The following document contains information on Cypress products. The document has the series name, product name, and ordering part numbering with the prefix “MB”. However, Cypress will offer these products to new and existing customers with the series name, product name, and ordering part number with the prefix “CY”.

How to Check the Ordering Part Number

1. Go to www.cypress.com/pcn.
2. Enter the keyword (for example, ordering part number) in the **SEARCH PCNS** field and click **Apply**.
3. Click the corresponding title from the search results.
4. Download the Affected Parts List file, which has details of all changes

For More Information

Please contact your local sales office for additional information about Cypress products and solutions.

About Cypress

Cypress is the leader in advanced embedded system solutions for the world's most innovative automotive, industrial, smart home appliances, consumer electronics and medical products. Cypress' microcontrollers, analog ICs, wireless and USB-based connectivity solutions and reliable, high-performance memories help engineers design differentiated products and get them to market first. Cypress is committed to providing customers with the best support and development resources on the planet enabling them to disrupt markets by creating new product categories in record time. To learn more, go to www.cypress.com.

FM4 Peripheral Driver Library, User Guide

Doc. No. 002-04460 Rev. *A

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
http://www.cypress.com

Copyrights

© Cypress Semiconductor Corporation, 2015-2016. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Trademarks

All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Source Code

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer

CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.

Contents

1. Introduction **8**

1.1 About Document 8
1.2 About PDL 8

2. PDL Structure **9**

2.1 File System 9
2.2 Characteristics 10
2.3 Graphical Example 10
2.4 Interrupt Flow 12

3. Using PDL in own Application Projects **13**

3.1 Use PDL Template as Base 13

4. Using the PDL Examples **14**

5. Getting started **15**

5.1 Adjustment of FM4 Device 15
5.1.1 FM4 Device Family 15
5.1.2 FM4 Device Package 15
5.1.3 Possible Compile Error 16
5.2 PDL User Settings 16
5.2.1 Peripheral Usage 16
5.2.2 Interrupts 16
5.3 Combination of Drivers 17
5.3.1 Example: ADC and DMA 17
5.4 Recommendations 17
5.4.1 Configuration 17

6. PDL Main Files **18**

6.1 File: base_types.h 18
6.1.1 Type Definitions 18
6.1.2 PDL Return Value Type (en_result_t) 18
6.2 File: pdl.c 19
6.3 File: pdl.h 20
6.4 File: pdl_user.h 22
6.5 File: interrupts.c 22
6.6 File: interrupts.h 23

7. **PDL Resource Driver API (Low Level) 24**

7.1 Preface 24
7.2 (ADC) Analog Digital Converter Module 26
7.2.1 Configuration Structure 26
7.2.2 ADC API 28
7.2.3 ADC Examples 38
7.3 (BT) Base Timer 38
7.3.1 Configuration Structure 42
7.3.2 BT API 47
7.4 (CAN) Controller Area Network 63
7.4.1 Configuration Structure 63
7.4.2 CAN API 64
7.4.3 CAN Examples 70
7.5 (CLK) Clock Module 70
7.5.1 Main Clock Configuration 71
7.5.2 Sub Clock Configuration 73
7.5.3 VBAT Domain Configuration 74
7.5.4 CLK API 75
7.5.5 CLK Examples 88
7.6 (CR) CR Clock Trimming 88
7.6.1 Configuration Structure 89
7.6.2 CR API 89
7.7 (CRC) Cyclic Redundancy Check 90
7.7.1 CRC Configuration Structure 91
7.7.2 API Reference 91
7.7.3 Example Code 92
7.8 (CSV) Clock Supervisor 94
7.8.1 Configuration Structure 94
7.8.2 CSV API 95
7.9 (DAC) Digital Analog Converter 99
7.9.1 Configuration Structure 100
7.9.2 DAC API 100
7.9.3 DAC Example 102
7.10 (DMA) Direct Memory Access 102
7.10.1 Configuration Structure 104
7.10.2 DMA API 105
7.10.3 DMA Example 107
7.11 (DSTC) Descriptor System Data Transfer Controller 107
7.11.1 Configuration Structure 108
7.11.2 DSTC DES Structures 110
7.11.3 DSTC API 114
7.11.4 DSTC Examples 123
7.12 (DT) Dual Timer 123
7.12.1 DT Configuration Structure 124
7.12.2 API Reference 124
7.12.3 Example Code 128

