ISELED Starter Kit

ILaS®

Guide for ISELED Library

Introduction

This document describes the ISELED library code.

Target Device

The target is the Renesas RH850/F1KM-S1 microcontroller. Same concept applies to RH850/F1KM-S2 and F1KM-S4 microcontrollers.

In this demonstration project the general product R7F701684 is used. Please note for ISELED additional products are available as described in the document RH850/F1KM, User's Manual Hardware, Addendum for ISELED (R01UH1024EJxxxx).

Prerequisites

Documents to read before this specification:

- ISELED Communication Protocol Control Commands document from Inova Semiconductor (ISP_INLC Revision 1.1)
- RH850/F1KM, User's Manual Hardware, Addendum for ISELED (R01UH1024EJxxxx)

Contents

1. ISELED Overview

On the ISP_INLC ISELED Communication Protocol - Control Commands Application for Inova Semiconductor, the ISELED Protocol is defined as:

"The ISELED communication protocol implements a half-duplex, bidirectional, high speed serial master-slave communication between a LED strip controller unit and up to 4079 ISELEDs.

The attachment to the adjacent devices in the chain is made up by two bidirectional differential serial communication lines. The direction towards the controlling microcontroller device is referred to as the "upstream" connection. The opposite direction towards the end of the chain is the "downstream" link. Both links are controlled by the communication unit. Incoming command frames from upstream and responses from downstream are passed to the main unit which is responsible for command processing and overall device control. Commands always originate from the controlling microcontroller. The microcontroller is referred to as the "host" in this document.

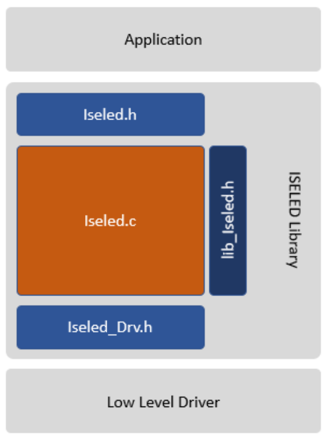
The gross data rate on the serial line is 2Mbit/s, i.e. each bit has a nominal duration of 500 ns. As the on-die oscillator has a very limited accuracy, the actual bit time may vary significantly. The whole system is designed for a maximum oscillator variance of ±30%. With the nominal oscillator frequency being 16 MHz, the actual frequency range is 11.2-20.8MHz.

The device directly attached to the host does not use the differential line mode on the upstream side. Instead a single-ended line mode with open-drain interface is used. The single-ended mode is intended to allow for an easy attachment to industry standard microcontrollers. Both single-ended lines require an external pull-up at the microcontroller to 5V.

During start-up the master interface detects single-ended or differential communication and enables termination to GND in case of differential mode. The slave interface operates differentially except for sleep mode. During initialization the slave interface physically checks for cable disconnects, respectively end of chain. In normal operation the ISELED software driver can regularly check for cable disconnects.

The protocol provides a set of control commands which are run length coded (RLC) and embedded in a serial frame structure."

2. Library Architecture



To use the ISELED library, the application needs to include the ISELED library APIs described in the Iseled.h file.

lib_Iseled.h contains all defines used in the ISELED library.

The ISELED library is running on top of a low-level serial driver such as SPI driver. As this driver can be adapted depending on the hardware/microcontroller used, it's not compiled with the ISELED library.

The low-level driver APIs are defined on Iseled_drv.h file embeds in the ISELED library.

To adapt ISELED library to any low-level serial driver, the Iseled_Drv.h APIs need to be coded.

3. Driver Protection

The first API calls by the application to initialize the ISELED library is digLED_Init_Driver function.

To avoid running the library on another chip, the digLED_Init_Driver performs a chip id check.
This check is based on:

- A CHIP_ID located on SCDS.CHIPIDxxxx registers.
- A list of mask (ISELED_CHIP_ID_MASK) applied on the CHIP_ID to select which bits are checked.
- A list of valid chip id (ISELED_CHIP_ID_VALID), refer to document "RH850/F1KM Hardware User's Manual – Addendum for ISELED" Tables 2.x for valid product part names).
- Note that the product part name is hard coded to the microcontroller. That means the ISELED library will only work with the dedicated devices with product part name as outlined in document "RH850/F1KM Hardware User's Manual – Addendum for ISELED".

4. Command Types

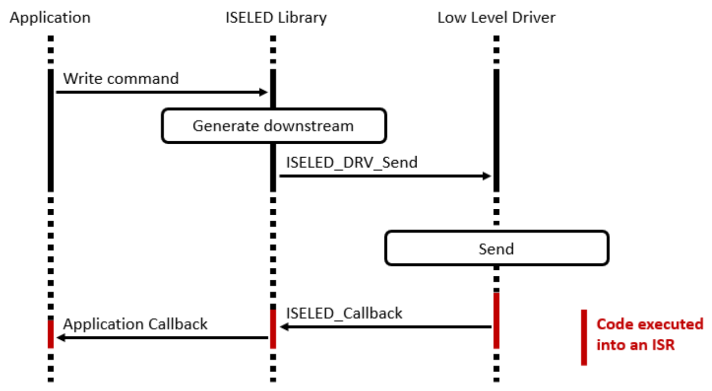
The protocol is divided into 4 command types:

- Write commands used to change attributes of LED(s). (i.e. RGB color, Max PWM, Temperature Offset, …)
- Read commands are used to get attributes of LED(s).
- Ping command used to get number of LED(s) on the strip.
- Init command used to initialize and enumerate all LED(s).

