occurred
INTSRE6 interrupt servicing
Saving reception error status (setting in 2.3.2 (2))
Discarding received data (setting in 2.3.2 (2))
Disabling reception (setting in 2.3.2 (2))
Processing performed when transmission ends
INTST6 interrupt servicing
Disabling transmission (setting in 2.3.2 (3))

Chapter 3 Appendix: Code Examples

3.1 Initialize DALI

```
void DALI_init( void )
{
    /* The timer to control the communication interval is initialized.  */
    daliCommunicationIntervalInit();

    /* The timer to control the fade is initialized. */
    daliFadeTimerInit();

    /* The ballast data is initialized. */
    daliBallastData_init();

    /* The port used by DALI is initialized. */
    DALI_port_init();
}
```

3.2 Start DALI Port

```
void DALI_port_init( void )
{
    /* The port used by DALI is initialized. */
    PORT_MODE_TXD6 = OUTPUT;
    PORT_TXD6 = LEVEL_HIGH;
    PORT_MODE_RXD6 = INPUT;
    UADLCTL = UADLCTL_INIT_VALUE;
    ASICL6_bit.no1 = BIT_CLR;
    ASIM6 = ASIM6_INIT_VALUE;
    CKSR6 = CKSR6_INIT_VALUE;
    BRGC6 = BRGC6_INIT_VALUE;
    SRMK6 = INTERRUPT_UNMASKED;
    SRPR6 = PRIORITY_HIGH;
    POWER6 = BIT_SET;
    TXE6 = BIT_SET;
    RXE6 = BIT_SET;
}
```

3.3 Initialize Ballast Data

```
void daliBallastData_init( void )
{
    unsigned char ucSceneCount;
    unsigned char *ptr;
    unsigned char ucRetValue = 255;

    /* Get the physical address of the memory. */
    ptr = (unsigned char *)(MEMORY_ADDRESS);

    /* Whether it was started two times or more is checked.  */
    if( *(ptr + INIT_FLAG) != STARTED_TWO_TIME )/* It was started for the first time.  */
    {
        /* Value set with GUI is set to the ballast data.  */
        g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL]          = DEFAULT_POWER_ON_LEVEL;
```

```
    g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL]    = DEFAULT_SYSTEM_FAILURE_LEVEL;
    g_ucBallastData[BALLAST_DATA_MIN_LEVEL]               = DEFAULT_MIN_LEVEL;
    g_ucBallastData[BALLAST_DATA_MAX_LEVEL]               = DEFAULT_MAX_LEVEL;
    g_ucBallastData[BALLAST_DATA_FADE_RATE]               = DEFAULT_FADE_RATE;
    g_ucBallastData[BALLAST_DATA_FADE_TIME]               = DEFAULT_FADE_TIME;
    g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS]           = DEFAULT_SHORT_ADDRESS;
    g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_H]        = DEFAULT_RANDOM_ADDRESS_H;
    g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_M]        = DEFAULT_RANDOM_ADDRESS_M;
    g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_L]        = DEFAULT_RANDOM_ADDRESS_L;
    g_ucBallastData[BALLAST_DATA_GROUP_0_7]               = DEFAULT_GROUP_0_7;
    g_ucBallastData[BALLAST_DATA_GROUP_8_15]              = DEFAULT_GROUP_8_15;

  for(ucSceneCount = BALLAST_DATA_SCENE_0; ucSceneCount<= BALLAST_DATA_SCENE_15; ucSceneCount++ )
    {
        g_ucBallastData[ucSceneCount]                     = DEFAULT_SCENE;
    }

    g_ucBallastData[INIT_FLAG]                            = STARTED_TWO_TIME;
    g_ucBallastData[BLANK_SPACE]                          = 0xFF;
    g_ucBallastData[BLANK_SPACE+1]                        = 0xFF;
    g_ucBallastData[BLANK_SPACE+2]                        = 0xFF;

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
}
else /* It was started two times or more.  */
{
    /* The data written in the memory is set to the ballast data.  */
    g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL]
                                      = *(ptr + BALLAST_DATA_POWER_ON_LEVEL);
    g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL]
                                      = *(ptr + BALLAST_DATA_SYSTEM_FAILURE_LEVEL);
    g_ucBallastData[BALLAST_DATA_MIN_LEVEL]
                                      = *(ptr + BALLAST_DATA_MIN_LEVEL);
    g_ucBallastData[BALLAST_DATA_MAX_LEVEL]
                                      = *(ptr + BALLAST_DATA_MAX_LEVEL);
    g_ucBallastData[BALLAST_DATA_FADE_RATE]
                                      = *(ptr + BALLAST_DATA_FADE_RATE);
    g_ucBallastData[BALLAST_DATA_FADE_TIME]
                                      = *(ptr + BALLAST_DATA_FADE_TIME);
    g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS]
                                      = *(ptr + BALLAST_DATA_SHORT_ADDRESS);
    g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_H]
                                     = *(ptr + BALLAST_DATA_RANDOM_ADDRESS_H);
    g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_M]
                                      = *(ptr + BALLAST_DATA_RANDOM_ADDRESS_M);
    g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_L]
                                      = *(ptr + BALLAST_DATA_RANDOM_ADDRESS_L);
    g_ucBallastData[BALLAST_DATA_GROUP_0_7]
                                      = *(ptr + BALLAST_DATA_GROUP_0_7);
    g_ucBallastData[BALLAST_DATA_GROUP_8_15]
                                      = *(ptr + BALLAST_DATA_GROUP_8_15);
    for( ucSceneCount = BALLAST_DATA_SCENE_0; ucSceneCount
                                      <= BALLAST_DATA_SCENE_15; ucSceneCount++ )
    {
        g_ucBallastData[ucSceneCount]                     = *(ptr + ucSceneCount);
    }

    ucRetValue = WRITE_OK;
}

STATUS_INFO_ALL = DEFAULT_STATUS_INFORMATION;

/* When writing in the memory has failed,
   the ballast actual level is set to System failure level. */
if( ucRetValue != WRITE_OK )
{
    daliLightingCommandSystemFailure();
}

/* It is checked whether a power on level is in the useful range. */
if( (g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL] > 0x00) &&
    (g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL] <
```

```
         g_ucBallastData[BALLAST_DATA_MIN_LEVEL]) )
    {
        g_ucActualLevel = g_ucBallastData[BALLAST_DATA_MIN_LEVEL];
        STATUS_INFO_LIMIT_ERROR = BIT_SET;
    }
    else if( (g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL] >
              g_ucBallastData[BALLAST_DATA_MAX_LEVEL]) &&
             (g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL] < 0xFF) )
    {
        g_ucActualLevel = g_ucBallastData[BALLAST_DATA_MAX_LEVEL];
        STATUS_INFO_LIMIT_ERROR = BIT_SET;
    }
    else if( (g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL] == 0x00) ||
            ((g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL]
             >= g_ucBallastData[BALLAST_DATA_MIN_LEVEL]) &&
            (g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL] <=
             g_ucBallastData[BALLAST_DATA_MAX_LEVEL])) )
    {
        g_ucActualLevel = g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL];
    }

    g_ucSearchAddressH = DEFAULT_SEARCHADDRESSH;
    g_ucSearchAddressM = DEFAULT_SEARCHADDRESSM;
    g_ucSearchAddressL = DEFAULT_SEARCHADDRESSL;
    g_ucDTR = DEFAULT_DTR;

    if( g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS] == SHORT_ADDRESS_INVALID )
    {
        STATUS_INFO_MISSING_SHORT_ADDRESS = BIT_CLR;
    }

}
```

3.4 Initialize DALI Communication Interval

```
void daliCommunicationIntervalInit( void )
{
    /* The timer to control the communication interval is initialized.  */
    TCE51 = BIT_CLR;
    CR51 = CR51_INIT_VALUE;
    TCL51 = TCL51_INIT_VALUE;
    TMIF51 = BIT_CLR;
    TMMK51 = INTERRUPT_UNMASKED;
    TMPR51 = PRIORITY_HIGH;
    TCE51 = BIT_SET;
    g_ucBackwardSendCount = 0;
    g_ucReceiveContinueCount = 0;
    g_ulAddressSettingCount = 0;
    g_ucBackwardSendFlag = FLAG_DOWN;
    g_ucReceiveContinueFlag = FLAG_DOWN;
    g_ucAddressSettingFlag = FLAG_DOWN;
    g_ucReceiveFlag = FLAG_UP;
}
```

3.5 Initialize DALI Fade Timer

```
void daliFadeTimerInit( void )
{
    /* The timer for the fade control is initialized. */
    TMC00 = TMC00_INIT_VALUE;
    CRC00 = CRC00_INIT_VALUE;
}
```