7.13 (EXINT) External Interrupts / NMI 132
7.13.1 Configuration Structure (External Interrupts) 132
7.13.2 Configuration Structure (NMI) 132
7.13.3 EXINT API 132
7.13.4 EXINT Examples 134
7.14 (EXTIF) External Bus Interface 135
7.14.1 Configuration Structure 135
7.14.2 EXTIF API 138
7.14.3 EXTIF Examples 140
7.15 (FLASH) Flash Memory 140
7.15.1 Main Flash 140
7.15.2 Work Flash 142
7.16 (GPIO) General Purpose I/O Ports 144
7.16.1 GPIO Macro API 144
7.16.2 GPIO additional Macros 146
7.16.3 GPIO Examples 147
7.17 (HWWDG) Hardware Watch Dog 148
7.17.1 HWWDG Configuration Structure 148
7.17.2 API reference 149
7.17.3 Example Code 152
7.18 (LPM) Low Power Modes 152
7.18.1 LPM API 153
7.18.2 LPM Example 160
7.19 (LVD) Low Voltage Detection 160
7.19.1 LVD Configuration Structure 161
7.19.2 API Reference 162
7.19.3 Example Code 163
7.20 (MFT) Multi Function Timer 166
7.20.1 FRT (Free-run Timer Unit) 166
7.20.2 OCU (Output Compare Unit) 175
7.20.3 WFG (Waveform Genarator Unit) 183
7.20.4 ICU (Input Capture Unit) 192
7.20.5 ADCMP (ADC Start Compare Unit) 197
7.21 (MFS_HL) Multi Function Serial Interface High Level 204
7.21.1 The usable mode in the Multi Function Serial Interface 206
7.21.2 Ring Buffer Principle 207
7.21.3 MFS_HL Configuration Structure 207
7.21.4 API Reference 213
7.21.5 Example Software 226
7.22 (MFS) Multi Function Serial Interface 270
7.22.1 MFS Configuration Structure 271
7.22.2 API Reference 275
7.22.3 Example Code 322
7.23 (PPG) Programmable Pulse Generator 425
7.23.1 Configuration Structure 426
7.23.2 PPG API 429
7.24 (QPRC) Quad Position and Revolution Counter 437

7.24.1 Configuration Structure 438
7.24.2 QPRC API 442
7.25 (RESET) Reset Cause 449
7.25.1 RESET API 449
7.25.2 RESET Example 450
7.26 (RTC) Real Time Clock 451
7.26.1 RTC Configuration Structures 452
7.26.2 RTC Definitions 456
7.26.3 API Reference 457
7.26.4 Example Code 466
7.27 (SD) SD Card Inerface 472
7.27.1 Configuration Structure 472
7.27.2 SD Card API 474
7.28 (SWWDG) Software Watchdog 478
7.28.1 SWWDG Configuration Structure 479
7.28.2 API reference 480
7.28.3 Example Code 482
7.29 (UID) Unique ID 486
7.29.1 UID API 486
7.29.2 UID Example 487
7.30 (WC) Watch Counter 487
7.30.1 Configuration Structure 488
7.30.2 WC API 489

8. Revision History 495

Document Revision History 495

Target Products

This application note is described about below products;

Series	Product Number (not included Package suffix）
MB9B160R	MB9BF166M/N/R, MB9BF167M/N/R, MB9BF168M/N/R
MB9B360R	MB9BF366M/N/R, MB9BF367M/N/R, MB9BF368M/N/R
MB9B460R	MB9BF466M/N/R, MB9BF467M/N/R, MB9BF468M/N/R
MB9B560R	MB9BF566M/N/R, MB9BF567M/N/R, MB9BF568M/N/R

1. Introduction

1.1 About Document

This document describes how to use the Peripheral Driver Library (Named PDL in the following text). It describes the API for each supported peripheral in detail and gives advises, if necessary.

1.2 About PDL

The PDL was designed to ease the use of the peripherals of the FM4 MCU series. The user does not need to know about the register and bit structure of those peripherals. Mainly he only needs to set up a configuration and make use of the API functions, such as initialization of peripheral, etc.