4.1 Write Command

Write command consists of sending an asynchronous downstream to the LED(s).

Write command APIs are a non-blocking function with flow describes below:



Every write command is structured like this:

- Check inactivity of driver
- Check arguments validity.
- Add arguments to specific command data payload.
- Generate downstream with Instruction, Address, Data payload, CRC option flag.
- Call low level send function with downstream.

When application calls a write command, an immediate status is returned in line with Inova Semiconductors specification:

- Status DIGLED_ERROR if library is already performing another command.
- Status DIGLED_ERROR if arguments are out of range.
- Status DIGLED_ERROR if an error occurred from low level driver.
- Status DIGLED_OK if no error.

The low level driver send function (ISELED_DRV_Send) is a non-blocking function.

At the end of sending, the low level driver call the ISELED_Callback function.

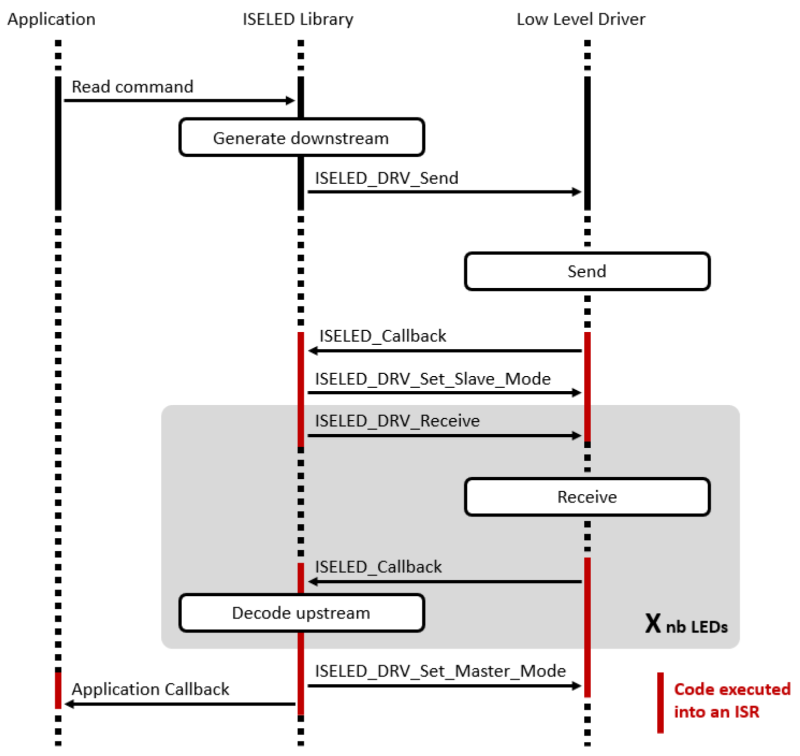
In ISELED_Callback function, the ISELED library calls the application callback (argument void(*Callback)(digLED_ReturnType event) of digLED_Init_Driver function).

The application callback returns a DIGLED_SUCCESS or DIGLED_ERROR status if a low-level driver error occurred during sending.

4.2 Read Command

Read command consists in a write command followed by synchronous receiving of upstream from each LED.

Read command APIs are a non-blocking function with flow describes below:



Every read command is structured like this:

- Check inactivity of driver.
- Check arguments validity.
- Add arguments to specific command data payload.
- Generate downstream with Instruction, Data payload, CRC option flag (Address is always broadcast)
- Call low level send function with downstream.
- Save information for asynchronous receiving.

When application calls a read command, an immediate status is returned in line with Inova Semiconductors specification:

- Status DIGLED_ERROR if library is already performing another command.
- Status DIGLED_ERROR if arguments are out of range.
- Status DIGLED_ERROR if an error occurred from low level driver.
- Status DIGLED_OK if no error.

The low-level driver send function (ISELED_DRV_Send) is a non-blocking function.

At the end of sending, the low-level driver call the ISELED_Callback function.

In ISELED_Callback function, the ISELED library changes the low-level driver to slave mode (ISELED_DRV_Set_Slave_Mode): clock for receiving upstream is now given by LED.

Then ISELED library calls low level receiving function (ISELED_DRV_Receive) to receive upstream from the last LED.

At end of receiving, the low-level driver calls the ISELED_Callback then ISELED library calls the next low level receiving function (ISELED_DRV_Receive) to receive upstream from the next LED.

When all LED upstream are received, the ISELED Library sets the low-level driver to master mode and calls the application callback (argument void(*Callback)(digLED_ReturnType event) of digLED_Init_Driver function).