3.6 Start DALI Fade Timer

```
void daliFadeTimerStart( unsigned char ucFadeLevel,unsigned long ulTotalFadeTime )
{
    unsigned long ulOneLevelFadeTime;
    unsigned long ulCalcWork;
    unsigned long ulRoundWork;
    unsigned char ucPRM;
    unsigned short ushDivide[3] = {1, 4, 256};

    /* The value of CR000 and PRM00 is calculated at the fade level and the fade time. */
    ulOneLevelFadeTime = (ulTotalFadeTime / ucFadeLevel);
    if( ulOneLevelFadeTime < 838000 )
    {
        g_usFadeTimerCount = 1;
        g_usFadeTimerCountBackup = 1;
    }
    else
    {
        g_usFadeTimerCount = (unsigned short)(ulOneLevelFadeTime / 1000);
        g_usFadeTimerCountBackup = g_usFadeTimerCount;
        ulOneLevelFadeTime = 1000;
    }

    for( ucPRM = 0; ucPRM < 3; ucPRM++ )
    {
        ulCalcWork = (ulOneLevelFadeTime * CLOCK_FREQ);
        ulRoundWork = (ulCalcWork % ushDivide[ucPRM]);
        ulCalcWork = (ulCalcWork / ushDivide[ucPRM]);

        if( (ulRoundWork * 2) < ushDivide[ucPRM] )
        {
            ulCalcWork--;
        }

        if( ulCalcWork <= 0xFFFF )
        {
            break;
        }
    }

    /* The timer for the fade control is started. */
    CR000 = (unsigned short)ulCalcWork;
    PRM00 = ucPRM;
    STATUS_INFO_FADE_READY = BIT_SET;
    TMIF000 = FLAG_DOWN;
    TMMK000 = INTERRUPT_UNMASKED;
    TMPR000 = PRIORITY_HIGH;
    TMC00 = FADE_START;
}
```

3.7 Stop DALI Fade Timer

```
void daliFadeTimerStop( unsigned char ucFlag )
{
    /* The timer for the fade control is stoped. */
    TMMK000 = INTERRUPT_MASKED;
    TMC00 = FADE_STOP;
    g_usFadeTimerCount = 0;
    g_usFadeTimerCountBackup = 0;
    if(ucFlag == FLAG_DOWN) {
        g_ucFadeFlag = FLAG_DOWN;
    }
    STATUS_INFO_FADE_READY = BIT_CLR;
```

```
        g_ucFadeCount = 0;
    }
```

3.8 Set DALI Communication Interval

```
void daliCommunicationIntervalSet( unsigned long ulSetValue )
{
    /* The control count of the communication interval begins. */
    switch( ulSetValue )
    {
        /* The count at backword transmission interval is started. */
        case BACKWARD_SEND_INTERBAL:
            g_ucBackwardSendCount = (unsigned char)ulSetValue;
            g_ucBackwardSendFlag = FLAG_DOWN;
            break;

        /* The count at continuous reception interval is started. */
        case RECEIVE_CONTINUE_INTERVAL:
            g_ucReceiveContinueCount = (unsigned char)ulSetValue;
            g_ucReceiveContinueFlag = FLAG_UP;
            break;

        /* The count at address setting period is started. */
        case ADDRESS_SETTING_INTERVAL:
            g_ulAddressSettingCount = ulSetValue;
            g_ucAddressSettingFlag = FLAG_UP;
            break;

        default:
            break;
    }
}
```

3.9 Send DALI Backward Interval Stop

```
void daliBackwardSendIntervalStop( void )
{
    /* The count at backword transmission interval is stopped. */
    g_ucBackwardSendCount = 0;
    g_ucBackwardSendFlag = FLAG_UP;
}
```

3.10 Receive DALI Interval Stop

```
void daliReceiveIntervalStop( void )
{
    /* The count at continuous reception interval is stopped. */
    g_ucReceiveContinueCount = 0;
    g_ucReceiveContinueFlag = FLAG_DOWN;
}
```

3.11 Stop Setting DALI Address Interval

```
void daliAddressSettingIntervalStop( void )
{
    /* The count at address setting period is stopped. */
    g_ulAddressSettingCount = 0;
    g_ucAddressSettingFlag = FLAG_DOWN;
    g_ucPhysicalSelection = FLAG_DOWN;
}
```

3.12 Receive a Byte Through DALI

```
unsigned short DALI_getValue( void )
{
    unsigned char ucAddressKind = FALSE;
    unsigned char ucCommandInfo = 255;
    unsigned char ucDutyElement = 0;

    /* Check on system malfunction */
    if( STATUS_INFO_LAMP_FAILURE == BIT_CLR )
    {

        /* Command analysis processing is done
           when there is Forward reception completion notification. */
        if( g_ucReceiveCompletion )
        {
            g_ucReceiveCompletion = FLAG_DOWN;
            ucAddressKind = daliAnalyzeAddress( g_ucAddressByte, &ucCommandInfo );
            switch( ucAddressKind )
            {
                case ADDRESS_KIND_MY_ADDRESS:
                case ADDRESS_KIND_MY_GROUP:
                case ADDRESS_KIND_BROADCAST:
                    daliAnalyzeCommand( g_ucCommandByte, ucCommandInfo );
                    break;
                case ADDRESS_KIND_EX_COMMAND:
                    daliAnalyzeExCommand( g_ucAddressByte, g_ucCommandByte );
                    break;
                default:
                    break;
            }
            g_ucReceiveFlag = FLAG_UP;
        }

        /* The fade is processed. */
        if( g_ucFadeFlag )
        {
            g_ucFadeFlag = FLAG_DOWN;
            switch( g_ucFadeDirection )
            {
                case FADE_UP:/* fade up */
                    if( g_ucActualLevel >= g_ucBallastData[BALLAST_DATA_MIN_LEVEL] )
                    {
                        g_ucActualLevel++;
                    }
                    else
                    {
                        g_ucActualLevel = g_ucBallastData[BALLAST_DATA_MIN_LEVEL];
                    }

                    if( g_ucActualLevel >= g_ucBallastData[BALLAST_DATA_MAX_LEVEL] )
                    {
                        daliFadeTimerStop(FLAG_DOWN);
                        g_ucActualLevel = g_ucBallastData[BALLAST_DATA_MAX_LEVEL];
```

```
                    }
                    break;
                case FADE_DOWN:/* fade down */
                    if( (g_ucActualLevel <= g_ucBallastData[BALLAST_DATA_MIN_LEVEL]) ||
                        (g_ucActualLevel == 255) )
                    {
                        daliFadeTimerStop(FLAG_DOWN);
                        if( g_ucCommandBackup != COMMAND_DOWN )
                        {
                            g_ucActualLevel = 0;
                        }
                    }
                    else
                    {
                        g_ucActualLevel--;
                    }

                    break;
                default:
                    break;
            }
        }

        /* Backward transmission is done
           when there is Backward transmission request. */
        if( g_ucBackwardSendFlag )
        {
            g_ucBackwardSendFlag = FLAG_DOWN;
            DALI_transmission(g_ucSendData);
        }
    }
    else
    {
        /* The ballast actual level is set to System failure level
           when there is a system malfunction. */
        daliLightingCommandSystemFailure();
    }

    /* return the conversion value */
    ucDutyElement = g_ucActualLevel;

    return (unsigned short)g_ucConvertValue[ucDutyElement];
}
```

3.13 DALI Receive Interrupt

```
__interrupt void DALI_receive_interrupt( void )
{
    unsigned short ushReservedData;

    /* Forward is received. */
    if( g_ucReceiveFlag )
    {
        g_ucReceiveFlag = FLAG_DOWN;
        ushReservedData = RXBDL;
       g_ucAddressByte = (unsigned char)((ushReservedData >> 8) & 0x00FF);
        g_ucCommandByte = (unsigned char)(ushReservedData & 0x00FF);
        g_ucReceiveCompletion = FLAG_UP;
    }
}
```

3.14 DALI Communication Interval Interrupt

```
__interrupt void daliCommunicationIntervalInterrupt( void )
{
    /* The count at backword transmission interval is stopped. */
    if( g_ucBackwardSendCount != 0 )
    {
        g_ucBackwardSendCount--;
        if( g_ucBackwardSendCount == 0 )
        {
            daliBackwardSendIntervalStop();
        }
    }

    /* The count at continuous reception interval is stopped. */
    if( g_ucReceiveContinueCount != 0 )
    {
        g_ucReceiveContinueCount--;
        if( g_ucReceiveContinueCount == 0 )
        {
            daliReceiveIntervalStop();
        }
    }

    /* The count at address setting period is stopped. */
    if( g_ulAddressSettingCount != 0 )
    {
        g_ulAddressSettingCount--;
        if( g_ulAddressSettingCount == 0 )
        {
            daliAddressSettingIntervalStop();
        }
    }

}
```

3.15 DALI Fade Time Interrupt

```
__interrupt void daliFadeTimeInterrupt( void )
{
    /* The fade timer is interrupted. */
    if( g_usFadeTimerCount== 0 )
    {
        return;
    }

    g_usFadeTimerCount--;

    if( g_usFadeTimerCount != 0 )
    {
        return;
    }

    g_ucFadeFlag = FLAG_UP;
    g_ucFadeCount--;

    if( g_ucFadeCount == 0 )
    {
        g_usFadeTimerCountBackup= 0;
        daliFadeTimerStop(FLAG_UP);
    }
    else
    {
        g_usFadeTimerCount = g_usFadeTimerCountBackup;
```

```
        }
    }
```

3.16 DALI Transmission

```
void DALI_transmission( unsigned char ucSendData )
{
    /* Backward is transmitted. */
    while( (ASIF6 & 0x02) != 0 )
    {
        ;
    }
    TXB6 = ucSendData;
}
```