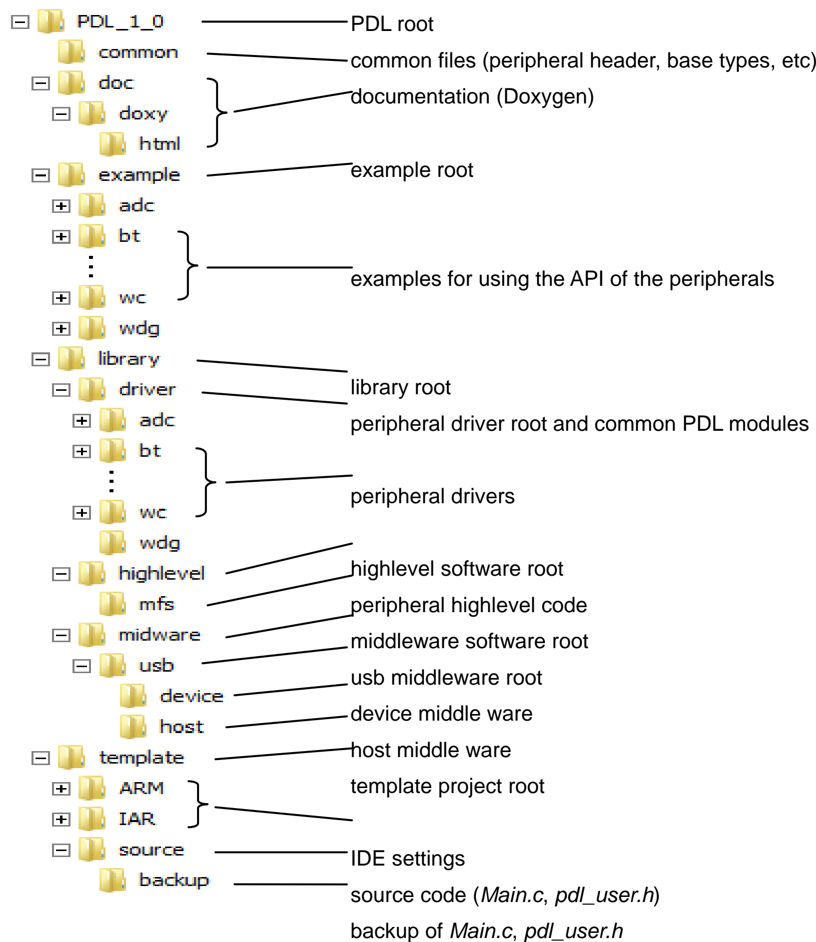
For most of the peripherals several example code exists, which is separated from the library, so that the user can make use of the PDL for own projects without these examples.

The recent version of the PDL is 1.0.

2. PDL Structure

2.1 File System

The PDL structure consists of the following directories and sub directories:



To use the PDL for own projects the user only have to copy the *library* and *common* directories and their sub directories. A special user setting header file pdl_user.h shall be placed in the software root of his project, where the *Main.c* module is located, often the *source* directory.

The same appears to *highlevel* and *midware* directories, if these software drivers are needed for the user's project