The application callback returns a DIGLED_SUCCESS or DIGLED_ERROR status if a low-level driver error occurred during sending or receiving.

If low level driver receiving stays stuck, a timeout is fired (timeout value is set by ISELED API digLED_Set_Timeout) by the low-level driver that call the ISELED_Callback with a timeout error.

If this case, ISELED library calls the application callback (argument void(*Callback)(digLED_ReturnType event) of digLED_Init_Driver function) with DIGLED_ERROR status.

4.3 Ping Command

The ping command is performed like a read command but only the last LED upstream is received.

4.4 Init Command

The init command is performed like a read command but the address is defined by application.

This address defines the maximum LEDs initialized.

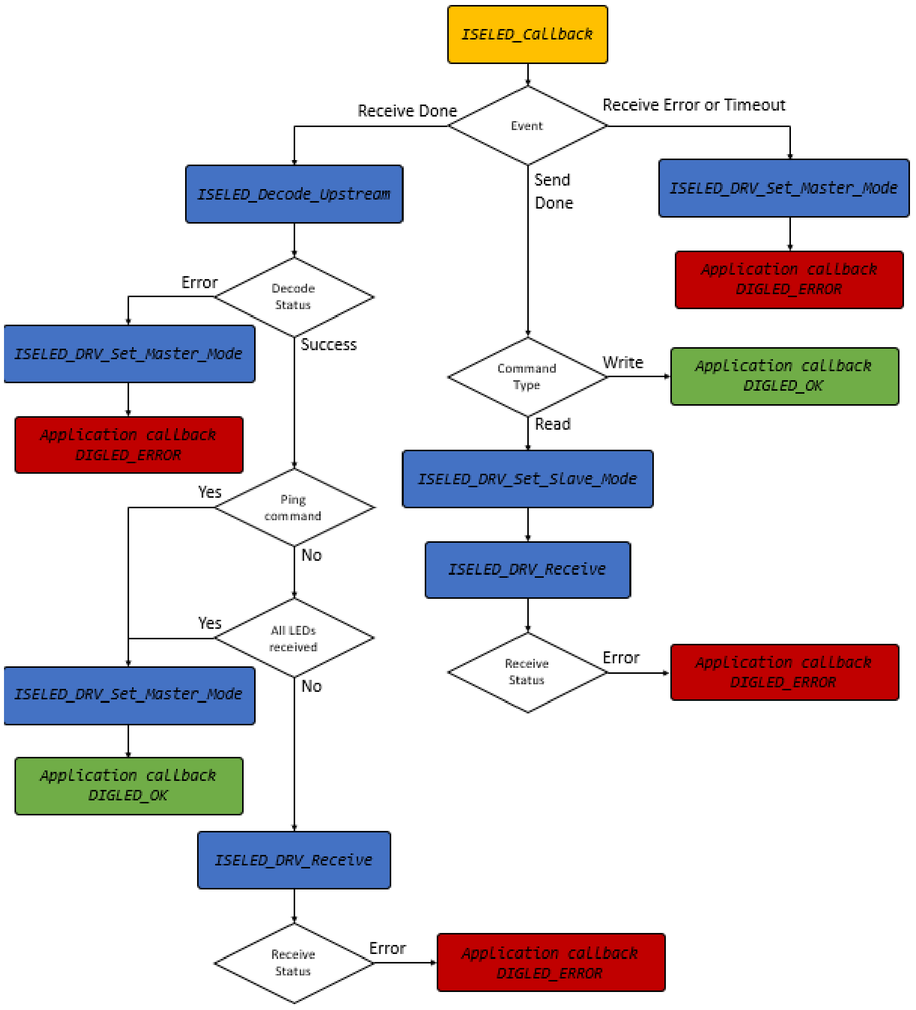
The initialization downstream and upstream is always with CRC.

5. ISELED Callback

When error, timeout, send complete or receive complete, the low-level driver calls the ISELED callback.

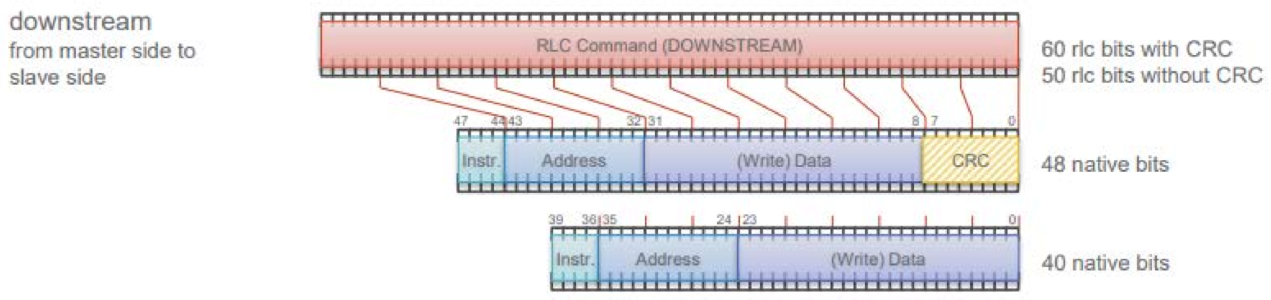
This callback raises status to the application but also manages multiple receives from LED(s).