3.17 Check Ballast Data is Cleared

```
unsigned char daliBallastData_ResetValueCheck( void )
{
    unsigned char ucSceneCount;

    /* It checks whether the ballast data is equal to the reset value.  */
    if( g_ucBallastData[BALLAST_DATA_MIN_LEVEL] != RESET_MINLEVEL )
    {
        return FALSE;
    }
    if( g_ucBallastData[BALLAST_DATA_MAX_LEVEL] != RESET_MAXLEVEL )
    {
        return FALSE;
    }
    if( g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL] != RESET_POWERONLEVEL )
    {
        return FALSE;
    }
    if( g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL] != RESET_SYSTEMFAILURELEVEL )
    {
        return FALSE;
    }
    if( g_ucBallastData[BALLAST_DATA_FADE_RATE] != RESET_FADERATE )
    {
        return FALSE;
    }
    if( g_ucBallastData[BALLAST_DATA_FADE_TIME] != RESET_FADETIME )
    {
        return FALSE;
    }
    if( g_ucSearchAddressH != RESET_SEARCHADDRESSH )
    {
        return FALSE;
    }
    if( g_ucSearchAddressM != RESET_SEARCHADDRESSM )
    {
        return FALSE;
    }
    if( g_ucSearchAddressL != RESET_SEARCHADDRESSL )
    {
        return FALSE;
    }
    if( g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_H] != RESET_RANDOMADDRESSH )
    {
        return FALSE;
    }
    if( g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_M] != RESET_RANDOMADDRESSM )
```

```
    {
        return FALSE;
    }
    if( g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_L] != RESET_RANDOMADDRESSH )
    {
        return FALSE;
    }
    if( g_ucBallastData[BALLAST_DATA_GROUP_0_7] != RESET_GROUP_0_7 )
    {
        return FALSE;
    }
    if( g_ucBallastData[BALLAST_DATA_GROUP_8_15] != RESET_GROUP_8_15 )
    {
        return FALSE;
    }
    for( ucSceneCount = BALLAST_DATA_SCENE_0 ; ucSceneCount <= BALLAST_DATA_SCENE_15 ;
         ucSceneCount++ )
    {
        if( g_ucBallastData[ucSceneCount] != RESET_SCENE )
        {
            return FALSE;
        }
    }
    if( g_ucActualLevel != RESET_ACTUAL_LEVEL )
    {
        return FALSE;
    }

    return TRUE;
}
```

3.18 Analyze DALI Address

```
unsigned char daliAnalyzeAddress( unsigned char  ucAddressByte,unsigned char *p_ucCommandInfo )
{
    unsigned char ucAddressKind = ADDRESS_KIND_OTHERS;    /* Address Kind */
    unsigned char ucAddress;                              /* Address */
    unsigned char ucIsBelong;                             /* Belonging flag */

    /* The Address Byte of receive data is analyzed.  */
    switch( ucAddressByte & EX_COMMAND_MASK )
    {
        case EX_COMMAND_PATTERN_1:/* Extension command */
        case EX_COMMAND_PATTERN_2:
            ucAddressKind = ADDRESS_KIND_EX_COMMAND;
            *p_ucCommandInfo = ucAddressByte;
            break;

        case BROADCAST_PATTERN:/* Broadcast */
            if( (ucAddressByte & BROADCAST_BIT) == BROADCAST_BIT )
            {
                ucAddressKind = ADDRESS_KIND_BROADCAST;
                *p_ucCommandInfo = (ucAddressByte & SELECTOR_BIT);
            }
            break;

        case GROUP_ADDRESS_PATTERN:/* Group */
            ucAddress = ((ucAddressByte >> 1) & ADDRESS_BIT);
            ucIsBelong = daliConformBelongGroup( ucAddress );
            if( ucIsBelong ){
                ucAddressKind = ADDRESS_KIND_MY_GROUP;
                *p_ucCommandInfo = ucAddressByte & SELECTOR_BIT;
            }
            break;

        default:/* Short Address */
            ucAddress = ((ucAddressByte >> 1) & ADDRESS_BIT);
            if( ucAddress == g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS] )
```

```
            {
                ucAddressKind = ADDRESS_KIND_MY_ADDRESS;
                *p_ucCommandInfo = (ucAddressByte & SELECTOR_BIT);
            }
            break;

    }

    return ucAddressKind;
}
```

3.19 Analyze DALI Command

```
void daliAnalyzeCommand( unsigned char ucCommandByte,unsigned char ucSelectBit )
{
    unsigned char ucRetValue;

    /* The Command Byte of receive data is analyzed. */
    switch( ucSelectBit )
    {
        case COMMAND_MODE_DIRECT:/* MODE : Direct arc power control command */
            /* The acceptance of continuous reception command is checked.   */
            ucRetValue = daliReceiveContinueCommandCheck( g_ucAddressByte );
            if( ucRetValue )
            {
                /* DIRECT ARC POWER CONTROL */
                daliLightingCommandDirectArcPowerControl( ucCommandByte );
            }
            break;

        case COMMAND_MODE_COMMAND:/* MODE : Command mode */
            /* The acceptance of continuous reception command is checked.  */
            ucRetValue = daliReceiveContinueSettingCommandCheck( ucCommandByte );
            if( ucRetValue )
            {
                /* Distribution of Lighting command */
                if( ucCommandByte < COMMAND_RESET )
                {
                    daliDistributeLightingCommand( ucCommandByte );
                }
                /* Distribution of Setting command */
             else if( (ucCommandByte >= COMMAND_RESET) && (ucCommandByte < COMMAND_QUERY_STATUS) )
                {
                    daliDistributeSettingCommand( ucCommandByte );
                }
                /* Distribution of Query command */
                else if( ucCommandByte >= COMMAND_QUERY_STATUS )
                {
                    daliDistributeQueryCommand( ucCommandByte );
                }
            }
            break;

        default:
            break;

    }

}

void daliAnalyzeExCommand( unsigned char ucCommandByte,unsigned char ucAddressByte )
{
    unsigned char ucRetValue;

    /* The acceptance of continuous reception command is checked.  */
    ucRetValue = daliReceiveContinueCommandCheck( ucCommandByte );
    if( !ucRetValue )
    {
```

```
        return;
    }

    /* Distribution of Extension command */
    switch( ucCommandByte )
    {
        case EXCOMMAND_TERMINATE:/* TERMINATE command */
            daliExCommandTerminate();
            break;

        case EXCOMMAND_DTR:/* DTR command */
            daliExCommandDTR( ucAddressByte );
            break;

        case EXCOMMAND_INITIALIZE:/* INITIALIZE command */
            daliExCommandInitialize( ucAddressByte );
            break;

        case EXCOMMAND_RANDOMISE:/* RANDOMISE command */
            daliExCommandRandomise();
            break;

        case EXCOMMAND_COMPARE:/* COMPARE command */
            daliExCommandCompare();
            break;

        case EXCOMMAND_WITHRAW:/* WITHDRAW command */
            daliExCommandWithdraw();
            break;

        case EXCOMMAND_SEARCHADDRH:/* SEARCHADDRH command */
            daliExCommandSearchAddrH( ucAddressByte );
            break;

        case EXCOMMAND_SEARCHADDRM:/* SEARCHADDRM command */
            daliExCommandSearchAddrM( ucAddressByte );
            break;

        case EXCOMMAND_SEARCHADDRL:/* SEARCHADDRL command */
            daliExCommandSearchAddrL( ucAddressByte );
            break;

        case EXCOMMAND_PROGRAM_SHORT_ADDRESS:/* PROGRAM SHORT ADDRESS command */
            if( ucAddressByte != 255 )
            {
                daliExCommandProgramShortAddress( ucAddressByte );
            }
            else
            {
                daliExCommandDeleteTheShortAddress();
            }
            break;

        case EXCOMMAND_VERIFY_SHORT_ADDRESS:/* VERIFY SHORT ADDRESS command */
            daliExCommandVerifyShortAddress(ucAddressByte);
            break;

        case EXCOMMAND_QUERY_SHORT_ADDRESS:/* QUERY SHORT ADDRESS command */
            daliExCommandQueryShortAddress();
            break;

        case EXCOMMAND_PHYSICAL_SELECTION:/* PHYSICAL SELECTION command */
            daliExCommandPhysicalSelection();
            break;

        case EXCOMMAND_ENABLE_DEVICE_TYPE_X:/* ENABLE DEVICE TYPE X command */
            break;

        default:
            break;

    }
}
```

3.20 Check DALI Groups

```
unsigned char daliConformBelongGroup( unsigned char ucGroupAddress )
{
    unsigned char ucBelongData;
    unsigned char ucIsBelong;

    /* It confirms whether to belong to the group. */
    if( ucGroupAddress < 8 )
    {
        ucBelongData = g_ucBallastData[BALLAST_DATA_GROUP_0_7];
    }
    else
    {
        ucBelongData = g_ucBallastData[BALLAST_DATA_GROUP_8_15];
        ucGroupAddress = (ucGroupAddress - 8);
    }

    ucIsBelong = ((ucBelongData >> ucGroupAddress) & 0x01);

    return ucIsBelong;
}
```