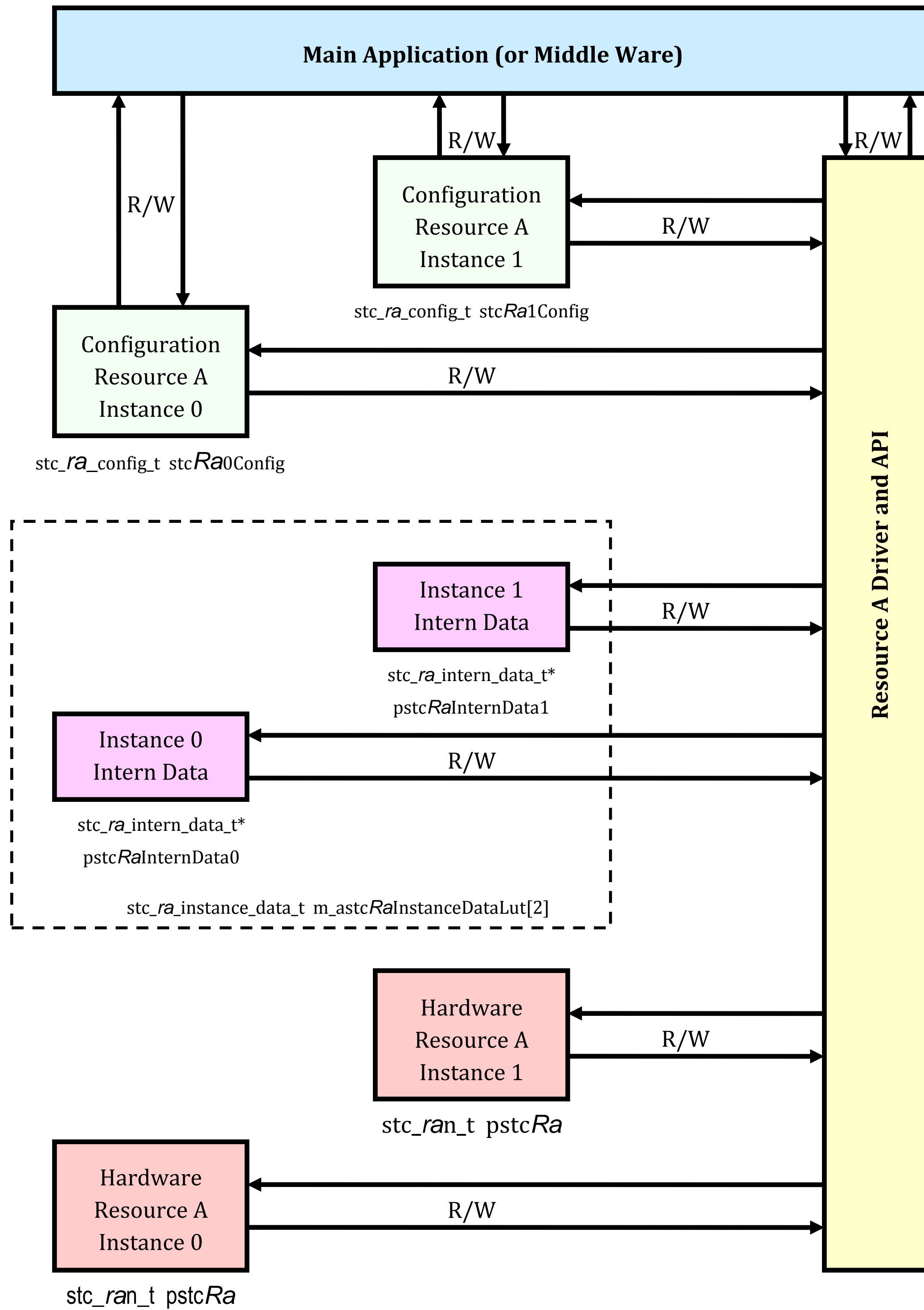
2.2 Characteristics

The PDL functions for the resources have to following characteristics:

- Each function of a resource driver is existing only once in the ROM (Flash) memory regardless of the number of resource instances. Internal data handles the instance configuration.
- Most of the resource functions use a configuration data structure per instance, which has to be configured in the user application.
- Most API (external) functions have a error code return value (at least just `en_result_t Ok`).
- Each driver of a resource, which has an enable bit, provides an Init and De-Init function.
- If the resource driver uses interrupts, the interrupt service routine and handling is part of the driver. If it is reasonable, callbacks to a user function are provided.
- If a resource is not used (by no activation in *l3_user.h*) its driver code is not compiled and linked, regardless of the driver module being a member of the project workspace.