The asynchronous mechanism implemented into the ISELED callback by the ISELED library is described below:


ISELED_Callback
Event
Receive Done
Receive Error or Timeout
Send Done
ISELED_Decode_Upstream
ISELED_DRV_Set_Master_Mode
Application callback DIGLED_ERROR
Decode Status
Error
Success
ISELED_DRV_Set_Master_Mode
Application callback DIGLED_ERROR
Command Type
Write
Read
Application callback DIGLED_OK
ISELED_DRV_Set_Slave_Mode
Ping command
Yes
No
ISELED_DRV_Receive
All LEDs received
Yes
No
Receive Status
Error
Application callback DIGLED_ERROR
ISELED_DRV_Set_Master_Mode
Application callback DIGLED_OK
ISELED_DRV_Receive
Receive Status
Error
Application callback DIGLED_ERROR

6. Downstream Generation

To ensure a proper bit clock recovery from the data stream, each 4 bits are encoded to 5 bits.



Due to the asynchronous transmission, a Frame-Sync byte and Freq-Sync byte are added before the downstream for start frame detection and frequency synchronization.

For frequency mismatch compensation, the protocol required more than 30% of frame length of idle bytes.

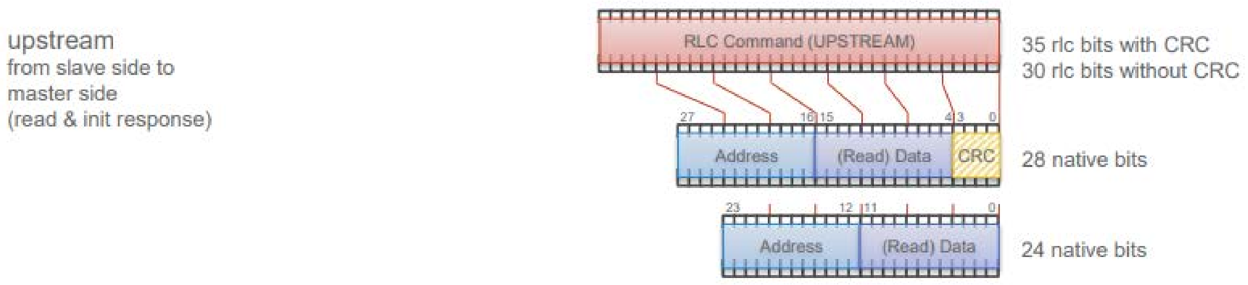
The downstream generation is structured like this:

- Set Frame-Sync and Freq-Sync
- Set Instruction with 4 to 5 bits conversion.
- Set Address with 4 to 5 bits conversion.
- Set Data with 4 to 5 bits conversion.
- If required, Compute and Set CRC with 4 to 5 bits conversion.
- Set idle bytes.
- Set additional idle byte for command block function.

The downstream generated has 14 bytes with CRC and 11 bytes without CRC.

7. Upstream Decode

Like the downstream, each 4 bits are encoded to 5 bits.



The upstream is a synchronous receive, but a Frame-Sync byte and a Freq-Sync byte are also in the upstream.

The upstream decoding is structured like this:

- Check Frame-Sync and Freq-Sync
- Get Address with 5 to 4 bits conversion.
- Get Data with 5 to 4 bits conversion.
- If required, Get CRC with 5 to 4 bits conversion.
- If required, Compute and compare with receive CRC.

The upstream received contains 5 bytes.

8. Command Block

To ease write command calls, a command block function is implemented.
This function permits to send multiple write commands with only one low level driver send call and consequently one ISELED callback call.

To do it, this API uses a list of blocks.
A block contains type of call, address, arguments and padding to match blocks size with downstream size.
To avoid RAM usage, this command API uses the same blocks to generate one multiple downstream.

The frame is sent to the low-level driver to perform only one send.

The command block is structured like this:

- Check inactivity of driver
- Check arguments validity for each block.
- For each block:
    - Add arguments to specific command data payload.
    - Generate downstream with Instruction, Address, Data payload, CRC option flag.
- Call low level driver send with a long payload with multiple downstream.

9. Low Level Driver Example

This section describes and shows an example of low-level driver.

This example doesn't represent the only way to code a low-level driver but gives some indications about restrictions and requirements of the low-level driver.

The easy way to perform the half-duplex bidirectional communication protocol of ISELED is to use a SPI serial interface.

This example of the low-level driver uses CSIH2 interface of the microcontroller.

The timeout feature is performed with the TAUB0 counter.