3.21 Distribute DALI Command

```
void daliDistributeLightingCommand( unsigned char ucCommand )
{
    /* Distribution of Lighting command */
    switch( ucCommand )
    {
        case COMMAND_OFF:/* OFF */
            daliLightingCommandOff();
            break;

        case COMMAND_UP:/* UP */
            daliLightingCommandUp();
            break;

        case COMMAND_DOWN:/* DOWN */
            daliLightingCommandDown();
            break;

        case COMMAND_STEP_UP:/* STEP UP */
            daliLightingCommandStepUp();
            break;

        case COMMAND_STEP_DOWN:/* STEP DOWN */
            daliLightingCommandStepDown();
            break;

        case COMMAND_RECALL_MAX_LEVEL:/* RECALL MAX LEVEL */
            daliLightingCommandRecallMaxLevel();
            break;

        case COMMAND_RECALL_MIN_LEVEL:/* RECALL MIN LEVEL */
            daliLightingCommandRecallMinLevel();
            break;

        case COMMAND_STEP_DOWN_AND_OFF:/* STEP DOWN AND OFF */
            daliLightingCommandStepDownAndOff();
            break;
```

```
        case COMMAND_ON_AND_STEP_UP:/* ON AND STEP UP */
            daliLightingCommandOnAndStepUp();
            break;

        default:
            /* GO TO SCENE */
            if( (ucCommand>= COMMAND_GO_TO_SCENE) && (ucCommand<= (COMMAND_GO_TO_SCENE+15)) )
            {
                daliLightingCommandGoToScene( ucCommand );
            }
            break;


    }
}

void daliDistributeSettingCommand( unsigned char ucCommand )
{
    /* Distribution of Setting command */
    switch( ucCommand )
    {
        case COMMAND_RESET:/* RESET */
            daliSettingCommandReset();
            break;

        case COMMAND_STORE_ACTUAL_LEVEL_IN_THE_DTR:/* STORE ACTUAL LEVEL IN THE DTR */
            daliSettingCommandStoreActualLevelInTheDTR();
            break;

        case COMMAND_STORE_THE_DTR_AS_MAX_LEVEL:/* STORE THE DTR AS MAX LEVEL */
            daliSettingCommandStoreTheDTRAsMaxLevel();
            break;

        case COMMAND_STORE_THE_DTR_AS_MIN_LEVEL:/* STORE THE DTR AS MIN LEVEL */
            daliSettingCommandStoreTheDTRAsMinLevel();
            break;

      case COMMAND_STORE_THE_DTR_AS_SYSTEM_FAILURE_LEVEL:/* STORE THE DTR AS SYSTEM FAILURE LEVEL */
            daliSettingCommandStoreTheDTRAsSystemFailureLevel();
            break;

        case COMMAND_STORE_THE_DTR_AS_POWER_ON_LEVEL:/* STORE THE DTR AS POWER ON LEVEL */
            daliSettingCommandStoreTheDTRAsPowerOnLevel();
            break;

        case COMMAND_STORE_THE_DTR_AS_FADE_TIME:/* STORE THE DTR AS FADE TIME */
            daliSettingCommandStoreTheDTRAsFadeTime();
            break;

        case COMMAND_STORE_THE_DTR_AS_FADE_RATE:/* STORE THE DTR AS FADE RATE */
            daliSettingCommandStoreTheDTRAsFadeRate();
            break;

        case COMMAND_STORE_DTR_AS_SHORT_ADDRESS:/* STORE DTR AS SHORT ADDRESS */
            daliSettingCommandStoreDTRAsShortAddress();
            break;

        default:
            /* STORE THE DTR AS SCENE */
            if( (ucCommand >= COMMAND_STORE_THE_DTR_AS_SCENE) &&
                (ucCommand < COMMAND_REMOVE_FROM_SCENE) )
            {
                daliSettingCommandStoreTheDTRAsScene( ucCommand );
            }
            /* REMOVE FROM SCENE */
            else if( (ucCommand >= COMMAND_REMOVE_FROM_SCENE) &&
                     (ucCommand < COMMAND_ADD_TO_GROUP) )
            {
                daliSettingCommandRemoveFromScene( ucCommand );
            }
            /* ADD TO GROUP */
            else if( (ucCommand >= COMMAND_ADD_TO_GROUP ) &&
                     (ucCommand < COMMAND_REMOVE_FROM_GROUP) )
```

```
            {
                daliSettingCommandAddToGroup( ucCommand );
            }
            /* REMOVE FROM GROUP */
            else if( (ucCommand >= COMMAND_REMOVE_FROM_GROUP ) &&
                     (ucCommand < COMMAND_STORE_DTR_AS_SHORT_ADDRESS) )
            {
                daliSettingCommandRemoveFromGroup( ucCommand );
            }
            break;

    }
}

void daliDistributeQueryCommand( unsigned char ucCommand )
{
    /* Distribution of Query */
    switch( ucCommand )
    {
        case COMMAND_QUERY_STATUS:/* QUERY STATUS */
            daliQueryCommandStatus();
            break;

        case COMMAND_QUERY_BALAST:/* QUERY BALLAST */
            daliQueryCommandBallast();
            break;

        case COMMAND_QUERY_LAMP_FAILURE:/* QUERY LAMP FAILURE */
            daliQueryCommandLampFailure();
            break;

        case COMMAND_QUERY_LAMP_POWER_ON:/* QUERY LAMP POWERON */
            daliQueryCommandLampPowerOn();
            break;

        case COMMAND_QUERY_LIMIT_ERROR:/* QUERY LIMIT ERROR */
            daliQueryCommandLimitError();
            break;

        case COMMAND_QUERY_RESET_STATE:/* QUERY RESET STATE */
            daliQueryCommandResetState();
            break;

        case COMMAND_QUERY_MISSING_SHORT_ADDRESS:/* QUERY MISSING SHORT ADDRESS */
            daliQueryCommandMissingShortAddress();
            break;

        case COMMAND_QUERY_VERSION_NUMBER:/* QUERY VERSION NUMBER */
            daliQueryCommandVersionNumber();
            break;

        case COMMAND_QUERY_CONTENT_DTR:/* QUERY CONTENT DTR */
            daliQueryCommandContentDTR();
            break;

        case COMMAND_QUERY_DEVICE_TYPE:/* QUERY DEVICE TYPE */
            daliQueryCommandDeviceType();
            break;

        case COMMAND_QUERY_PHYSICAL_MINIMUM_LEVEL:/* QUERY PHYSICAL MINIMUM LEVEL */
            daliQueryCommandPhysicalMinLevel();
            break;

        case COMMAND_QUERY_POWER_FAILURE:/* QUERY POWER FAILURE */
            daliQueryCommandPowerFailure();
            break;

        case COMMAND_QUERY_ACTUAL_LEVEL:/* QUERY ACTUAL LEVEL */
            daliQueryCommandActualLevel();
            break;

        case COMMAND_QUERY_MAX_LEVEL:/* QUERY MAX LEVEL */
            daliQueryCommandMaxLevel();
```

```
            break;

        case COMMAND_QUERY_MIN_LEVEL:/* QUERY MIN LEVEL */
            daliQueryCommandMinLevel();
            break;

        case COMMAND_QUERY_POWER_ON_LEVEL:/* QUERY POWER ON LEVEL */
            daliQueryCommandPowerOnLevel();
            break;

        case COMMAND_QUERY_SYSTEM_FAILURE_LEVEL:/* QUERY SYSTEM FAILURE LEVEL */
            daliQueryCommandSystemFailureLevel();
            break;

        case COMMAND_QUERY_FADE_SETTING:/* QUERY FADE SETTING */
            daliQueryCommandFadeTimeFadeRate();
            break;

        case COMMAND_QUERY_GROUPS_0_7:/* QUERY GROUPS 0-7 */
            daliQueryCommandGroup0_7();
            break;

        case COMMAND_QUERY_GROUPS_8_15:/* QUERY GROUPS 8-15 */
            daliQueryCommandGroup8_15();
            break;

        case COMMAND_QUERY_RANDOM_ADDRESS_H:/* QUERY RANDOM ADDRESS(H) */
            daliQueryCommandRandomAddressH();
            break;

        case COMMAND_QUERY_RANDOM_ADDRESS_M:/* QUERY RANDOM ADDRESS(M) */
            daliQueryCommandRandomAddressM();
            break;

        case COMMAND_QUERY_RANDOM_ADDRESS_L:/* QUERY RANDOM ADDRESS(L) */
            daliQueryCommandRandomAddressL();
            break;

        default:
            /* QUERY SCENE LEVEL */
            if( (ucCommand >= COMMAND_QUERY_SCENE_LEVEL) &&
                (ucCommand < COMMAND_QUERY_GROUPS_0_7) )
            {
                daliQueryCommandSceneLevel(ucCommand);
            }
            break;

    }
}
```

3.22 Continuous Reception of DALI Commands

```
unsigned char daliReceiveContinueCommandCheck( unsigned char ucCommandByte )
{
    unsigned char ucRetValue;

    /* The acceptance of continuous reception command is checked. */
    /* The first reception */
    if( g_ucReceiveContinueFlag == FLAG_DOWN )
    {
        ucRetValue = TRUE;
    }
    /* The first reception */
    else
    {
        if( g_ucCommandBackup == ucCommandByte )
        {
            ucRetValue = TRUE;
```

```
        }
        else
        {
            g_ucCommandBackup = COMMAND_INVALID;
            daliReceiveIntervalStop();
            ucRetValue = FALSE;
        }
    }

    return ucRetValue;
}

unsigned char daliReceiveContinueSettingCommandCheck( unsigned char ucCommandByte )
{
    unsigned char ucRetValue;

    /* The acceptance of continuous reception command is checked. */
    /* The first reception */
    if( g_ucReceiveContinueFlag == FLAG_DOWN )
    {
        if( (ucCommandByte >= COMMAND_RESET) &&
            (ucCommandByte <= COMMAND_STORE_DTR_AS_SHORT_ADDRESS) )
        {
            g_ucCommandBackup = ucCommandByte;
            daliCommunicationIntervalSet( RECEIVE_CONTINUE_INTERVAL );
            ucRetValue = FALSE;
        }
        else
        {
            g_ucCommandBackup = ucCommandByte;
            ucRetValue = TRUE;
        }
    }
    /* The first reception */
    else
    {
        if( g_ucCommandBackup == ucCommandByte )
        {
            ucRetValue = TRUE;
        }
        else
        {
            g_ucCommandBackup = COMMAND_INVALID;
            daliReceiveIntervalStop();
            ucRetValue = FALSE;
        }
    }

    return ucRetValue;
}
```