The ISELED library calls these 5 functions:

```
void                ISELED_DRV_Init          (void(*callback)(ISELED_DRV_ReturnType ret));
void                ISELED_DRV_Set_Master_Mode(void);
void                ISELED_DRV_Set_Slave_Mode (void);
ISELED_DRV_ReturnType ISELED_DRV_Send          (volatile uint8_t buffer[], uint32_t size);
ISELED_DRV_ReturnType ISELED_DRV_Receive       (volatile uint8_t buffer[], uint32_t size, uint32_t timeout);
```

with this type

```
typedef enum {
   ISELED_DRV_ERROR            = 0,
   ISELED_DRV_SUCCESS          = 1,
   ISELED_DRV_SEND_DONE        = 2,
   ISELED_DRV_RECIEVE_DONE     = 3,
   ISELED_DRV_RECIEVE_TIMEOUT  = 4
} ISELED_DRV_ReturnType;
```

9.1 ISELED_DRV_Init

The ISELED_DRV_Init function is called in the digLED_Init_Driver function.

This function stores the library callback reference and initialize the TAUB0 counter.

```
void ISELED_DRV_Init(void(*callback)(ISELED_DRV_ReturnType ret)) {
  uint8_t status = 0;
  (void)status; //Cast to avoid unused variable compilation warning

  /* Set callback */
  iseled_drv_callback = callback;

    /* Disable channel 0 counter operation */
    TAUB0.TT |= _TAUB_CHANNEL0_COUNTER_STOP;

    /* Disable INTTAUB0I0 operation and clear request */
    INTC2.ICTAUB0I0.BIT.MKTAUB0I0 = _INT_PROCESSING_DISABLED;
    INTC2.ICTAUB0I0.BIT.RFTAUB0I0 = _INT_REQUEST_NOT_OCCUR;

    /* Set INTTAUB0I0 setting */
    INTC2.ICTAUB0I0.BIT.TBTAUB0I0 = _INT_TABLE_VECTOR;
    INTC2.ICTAUB0I0.UINT16 &= _INT_PRIORITY_LEVEL1;
    TAUB0.TPS &= _TAUB_CK0_PRS_CLEAR;
    TAUB0.TPS |= _TAUB_CK0_PRE_PCLK_6;

    /* Set channel 0 setting */
    TAUB0.CMOR0 = _TAUB_SELECTION_CK0 | _TAUB_COUNT_CLOCK_PCLK | _TAUB_INDEPENDENT_CHANNEL |
_TAUB_SOFTWARE_TRIGGER |
                  _TAUB_OVERFLOW_AUTO_CLEAR | _TAUB_INTERVAL_TIMER_MODE | _TAUB_START_INT_NOT_GENERATED;

    /* Set compare match register */
    TAUB0.CMUR0 = _TAUB_INPUT_EDGE_UNUSED;
    TAUB0.CDR0 = ISELED_DRV_TIME_COMPARE_VALUE;

    /* Set output mode setting */
    TAUB0.TOE |= _TAUB_CHANNEL0_ENABLES_OUTPUT_MODE;
    TAUB0.TOM &= _TAUB_CHANNEL0_INDEPENDENT_OUTPUT_MODE;
    TAUB0.TOC &= _TAUB_CHANNEL0_OPERATION_MODE1;
    TAUB0.TOL &= _TAUB_CHANNEL0_POSITIVE_LOGIC;
    TAUB0.TDE &= _TAUB_CHANNEL0_DISABLE_DEAD_TIME_OPERATE;
    TAUB0.TDL &= _TAUB_CHANNEL0_POSITIVE_PHASE_PERIOD;

    /* Synchronization processing */
    status = TAUB0.TPS;
    __syncp();

    /* Clear INTTAUB0I0 request and enable operation */
    INTC2.ICTAUB0I0.BIT.RFTAUB0I0 = _INT_REQUEST_NOT_OCCUR;
    INTC2.ICTAUB0I0.BIT.MKTAUB0I0 = _INT_PROCESSING_ENABLED;

    /* Enable channel 0 counter operation */
    TAUB0.TS |= _TAUB_CHANNEL0_COUNTER_START;

    /* Set P11.4 as MISO pin */
    PORT.PMC11   |=  (1<<4);
    PORT.PIPC11  &= ~(1<<4);
    PORT.PM11    |=  (1<<4);
    PORT.PFCAE11 &= ~(1<<4);
    PORT.PFCE11  &= ~(1<<4);
    PORT.PFC11   &= ~(1<<4);}
```

9.2 ISELED_DRV_Set_Master_Mode

The send function is an asynchronous transfer, no clock pin is needed.

The ISELED communication protocol is bidirectional communication so CSIH2 output pin and CSIH2 input pins need to be connected to ISELED DATA pin of LED.

To ensure that the data line is high in idle, the function end by sending a 0xFF byte.

In ISELED library, at the end of receiving all LED(s) upstream, the library calls the ISELED_DRV_Set_Master_Mode function to take the lead on the ISELED DATA line.

However, between the configuration of CSIH2 in master mode and the sending of the 0xFF byte, the ISELED DATA line is set by the microcontroller to a low level.

If ISELED_DRV_Set_Master_Mode is called too late or the call is too long, the line is at low level when the LED is ready to receive a new downstream. This low level generates an Frame Sync Error on LED(s).

The function is structured like this:

- Stop CSIH2, clean and disable receive interruption.
- Configure CSIH2 as master mode.
- Disable CSIH2 Clock pin (P11.3)
- Disable CSIH2 Input pin (P11.4)
- Enable CSIH2 Output pin (P11.2)
- Clear and enable CSIH2 transmit interrupt.
- Send 0xFF.

```
void ISELED_DRV_Set_Master_Mode(void) {
  uint8_t status = 0;
  (void)status; //Cast to avoid unused variable compilation warning

  //----Stop Slave Operation---//
    /* Disable CSIH2 operation */
    CSIH2.CTL0 = _CSIH_OPERATION_CLOCK_STOP;
    /* Disable CSIH2 interrupt operation */
    INTC2.ICCSIH2IR.BIT.MKCSIH2IR = _INT_PROCESSING_DISABLED;
    INTC2.ICCSIH2IRE.BIT.MKCSIH2IRE = _INT_PROCESSING_DISABLED;
    INTC2.ICCSIH2IR.BIT.RFCSIH2IR = _INT_REQUEST_NOT_OCCUR;
    INTC2.ICCSIH2IRE.BIT.RFCSIH2IRE = _INT_REQUEST_NOT_OCCUR;

  //----Config Master Operation---//
    /* Set CSIH2 interrupt(INTCSIH2IC) setting */
    INTC2.ICCSIH2IC.BIT.TBCSIH2IC = _INT_TABLE_VECTOR;
    INTC2.ICCSIH2IC.UINT16 &= _INT_PRIORITY_LEVEL10;

    /* Set CSIH2 control setting */
    CSIH2.CTL1 = _CSIH_CLOCK_INVERTING_LOW | _CSIH_INTERRUPT_TIMING_TRANSFERRED |
_CSIH_DATA_CONSISTENCY_CHECK_DISABLE |
                 _CSIH_NO_DELAY | _CSIH_CHIPSELECT0_ACTIVE_HIGH | _CSIH_HANDSHAKE_DISABLE |
                 _CSIH_CHIPSELECT_SIGNAL_HOLD_INACTIVE | _CSIH_SLAVE_SELECT_DISABLE;

    CSIH2.CTL2 = ISELED_DRV_MASTER_CLOCK;
    CSIH2.BRS0 = ISELED_DRV_MASTER_BAUD_RATE;

    /* Set CSIH2 configuration setting */
    CSIH2.CFG0 = _CSIH_USED_BAUDRATE_0 | _CSIH_PARITY_NO | _CSIH_DATA_LENGTH_8 | _CSIH_DATA_DIRECTION_MSB |
_CSIH_PHASE_SELECTION_TYPE3 |
                 _CSIH_IDLE_INSERTED_NOT_ALWAYS | _CSIH_IDLE_TIME_0 | _CSIH_HOLD_TIME_0 |
                 _CSIH_INTER_DATA_DELAY_TIME_0 | _CSIH_SETUP_TIME_0;
    /* Synchronization processing */
    status = CSIH2.CTL1;
    __syncp();

  /* Set P11.2 as MOSI pin */
  PORT.PIBC11  &= ~(1<<2);
  PORT.PMC11   |=  (1<<2);
  PORT.PIPC11  &= ~(1<<2);
  PORT.PM11    &= ~(1<<2);
  PORT.PFCAE11 &= ~(1<<2);
  PORT.PFCE11  &= ~(1<<2);
  PORT.PFC11   &= ~(1<<2);

  /* Set P11.3 as input pin (no CLOCK) */
  PORT.PIBC11  |=  (1<<3);
  PORT.PMC11   &= ~(1<<3);
  PORT.PM11    |=  (1<<3);
  PORT.PFCAE11 &= ~(1<<3);
  PORT.PFCE11  &= ~(1<<3);
  PORT.PFC11   &= ~(1<<3);

  //----Start Master Operation---//
    /* Enable CSIH2 operation */
    CSIH2.CTL0 = _CSIH_OPERATION_CLOCK_PROVIDE | _CSIH_TRANSMISSION_PERMIT | _CSIH_RECEPTION_PROHIBIT |
_CSIH_DIRECTACCESS;
```

```
    /* Clear CSIH2 interrupt request and enable operation */
    INTC2.ICCSIH2IC.BIT.RFCSIH2IC = _INT_REQUEST_NOT_OCCUR;
    INTC2.ICCSIH2IC.BIT.MKCSIH2IC = _INT_PROCESSING_ENABLED;

    /* Set MOSI to high level */
    CSIH2.TX0W = ISELED_DRV_CHIP_BASE_DATA | 0xFF;
}
```

9.3 ISELED_DRV_Set_Slave_Mode

The receive is a synchronous transfer, the clock is given by ISELED LED(s).

To avoid collision on the ISELED DATA line, the CSIH2 output pin needs to be disable.