3.23 Processing DALI Commands

```
void daliLightingCommandDirectArcPowerControl( unsigned char ucDataByte )
{
    /* The timer that controls the fade is stopped. */
    daliFadeTimerStop(FLAG_DOWN);

    /* The fade is set. */
    daliSubTimeFadeSet( ucDataByte );
}

void daliLightingCommandOff( void )
{
    /* The timer that controls the fade is stopped. */
    daliFadeTimerStop(FLAG_DOWN);

    /* 0 is set to the ballast actual Level */
```

```
    g_ucActualLevel = 0;

    STATUS_INFO_LIMIT_ERROR = BIT_CLR;
    STATUS_INFO_POWER_FAILURE = BIT_CLR;
}

void daliLightingCommandUp( void )
{
    /* The timer that controls the fade is stopped. */
    daliFadeTimerStop(FLAG_DOWN);

    /* The fade-up is done according to the fade fade rate.  */
    if( (g_ucActualLevel != 0) &&
        (g_ucActualLevel < g_ucBallastData[BALLAST_DATA_MAX_LEVEL]) )
    {
        daliSubRateFadePrepare(g_ucBallastData[BALLAST_DATA_FADE_RATE], FADE_UP );
        STATUS_INFO_LIMIT_ERROR = BIT_CLR;
        STATUS_INFO_POWER_FAILURE = BIT_CLR;
    }
}

void daliLightingCommandDown( void )
{
    /* The timer that controls the fade is stopped. */
    daliFadeTimerStop(FLAG_DOWN);

    /* The fade-down is done according to the fade fade rate.  */
    if( (g_ucActualLevel != 0) &&
        (g_ucActualLevel > g_ucBallastData[BALLAST_DATA_MIN_LEVEL]) )
    {
        daliSubRateFadePrepare( g_ucBallastData[BALLAST_DATA_FADE_RATE], FADE_DOWN );
        STATUS_INFO_LIMIT_ERROR = BIT_CLR;
        STATUS_INFO_POWER_FAILURE = BIT_CLR;
    }
}

void daliLightingCommandStepUp( void )
{
    /* The timer that controls the fade is stopped. */
    daliFadeTimerStop(FLAG_DOWN);

    /* Acutual level is raised by one step.  */
    if( (g_ucActualLevel!= 0) &&
         (g_ucActualLevel < g_ucBallastData[BALLAST_DATA_MAX_LEVEL]) )
    {
        g_ucActualLevel++;
        STATUS_INFO_LIMIT_ERROR = BIT_CLR;
        STATUS_INFO_POWER_FAILURE = BIT_CLR;
    }
}

void daliLightingCommandStepDown( void )
{
    /* The timer that controls the fade is stopped. */
    daliFadeTimerStop(FLAG_DOWN);

    /* Acutual level is lowered by one step. */
    if( (g_ucActualLevel!= 0) &&
        (g_ucActualLevel>g_ucBallastData[BALLAST_DATA_MIN_LEVEL]) )
    {
        g_ucActualLevel--;
        STATUS_INFO_LIMIT_ERROR = BIT_CLR;
        STATUS_INFO_POWER_FAILURE = BIT_CLR;
    }
}

void daliLightingCommandRecallMaxLevel( void )
{
    /* The timer that controls the fade is stopped. */
    daliFadeTimerStop(FLAG_DOWN);

    /* MAX LEVEL is set to actual level. */
    g_ucActualLevel = g_ucBallastData[BALLAST_DATA_MAX_LEVEL];
```

```
    STATUS_INFO_LIMIT_ERROR = BIT_CLR;
    STATUS_INFO_POWER_FAILURE = BIT_CLR;
}

void daliLightingCommandRecallMinLevel( void )
{
    /* The timer that controls the fade is stopped. */
    daliFadeTimerStop(FLAG_DOWN);

    /* MIN LEVEL is set to actual level. */
    g_ucActualLevel = g_ucBallastData[BALLAST_DATA_MIN_LEVEL];

    STATUS_INFO_LIMIT_ERROR = BIT_CLR;
    STATUS_INFO_POWER_FAILURE = BIT_CLR;
}

void daliLightingCommandStepDownAndOff( void )
{
    /* The timer that controls the fade is stopped. */
    daliFadeTimerStop(FLAG_DOWN);

    /* Acutual level is lowered by one step. */
    /* 0 is set to actual level
       when actual level is below MIN LEVEL. */
    if( g_ucActualLevel <= g_ucBallastData[BALLAST_DATA_MIN_LEVEL] )
    {
        g_ucActualLevel = 0;
    }
    else
    {
        g_ucActualLevel --;
    }

    STATUS_INFO_LIMIT_ERROR = BIT_CLR;
    STATUS_INFO_POWER_FAILURE = BIT_CLR;
}

void daliLightingCommandOnAndStepUp( void )
{
    /* The timer that controls the fade is stopped. */
    daliFadeTimerStop(FLAG_DOWN);

    /* Acutual level is raised by one step. */
    /* When actual level is 0,
       MIN LEVEL is set to actual level. */
    if( g_ucActualLevel != 0 )
    {
        if( g_ucActualLevel < g_ucBallastData[BALLAST_DATA_MAX_LEVEL] )
        {
            g_ucActualLevel ++;
            STATUS_INFO_LIMIT_ERROR = BIT_CLR;
            STATUS_INFO_POWER_FAILURE = BIT_CLR;
        }
    }
    else
    {
        g_ucActualLevel = g_ucBallastData[BALLAST_DATA_MIN_LEVEL];
        STATUS_INFO_LIMIT_ERROR = BIT_CLR;
        STATUS_INFO_POWER_FAILURE = BIT_CLR;
    }
}

void daliLightingCommandGoToScene( unsigned char ucDataByte )
{
    unsigned char ucSceneNumber;

    /* The timer that controls the fade is stopped. */
    daliFadeTimerStop(FLAG_DOWN);

    ucSceneNumber = ucDataByte & 0x0F;

    /* The fade is set. */
```

```
    daliSubTimeFadeSet( g_ucBallastData[(BALLAST_DATA_SCENE_0 + ucSceneNumber)] );
}

void daliLightingCommandSystemFailure(void)
{
    /* The timer that controls the fade is stopped. */
    daliFadeTimerStop(FLAG_DOWN);

    /* System failure level is set to actual level. */
    if( (g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL] > 0x00) &&
        (g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL] <
         g_ucBallastData[BALLAST_DATA_MIN_LEVEL]) )
    {
        g_ucActualLevel = g_ucBallastData[BALLAST_DATA_MIN_LEVEL];
        STATUS_INFO_LIMIT_ERROR = BIT_SET;
    }
    else if( (g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL] >
              g_ucBallastData[BALLAST_DATA_MAX_LEVEL]) &&
             (g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL] < 0xFF) )
    {
        g_ucActualLevel = g_ucBallastData[BALLAST_DATA_MAX_LEVEL];
        STATUS_INFO_LIMIT_ERROR = BIT_SET;
    }
    else if( (g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL] == 0x00) ||
            ((g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL] >=
              g_ucBallastData[BALLAST_DATA_MIN_LEVEL]) &&
             (g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL] <=
              g_ucBallastData[BALLAST_DATA_MAX_LEVEL])) )
    {
        g_ucActualLevel = g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL];
        STATUS_INFO_LIMIT_ERROR = BIT_CLR;
    }

}

void daliSettingCommandReset( void )
{
    unsigned char ucSceneCount;
    unsigned char ucRetValue = 255;

    /* The ballast data is set to the reset value. */
    g_ucBallastData[BALLAST_DATA_MIN_LEVEL] = RESET_MINLEVEL;
    g_ucBallastData[BALLAST_DATA_MAX_LEVEL] = RESET_MAXLEVEL;
    g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL] = RESET_POWERONLEVEL;
    g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL] = RESET_SYSTEMFAILURELEVEL;
    g_ucBallastData[BALLAST_DATA_FADE_RATE] = RESET_FADERATE;
    g_ucBallastData[BALLAST_DATA_FADE_TIME] = RESET_FADETIME;
    g_ucSearchAddressH = RESET_SEARCHADDRESSH;
    g_ucSearchAddressM = RESET_SEARCHADDRESSM;
    g_ucSearchAddressL = RESET_SEARCHADDRESSL;
    g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_H] = RESET_RANDOMADDRESSH;
    g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_M] = RESET_RANDOMADDRESSM;
    g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_L] = RESET_RANDOMADDRESSH;
    g_ucBallastData[BALLAST_DATA_GROUP_0_7] = RESET_GROUP_0_7;
    g_ucBallastData[BALLAST_DATA_GROUP_8_15] = RESET_GROUP_8_15;

   for( ucSceneCount = BALLAST_DATA_SCENE_0 ; ucSceneCount <= BALLAST_DATA_SCENE_15 ; ucSceneCount++ )
    {
        g_ucBallastData[ucSceneCount] = RESET_SCENE;
    }

    STATUS_INFO_ALL = RESET_STATUS_INFORMATION;
    g_ucActualLevel = RESET_ACTUAL_LEVEL;

    if( g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS] == RESET_SHORTADDRESS )
    {
        STATUS_INFO_MISSING_SHORT_ADDRESS = BIT_SET;
    }
    else
    {
        STATUS_INFO_MISSING_SHORT_ADDRESS = BIT_CLR;
    }
```

```
    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}

void daliSettingCommandStoreActualLevelInTheDTR( void )
{
    /* Store actual level in the DTR. */
    g_ucDTR = g_ucActualLevel;
}

void daliSettingCommandStoreTheDTRAsMaxLevel( void )
{
    unsigned char ucRetValue = 255;

    if( g_ucDTR<g_ucBallastData[BALLAST_DATA_MIN_LEVEL]||
        g_ucDTR>RANGE_MAXLEVEL )
    {
        return;
    }

    /* The value of DTR is stored in MAX LEVEL of the ballast data. */
    g_ucBallastData[BALLAST_DATA_MAX_LEVEL] = g_ucDTR;
    if( g_ucBallastData[BALLAST_DATA_MAX_LEVEL] < g_ucActualLevel )
    {
        if( g_ucFadeFlag == FLAG_UP )
        {
            daliFadeTimerStop(FLAG_DOWN);
        }
        g_ucActualLevel = g_ucBallastData[BALLAST_DATA_MAX_LEVEL];
    }
    STATUS_INFO_RESET_STATE = BIT_CLR;

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}

void daliSettingCommandStoreTheDTRAsMinLevel( void )
{
    unsigned char ucRetValue = 255;

    if( g_ucDTR > g_ucBallastData[BALLAST_DATA_MAX_LEVEL] )
    {
        return;
    }
    /* The value of DTR is stored in MIN LEVEL of the ballast data. */
    if( g_ucDTR >= g_ucSleaveData[SLEAVE_DATA_PHYSICAL_MIN_LEVEL] )
    {
        g_ucBallastData[BALLAST_DATA_MIN_LEVEL] = g_ucDTR;
    }
    else
    {
        g_ucBallastData[BALLAST_DATA_MIN_LEVEL] = g_ucSleaveData[SLEAVE_DATA_PHYSICAL_MIN_LEVEL];
    }

    if( g_ucActualLevel != 0 )
    {
        if( g_ucBallastData[BALLAST_DATA_MIN_LEVEL] > g_ucActualLevel )
        {
            if( g_ucFadeFlag == FLAG_UP )
            {
                daliFadeTimerStop(FLAG_DOWN);
            }
            g_ucActualLevel = g_ucBallastData[BALLAST_DATA_MIN_LEVEL];

        }
```

```
    }

    STATUS_INFO_RESET_STATE = BIT_CLR;

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}

void daliSettingCommandStoreTheDTRAsSystemFailureLevel( void )
{
    unsigned char ucRetValue = 255;

    /* The value of DTR is stored in SYSTEM FAILURE LEVEL of the ballast data. */
    g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL] = g_ucDTR;

    STATUS_INFO_RESET_STATE = BIT_CLR;

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}

void daliSettingCommandStoreTheDTRAsPowerOnLevel( void )
{
    unsigned char ucRetValue = 255;

    if( (g_ucDTR == 0) || (g_ucDTR == 255) )
    {
        return;
    }

    /* The value of DTR is stored in POWER ON LEVEL of the ballast data. */
    g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL] = g_ucDTR;

    STATUS_INFO_RESET_STATE = BIT_CLR;

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}

void daliSettingCommandStoreTheDTRAsFadeTime( void )
{
    unsigned char ucRetValue = 255;

    if( (g_ucDTR < 0) || (g_ucDTR > 15) )
    {
        return;
    }

    /* The value of DTR is stored in FADE TIME of the ballast data. */
    g_ucBallastData[BALLAST_DATA_FADE_TIME] = g_ucDTR;

    STATUS_INFO_RESET_STATE = BIT_CLR;

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}
```

```
void daliSettingCommandStoreTheDTRAsFadeRate( void )
{
    unsigned char ucRetValue = 255;

    if( g_ucDTR < 1 || g_ucDTR > 15 )
    {
        return;
    }

    /* The value of DTR is stored in FADE RATE of the ballast data. */
    g_ucBallastData[BALLAST_DATA_FADE_RATE] = g_ucDTR;

    STATUS_INFO_RESET_STATE = BIT_CLR;

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}

void daliSettingCommandStoreTheDTRAsScene( unsigned char ucDataByte )