The function is structured like this:

- Enable CSIH2 Clock pin (P11.3)
- Enable CSIH2 Input pin (P11.4)
- Disable CSIH2 Output pin (P11.2)
- Stop CSIH2, clean and disable sending interruption.
- Configure CSIH2 as slave mode.
- Clear and enable CSIH2 reception interrupt.

```
void ISELED_DRV_Set_Slave_Mode(void) {
  uint8_t status = 0;
  (void)status; //Cast to avoid unused variable compilation warning

    /* Set P11.2 as input pin to release ISELED_DATA signal*/
  PORT.PIBC11  |=  (1<<2);
  PORT.PMC11   &= ~(1<<2);
  PORT.PM11    |=  (1<<2);
  PORT.PFCAE11 &= ~(1<<2);
  PORT.PFCE11  &= ~(1<<2);
  PORT.PFC11   &= ~(1<<2);

  /* Set P11.3 as CLOCK pin */
  PORT.PMC11   |=  (1<<3);
  PORT.PIPC11  &= ~(1<<3);
  PORT.PM11    |=  (1<<3);
  PORT.PFCAE11 &= ~(1<<3);
  PORT.PFCE11  &= ~(1<<3);
  PORT.PFC11   &= ~(1<<3);

  //----Stop Master Operation---//
    /* Disable CSIH2 operation */
    CSIH2.CTL0 = _CSIH_OPERATION_CLOCK_STOP;
    /* Disable CSIH2 interrupt operation */
  INTC2.ICCSIH2IC.BIT.MKCSIH2IC = _INT_PROCESSING_DISABLED;
    INTC2.ICCSIH2IC.BIT.RFCSIH2IC = _INT_REQUEST_NOT_OCCUR;

  //----Config Slave Operation---//
    /* Set CSIH2 interrupt(INTCSIH2IR) setting */
    INTC2.ICCSIH2IR.BIT.TBCSIH2IR = _INT_TABLE_VECTOR;
    INTC2.ICCSIH2IR.UINT16 &= _INT_PRIORITY_LEVEL9;

    /* Set CSIH2 interrupt(INTCSIH2IRE) setting */
    INTC2.ICCSIH2IRE.BIT.TBCSIH2IRE = _INT_TABLE_VECTOR;
    INTC2.ICCSIH2IRE.UINT16 &= _INT_PRIORITY_LEVEL9;

    /* Set CSIH2 control setting */
    CSIH2.CTL1 = _CSIH_HANDSHAKE_DISABLE | _CSIH_CHIPSELECT_SIGNAL_HOLD_ACTIVE | _CSIH_SLAVE_SELECT_DISABLE;
    CSIH2.CTL2 = ISELED_DRV_SLAVE_CLOCK;

    /* Set CSIH2 configuration setting */
    CSIH2.CFG0 = _CSIH_PARITY_NO | _CSIH_DATA_LENGTH_8 | _CSIH_DATA_DIRECTION_MSB | _CSIH_PHASE_SELECTION_TYPE3;

    /* Synchronization processing */
    status = CSIH2.CTL1;
    __syncp();

  //----Start Slave Operation---//
  /* Enable CSIH2 operation */
  CSIH2.CTL0 = _CSIH_OPERATION_CLOCK_PROVIDE | _CSIH_TRANSMISSION_PROHIBIT | _CSIH_RECEPTION_PERMIT |
_CSIH_DIRECTACCESS;

  /* Clear CSIH2 interrupt request and enable operation */
  INTC2.ICCSIH2IR.BIT.RFCSIH2IR = _INT_REQUEST_NOT_OCCUR;
  INTC2.ICCSIH2IRE.BIT.RFCSIH2IRE = _INT_REQUEST_NOT_OCCUR;
  INTC2.ICCSIH2IR.BIT.MKCSIH2IR = _INT_PROCESSING_ENABLED;
  INTC2.ICCSIH2IRE.BIT.MKCSIH2IRE = _INT_PROCESSING_ENABLED;
}
```

9.4 ISELED_DRV_Send

The ISELED_DRV_Send is a non-blocking function. In this example the driver uses interrupt mode.
The function checks if low level driver is ready and sends the first byte of the payload.
Next bytes will be sent in the ISELED_DRV_Master_Isr bound to the transmit complete IRQ of CSIH2.
When all bytes are sent, ISELED_DRV_Master_Isr calls the library callback with send done status.

```
ISELED_DRV_ReturnType ISELED_DRV_Send(uint8_t buffer[], uint32_t size) {
  uint8_t status = 0;
  (void)status; //Cast to avoid unused variable compilation warning

  if (iseled_drv_spiUsed) {
    return ISELED_DRV_ERROR;
  }

  while(CSIH2.STR0 & (1<<7));  /* Wait until transmission finished */

  iseled_drv_spiSize = size;
  iseled_drv_spiBuffer = buffer;
  iseled_drv_spiIndex = 0;
  iseled_drv_spiUsed = 1;

  /* Set transmit data */
  CSIH2.TX0W = ISELED_DRV_CHIP_BASE_DATA | iseled_drv_spiBuffer[iseled_drv_spiIndex];
  iseled_drv_spiIndex++;

  return ISELED_DRV_SUCCESS;
}

#pragma interrupt ISELED_DRV_Master_Isr(enable=false, channel=132, fpu=true, callt=false)
void ISELED_DRV_Master_Isr(void) {
  if (iseled_drv_spiUsed == 0) {
    return;
  }
  if (iseled_drv_spiIndex < iseled_drv_spiSize) {
    CSIH2.TX0W = ISELED_DRV_CHIP_BASE_DATA | iseled_drv_spiBuffer[iseled_drv_spiIndex];
    iseled_drv_spiIndex++;
  } else {
    iseled_drv_spiUsed = 0;
    iseled_drv_callback(ISELED_DRV_SEND_DONE);
  }
}
```

9.5 ISELED_DRV_Receive

The ISELED_DRV_Receive must be a non-blocking receive. In this example the driver uses interrupt mode.