{
    unsigned char ucRetValue = 255;
    unsigned char ucSceneNumber;

    /* The value of DTR is stored in Scene XX of the ballast data. */
    ucSceneNumber = (ucDataByte & 0x0F);
    g_ucBallastData[(BALLAST_DATA_SCENE_0 + ucSceneNumber)] = g_ucDTR;

    STATUS_INFO_RESET_STATE = BIT_CLR;

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}

void daliSettingCommandRemoveFromScene( unsigned char ucDataByte )
{
    unsigned char ucRetValue = 255;
    unsigned char ucSceneNumber;

    /* Scene XX of the ballast data is removed. */
    ucSceneNumber = ( ucDataByte & 0x0F );
    g_ucBallastData[(BALLAST_DATA_SCENE_0 + ucSceneNumber)] = REMOVE_FROM_SCENE;

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}

oid daliSettingCommandAddToGroup( unsigned char ucDataByte )
{
    unsigned char ucRetValue = 255;
    unsigned char ucGroup;

    /* The ballast is added to group XX. */
    ucGroup = (ucDataByte & 0x0F);
    if( ucGroup < 8 )
    {
        g_ucBallastData[BALLAST_DATA_GROUP_0_7] = (g_ucBallastData[BALLAST_DATA_GROUP_0_7] |
        g_ucGroupPattern[ucGroup]);
    }
    else
    {
        g_ucBallastData[BALLAST_DATA_GROUP_8_15] = (g_ucBallastData[BALLAST_DATA_GROUP_8_15] |
```

```
        g_ucGroupPattern[ucGroup]);
    }

    STATUS_INFO_RESET_STATE = BIT_CLR;

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}

void daliSettingCommandRemoveFromGroup( unsigned char ucDataByte )
{
    unsigned char ucRetValue = 255;
    unsigned char ucGroup;

    /* The ballast is removed from group XX. */
    ucGroup = (ucDataByte & 0x0F);
    if( ucGroup < 8 )
    {
        g_ucBallastData[BALLAST_DATA_GROUP_0_7] = (g_ucBallastData[BALLAST_DATA_GROUP_0_7] &
        (~g_ucGroupPattern[ucGroup]));
    }
    else
    {
        g_ucBallastData[BALLAST_DATA_GROUP_8_15] = (g_ucBallastData[BALLAST_DATA_GROUP_8_15] &
        (~g_ucGroupPattern[ucGroup]));
    }

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}

void daliSettingCommandStoreDTRAsShortAddress( void )
{
    unsigned char ucRetValue = 255;

    /* If the value of DTR is 0b11111111 then a short address is deleted,
       or the value of DTR is stored at a short address. */
    if( g_ucDTR == 0xFF )
    {
        g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS] = SHORT_ADDRESS_INVALID;
    }
    else if( (g_ucDTR & 0x81) == 0x01 )
    {
        g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS] = ((g_ucDTR >> 1) & 0x3F);
        STATUS_INFO_RESET_STATE = BIT_CLR;
    }

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}

void daliQueryCommandStatus( void )
{
    if( g_ucActualLevel != 0 )
    {
        STATUS_INFO_LAMP_POWER_ON = BIT_SET;
    }
    else
    {
        STATUS_INFO_LAMP_POWER_ON = BIT_CLR;
    }
```

```
    /* Status information is transmitted. */
    g_ucSendData = STATUS_INFO_ALL;
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}
void daliQueryCommandBallast( void )
{
    /* When the state of the ballast is NG, YES is transmitted. */
    if( !STATUS_INFO_BALLAST )
    {
        g_ucSendData = SEND_YES;
        daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
    }
}


void daliQueryCommandLampFailure( void )
{
    /* When the breakdown of the lamp is found, YES is transmitted. */
    if( STATUS_INFO_LAMP_FAILURE )
    {
        g_ucSendData = SEND_YES;
        daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
    }
}

void daliQueryCommandLampPowerOn( void )
{
    /* When the lamp power supply is turning on, YES is transmitted. */
    if( g_ucActualLevel != 0 )
    {
        STATUS_INFO_LAMP_POWER_ON = BIT_SET;
        g_ucSendData = SEND_YES;
        daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
    }
    else
    {
        STATUS_INFO_LAMP_POWER_ON = BIT_CLR;
    }
}

void daliQueryCommandLimitError( void )
{
    /* When the limit error of actul level is found, YES is transmitted */
    if( STATUS_INFO_LIMIT_ERROR )
    {
        g_ucSendData = SEND_YES;
        daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
    }
}
void daliQueryCommandResetState( void )
{
    unsigned char ucRetValue;

    /* When the ballast data is equal to the reset value,
       YES is transmitted. */
    ucRetValue = daliBallastData_ResetValueCheck();
    if( ucRetValue )
    {
        STATUS_INFO_RESET_STATE = BIT_SET;
        g_ucSendData = SEND_YES;
        daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
    }
    else
    {
        STATUS_INFO_RESET_STATE = BIT_CLR;
    }
}


void daliQueryCommandMissingShortAddress( void )
{
    /* When the ballast doesn't have a short address, YES is transmitted. */
```

```
    if( STATUS_INFO_MISSING_SHORT_ADDRESS )
    {
        g_ucSendData = SEND_YES;
        daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
    }
}

void daliQueryCommandVersionNumber( void )
{
    /* Version number is transmitted. */
    g_ucSendData = g_ucSleaveData[SLEAVE_DATA_VERSION_NUMBER];
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}

void daliQueryCommandContentDTR( void )
{
    /* The value stored in DTR is transmitted. */
    g_ucSendData = g_ucDTR;
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );

}

void daliQueryCommandDeviceType( void )
{
    /* Device type is transmitted. */
    g_ucSendData = g_ucSleaveData[SLEAVE_DATA_DEVICE_TYPE];
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}

void daliQueryCommandPhysicalMinLevel( void )
{
    /* Physical Min. Level is transmitted. */
    g_ucSendData = g_ucSleaveData[SLEAVE_DATA_PHYSICAL_MIN_LEVEL];
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}

void daliQueryCommandPowerFailure( void )
{
    /* When it doesn't reset after the power supply is turned on
       or actual level is not set, YES is transmitted. */
    if( STATUS_INFO_POWER_FAILURE )
    {
        g_ucSendData = SEND_YES;
        daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
    }
}

void daliQueryCommandActualLevel( void )
{
    /* Actual level is transmitted.
     * When the limit error of actul level is found, 0xFF is transmitted. */
    if( !STATUS_INFO_LIMIT_ERROR )
    {
        g_ucSendData = g_ucActualLevel;
    }
    else
    {
        g_ucSendData = 0xFF;
    }
        daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}

void daliQueryCommandMaxLevel( void )
{
    /* MAX LEVEL is transmitted. */
    g_ucSendData = g_ucBallastData[BALLAST_DATA_MAX_LEVEL];
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}


void daliQueryCommandMinLevel( void )
{
    /* MIN LEVEL is transmitted. */
```

```
    g_ucSendData = g_ucBallastData[BALLAST_DATA_MIN_LEVEL];
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}


void daliQueryCommandPowerOnLevel( void )
{
    /* POWER ON LEVEL is transmitted. */
    g_ucSendData = g_ucBallastData[BALLAST_DATA_POWER_ON_LEVEL];
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}



void daliQueryCommandSystemFailureLevel( void )
{
    /* SYSTEMS FAILURE LEVEL is transmitted. */
    g_ucSendData = g_ucBallastData[BALLAST_DATA_SYSTEM_FAILURE_LEVEL];
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}


void daliQueryCommandFadeTimeFadeRate( void )
{
    unsigned char ucSendBuff;

    /* FADE TIME/FADE RATE is transmitted. */
    /* FADE TIME is set to high rank 4 bits,
       and FADE RATE is set to subordinate position 4 bits.  */
    ucSendBuff = (g_ucBallastData[BALLAST_DATA_FADE_TIME] << 4) ;
    ucSendBuff = ucSendBuff + g_ucBallastData[BALLAST_DATA_FADE_RATE];
    g_ucSendData = ucSendBuff;
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}



void daliQueryCommandSceneLevel( unsigned char ucDataByte )
{
    unsigned char unSceneNumber;

    /* Actual level registered in SCENE XX is transmitted. */
    unSceneNumber = (ucDataByte & 0x0F);
    g_ucSendData = g_ucBallastData[(BALLAST_DATA_SCENE_0 + unSceneNumber)];
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}



void daliQueryCommandGroup0_7( void )
{
    /* The belonging situation from GROUP 0-7 is transmitted. */
    g_ucSendData = g_ucBallastData[BALLAST_DATA_GROUP_0_7];
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}


void daliQueryCommandGroup8_15( void )
{
    /* The belonging situation from GROUP 8-15 is transmitted. */
    g_ucSendData = g_ucBallastData[BALLAST_DATA_GROUP_8_15];
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}


void daliQueryCommandRandomAddressH( void )
{
    /* RANDOM ADDRESS(H) is transmitted. */
    g_ucSendData = g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_H];
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}
```

```
void daliQueryCommandRandomAddressM( void )
{
    /* RANDOM ADDRESS(M) is transmitted. */
    g_ucSendData = g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_M];
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}


void daliQueryCommandRandomAddressL( void )
{
    /* RANDOM ADDRESS(L) is transmitted. */
    g_ucSendData = g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_L];
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}


void daliExCommandTerminate( void )
{
    /* The address setting period is stopped. */
    daliAddressSettingIntervalStop();
}


void daliExCommandDTR( unsigned char ucDataByte )
{
    /* The value is stored in DTR. */
    g_ucDTR = ucDataByte;
}


void daliExCommandInitialize( unsigned char ucDataByte )
{
    unsigned char ucRetValue = 0;

    ucRetValue = daliExCommandInitializeBallastCheck( ucDataByte );
    if( !ucRetValue )
    {
        return;
    }

    if( g_ucAddressSettingFlag == FLAG_DOWN )
    {
        /* The command is backed up at the first reception. */
        if( g_ucReceiveContinueFlag == FLAG_DOWN )
        {
            g_ucCommandBackup = EXCOMMAND_INITIALIZE;
            daliCommunicationIntervalSet( RECEIVE_CONTINUE_INTERVAL );
        }
        /* The address setting period is begun at the second reception. */
        else
        {
            if( g_ucCommandBackup == EXCOMMAND_INITIALIZE )
            {
                daliReceiveIntervalStop();
                daliCommunicationIntervalSet( ADDRESS_SETTING_INTERVAL );
            }
            else
            {
                g_ucCommandBackup = COMMAND_INVALID;
                daliReceiveIntervalStop();
            }
        }

    }
    /* The address setting period is extended
       at the reception since the third times. */
    else
    {
        daliCommunicationIntervalSet( ADDRESS_SETTING_INTERVAL );
    }
}
```

```
void daliExCommandRandomise( void )
{
    /* RANDOM ADDRESS is set. */

    /* The command is effective only for the address setting period. */
    if( g_ucAddressSettingFlag == FLAG_DOWN )
    {
        return;
    }

    /* The command is backed up at the first reception. */
    if( g_ucReceiveContinueFlag == FLAG_DOWN )
    {
        g_ucCommandBackup = EXCOMMAND_RANDOMISE;
        daliCommunicationIntervalSet( RECEIVE_CONTINUE_INTERVAL );
    }
    /* RANDOM ADDRESS is set at the second reception. */