The function checks if low level driver is ready and set timeout.

Each received byte will be stored by the ISELED_DRV_Slave_Isr bound to the reception complete IRQ of CSIH2.

When all bytes are received, ISELED_DRV_Slave_Isr calls the library callback with a receive done status.

An ISELED_DRV_Slave_Error_Isr is bound to the reception error IRQ of CSIH2.

If an error occured, ISELED_DRV_Slave_Error_Isr calls the library callback with an error status.

```
ISELED_DRV_ReturnType ISELED_DRV_Receive(uint8_t buffer[], uint32_t size, uint32_t timeout) {
  uint32_t time;

  if (iseled_drv_spiUsed) {
    return ISELED_DRV_ERROR;
  }
  if (timeout > ISELED_DRV_TIMEOUT_MAX_VALUE) {
    return ISELED_DRV_ERROR;
  }

  iseled_drv_spiSize = size;
  iseled_drv_spiIndex = 0;
  iseled_drv_spiBuffer = buffer;
  iseled_drv_spiUsed = 1;

  if (timeout == 0) {
    iseled_drv_timeout = 0;
  } else {
    ISELED_DRV_GetCurrentTime(&time);
    iseled_drv_timeout = time + timeout*1000; //timeout in seconds
  }

  return ISELED_DRV_SUCCESS;
}

#pragma interrupt ISELED_DRV_Slave_Isr(enable=false, channel=133, fpu=true, callt=false)
void ISELED_DRV_Slave_Isr(void) {

  if (iseled_drv_spiUsed == 0) {
    return;
  }

    iseled_drv_spiBuffer[iseled_drv_spiIndex] = CSIH2.RX0H;
    iseled_drv_spiIndex++;
  if (iseled_drv_spiIndex >= iseled_drv_spiSize) {
    iseled_drv_spiUsed = 0;
        iseled_drv_timeout = 0;
    iseled_drv_callback(ISELED_DRV_RECIEVE_DONE);
  }
}

#pragma interrupt ISELED_DRV_Slave_Error_Isr(enable=false, channel=134, fpu=true, callt=false)
void ISELED_DRV_Slave_Error_Isr(void) {
  uint8_t dummy = 0;
  (void)dummy; //Cast to avoid unused variable compilation warning

    CSIH2.STCR0 |= (_CSIH_CONSISTENCY_ERROR_CLEAR | _CSIH_PARITY_ERROR_CLEAR | _CSIH_OVERRUN_ERROR_CLEAR);
    dummy = CSIH2.RX0W;
    iseled_drv_spiUsed = 0;
    iseled_drv_timeout = 0;
    iseled_drv_callback(ISELED_DRV_ERROR);
}
```

9.6 Timeout

The timeout is checked into the TAUB0 interruption routine ISELED_DRV_Time_Isr.
Every millisecond, if a receiving is ongoing with a timeout, the timeout is checked.
If the timeout is reached, the ISELED_DRV_Time_Isr calls the library callback with a receive timeout status.

If the TAUB0 counter reaches the maximum time, the TAUB0 counter is disabled to avoid overflow.

```
#pragma interrupt ISELED_DRV_Time_Isr(enable=false, channel=142, fpu=true, callt=false)
void ISELED_DRV_Time_Isr(void) {
  uint8_t status = 0;
  (void)status; //Cast to avoid unused variable compilation warning

  iseled_drv_timeMs++;

  if (iseled_drv_timeMs >= ISELED_DRV_TIMEOUT_MAX_VALUE_MS) {

      /* Disable channel 0 counter operation */
      TAUB0.TT |= _TAUB_CHANNEL0_COUNTER_STOP;

      /* Disable INTTAUB0I0 operation and clear request */
      INTC2.ICTAUB0I0.BIT.MKTAUB0I0 = _INT_PROCESSING_DISABLED;
      INTC2.ICTAUB0I0.BIT.RFTAUB0I0 = _INT_REQUEST_NOT_OCCUR;

      /* Synchronization processing */
      status = TAUB0.TT;
      __syncp();
  }

  if (iseled_drv_timeout != 0) {
    if (iseled_drv_timeMs >= iseled_drv_timeout) {
        iseled_drv_spiUsed = 0;
        iseled_drv_timeout = 0;
        iseled_drv_callback(ISELED_DRV_RECIEVE_TIMEOUT);
    }
  }
}
```

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	May 07, 2024	-	Initial release

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit: www.renesas.com/contact/.