    else
    {
        if( g_ucCommandBackup == EXCOMMAND_RANDOMISE )
        {
            daliExCommandRandomiseRandomaddr();
        }
        else
        {
            g_ucCommandBackup = COMMAND_INVALID;
        }
        daliReceiveIntervalStop();
    }
}


void daliExCommandCompare( void )
{
    /* When the address setting period, the WITHDRAW setting is invalid
       and the physical selection mode is invalid, the command is effective. */
    if( (g_ucAddressSettingFlag == FLAG_DOWN) ||
        (g_ucWithdraw == FLAG_UP) ||
        (g_ucPhysicalSelection == FLAG_UP) )
    {
        return;
    }

    /* When RANDOM ADDRESS is smaller than SEARCH ADDRESS, YES is transmitted. */
    if( g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_H] > g_ucSearchAddressH)
    {
        return;
    }
    else if( g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_H] == g_ucSearchAddressH&&
        g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_M] > g_ucSearchAddressM)
    {
        return;
    }
    else if( g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_H] == g_ucSearchAddressH&&
        ( g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_M] == g_ucSearchAddressM) &&
        ( g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_L] > g_ucSearchAddressL) )
    {
        return;
    }

    g_ucSendData = SEND_YES;
    daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
}


void daliExCommandWithdraw( void )
{
    /* When the address setting period and
       RANDOM ADDRESS is smaller than SEARCH ADDRESS, the WITDRAW setting is turned on. */
    if( (g_ucAddressSettingFlag == FLAG_DOWN) ||
        (g_ucSearchAddressH != g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_H]) ||
        (g_ucSearchAddressM != g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_M]) ||
        (g_ucSearchAddressL != g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_L]) )
```

```
    {
        return;
    }

    g_ucWithdraw = FLAG_UP;
}


void daliExCommandSearchAddrH( unsigned char ucDataByte )
{
    /* When the address setting period, SEATH ADDRESS(H) is set. */
    if( (g_ucAddressSettingFlag == FLAG_DOWN) )
    {
        return;
    }

    g_ucSearchAddressH = ucDataByte;
}


void daliExCommandSearchAddrM( unsigned char ucDataByte )
{
    /* When the address setting period, SEATH ADDRESS(M) is set. */
    if( (g_ucAddressSettingFlag == FLAG_DOWN) )
    {
        return;
    }

    g_ucSearchAddressM = ucDataByte;
}

void daliExCommandSearchAddrL( unsigned char ucDataByte)
{
    /* When the address setting period, SEATH ADDRESS(L) is set. */
    if( (g_ucAddressSettingFlag == FLAG_DOWN) )
    {
        return;
    }

    g_ucSearchAddressL = ucDataByte;
}


void daliExCommandProgramShortAddress( unsigned char ucDataByte )
{
    unsigned char ucRetValue = 255;

    /* When the address setting period and the physical selection mode is invalid or
       SEARCH ADDRESS is equal to RANDOM ADDRESS, a short address is set. */
    if( g_ucAddressSettingFlag == FLAG_DOWN )
    {
        return;
    }
    if( g_ucPhysicalSelection == FLAG_UP )
    {
        g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS] = ((ucDataByte >> 1) & 0x3F);
    }
    else if( (g_ucSearchAddressH == g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_H]) &&
            (g_ucSearchAddressM == g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_M]) &&
            (g_ucSearchAddressL == g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_L]) )
    {
        g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS] = ((ucDataByte >> 1) & 0x3F);
    }

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}
```

```
void daliExCommandDeleteTheShortAddress( void )
{
    unsigned char ucRetValue = 255;

    /* When the address setting period and the physical selection mode is invalid or
       SEARCH ADDRESS is equal to RANDOM ADDRESS, a short address is deleted. */
    if( g_ucAddressSettingFlag == FLAG_DOWN )
    {
        return;
    }
    if( g_ucPhysicalSelection == FLAG_UP )
    {
        g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS] = SHORT_ADDRESS_INVALID;
    }
    else if( (g_ucSearchAddressH == g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_H]) &&
            (g_ucSearchAddressM == g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_M]) &&
            (g_ucSearchAddressL == g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_L]) )
    {
        g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS] = SHORT_ADDRESS_INVALID;
    }

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}


void daliExCommandVerifyShortAddress( unsigned char ucDataByte )
{
    unsigned char ucDataByteBuff;

    /* When the address setting period, the command is effective. */
    if( g_ucAddressSettingFlag == FLAG_DOWN )
    {
        return;
    }

    /* When the value of the databyte is an unexpected value,
       the command is not processed.  */
    if( (ucDataByte & 0x81) != 0x01 )
    {
        return;
    }

    /* When it is equal to a short address of the ballast, YES is transmitted. */
    ucDataByteBuff = ((ucDataByte >> 1) & 0x3F);

    if( g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS] == ucDataByteBuff )
    {
        g_ucSendData = SEND_YES;
        daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
    }
}


void daliExCommandQueryShortAddress( void )
{
    unsigned char ucShortAddress;

    /* When the address setting period and the physical selection mode is invalid
       or SEARCH ADDRESS is equal to RANDOM ADDRESS,
       a short address of the ballast is transmitted. */
    if( g_ucAddressSettingFlag == FLAG_DOWN )    {
        return;
    }

    if( (g_ucPhysicalSelection == FLAG_UP) ||
        ( (g_ucSearchAddressH == g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_H]) &&
          (g_ucSearchAddressM == g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_M]) &&
          (g_ucSearchAddressL == g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_L]) ) )
```

```
    {
        ucShortAddress = (g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS] << 1);
        ucShortAddress = (ucShortAddress | 0x01);

        g_ucSendData = ucShortAddress;
        daliCommunicationIntervalSet( BACKWARD_SEND_INTERBAL );
    }
}


void daliExCommandPhysicalSelection( void )
{
    /* When the address setting period, the physical selection mode is turned on. */
    if( g_ucAddressSettingFlag == FLAG_DOWN )
    {
        return;
    }

    g_ucPhysicalSelection = FLAG_UP;
}


unsigned char daliExCommandInitializeBallastCheck( unsigned char ucDataByte )
{
    unsigned char ucShortAddressBuff;
    unsigned char ucRetValue;

    /* It is checked whether the ballast is a target. */
    /* All ballasts are target. */
    if( ucDataByte == 0x00 )
    {
        ucRetValue = TRUE;

    }
    /* The ballast without a short address is a target. */
    else if( ucDataByte == 0xFF )
    {
        if(g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS] == SHORT_ADDRESS_INVALID)
        {
            ucRetValue = TRUE;
        }
        else
        {
            ucRetValue = FALSE;
        }
    }
    /* The ballast with the specified short address is a target. */
    else if( (ucDataByte & 10000001) == 0x01 )
    {
        ucShortAddressBuff = ((ucDataByte >> 1) & 0x3F);

        if( g_ucBallastData[BALLAST_DATA_SHORT_ADDRESS] == ucShortAddressBuff )
        {
            ucRetValue = TRUE;
        }
        else
        {
            ucRetValue = FALSE;
        }
    }
    else
    {
        ucRetValue = FALSE;
    }

    return ucRetValue;
}


void daliExCommandRandomiseRandomaddr( void )
{
    unsigned short usRandNumber;
    unsigned char ucRetValue = 255;
```

```
    /* RANDOM ADDRESS is generated. */
    srand( (unsigned int)g_ulAddressSettingCount );

    usRandNumber = rand();
   g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_H] = (unsigned char)((usRandNumber >>  8) & 0x00FF);
    g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_M] = (unsigned char)(usRandNumber & 0x00FF);

    usRandNumber = rand();
    g_ucBallastData[BALLAST_DATA_RANDOM_ADDRESS_L] = (unsigned char)(usRandNumber & 0x00FF);

    /* The ballast data is written in the memory. */
    ucRetValue = self_Write();
    if( ucRetValue != WRITE_OK )
    {
        daliLightingCommandSystemFailure();
    }
}


void daliSubTimeFadePrepare( unsigned char ucStartLevel,unsigned char ucEndLevel )
{
    unsigned long ulTotalFadeTime;

    /* The timer for the fade control is started. */
    if( ucStartLevel == ucEndLevel )
    {
        return;
    }
    else if( ucStartLevel > ucEndLevel )
    {
        g_ucFadeDirection = FADE_DOWN;
        g_ucFadeCount = ucStartLevel - ucEndLevel;
    }
    else
    {
        g_ucFadeDirection = FADE_UP;
        g_ucFadeCount = ucEndLevel - ucStartLevel;
    }

    ulTotalFadeTime = g_ulFadeTimeData[g_ucBallastData[BALLAST_DATA_FADE_TIME]];
    daliFadeTimerStart( g_ucFadeCount, ulTotalFadeTime );

}


void daliSubRateFadePrepare( unsigned char ucFadeRate,unsigned char ucFadeDir )
{
    unsigned long ulTotalFadeTime;
    unsigned long ulFadeRateData;

    /* To do 200 millisecond fade, the timer is started. */
    ulFadeRateData = g_ulFadeRateData[ucFadeRate];

    if( ulFadeRateData != 0 )
    {
        g_ucFadeDirection = ucFadeDir;
        g_ucFadeCount = (unsigned char)(200000 / ulFadeRateData);
        ulTotalFadeTime = 200000;

        daliFadeTimerStart( g_ucFadeCount, ulTotalFadeTime );
    }
}


void daliSubTimeFadeSet( unsigned char ucEndLevel )
{
    /* The fade is set. */
    if( ucEndLevel == 0xFF )
    {
        STATUS_INFO_LIMIT_ERROR = BIT_CLR;
        return ;
    }
```

```
    else if( ucEndLevel == 0x00 )
    {
        ucEndLevel = g_ucBallastData[BALLAST_DATA_MIN_LEVEL] - 1;
        STATUS_INFO_LIMIT_ERROR = BIT_CLR;
        STATUS_INFO_POWER_FAILURE = BIT_CLR;
    }
    else if( ucEndLevel < g_ucBallastData[BALLAST_DATA_MIN_LEVEL] )
    {
        ucEndLevel = g_ucBallastData[BALLAST_DATA_MIN_LEVEL];
        STATUS_INFO_LIMIT_ERROR = BIT_SET;
    }
    else if( ucEndLevel > g_ucBallastData[BALLAST_DATA_MAX_LEVEL] )
    {
        ucEndLevel = g_ucBallastData[BALLAST_DATA_MAX_LEVEL];
        STATUS_INFO_LIMIT_ERROR = BIT_SET;
    }
    else
    {
        STATUS_INFO_LIMIT_ERROR = BIT_CLR;
        STATUS_INFO_POWER_FAILURE = BIT_CLR;
    }

    if( g_ucBallastData[BALLAST_DATA_FADE_TIME] != 0 )
    {
        if( g_ucActualLevel == 0 )
        {
            daliSubTimeFadePrepare( g_ucBallastData[BALLAST_DATA_MIN_LEVEL] - 1, ucEndLevel );
        }
        else
        {
            daliSubTimeFadePrepare( g_ucActualLevel, ucEndLevel );
        }
    }
    else
    {
        if( ucEndLevel < g_ucBallastData[BALLAST_DATA_MIN_LEVEL] )
        {
            g_ucActualLevel = 0;
        }
        else
        {
            g_ucActualLevel = ucEndLevel;
        }
    }
}
```