2.3 Graphical Example

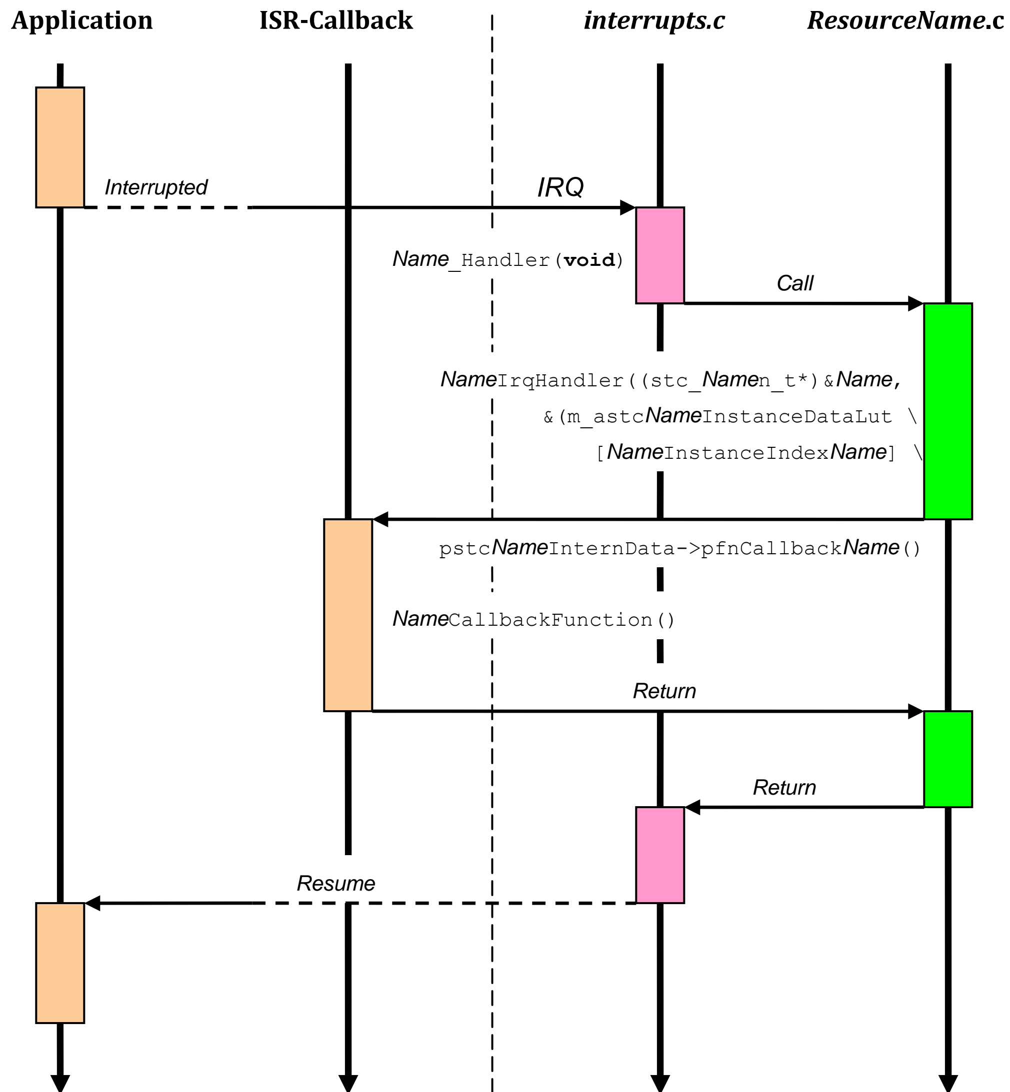
The following graphic illustrates a driver for resource A (*Ra*) which has 3 active instances.



Main Application (or Middle Ware)
R/W
R/W
R/W
Configuration
Resource A
Instance 1
R/W
stc_ra_config_t stcRa1Config
Configuration
Resource A
Instance 0
R/W
stc_ra_config_t stcRa0Config
Resource A Driver and API
Instance 1
Intern Data
R/W
stc_ra_intern_data_t*
pstcRaInternData1
Instance 0
Intern Data
R/W
stc_ra_intern_data_t*
pstcRaInternData0
stc_ra_instance_data_t m_astcRaInstanceDataLut[2]
Hardware
Resource A
Instance 1
R/W
stc_ran_t pstcRa
Hardware
Resource A
Instance 0
R/W
stc_ran_t pstcRa

2.4 Interrupt Flow

The following flow shows how the interrupts internally are handled in the PDL:



The user does not need to take care of clearing any interrupt, because this is automatically done in the driver's service routine. A so-called callback function is called to notify the application about an occurred interrupt, if configured and needed.

The PDL also controls the shared vector interrupts and always executes the actual interrupt service routine.

Some callback functions need to have argument (e.g. for errors) to notify a certain status via these arguments. This is explained in the API section of this manual.

3. Using PDL in own Application Projects

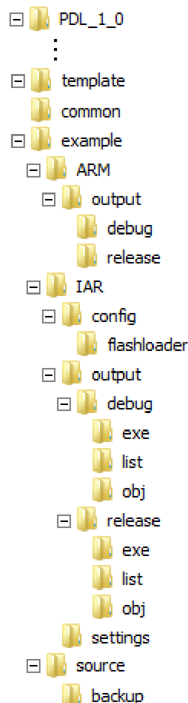
3.1 Use PDL Template as Base

The PDL project contains a sub folder *template*. In this folder the user will find the same directory structure as he is used by the usual Cypress FM4 template projects.

The template folder contains also the usual predefined settings for the following tool chains:

- KEIL/ARM: µVision
- IAR: Embedded Workbench

The directory structure is as follows:



The template project also contains all necessary linker settings and preprocessor search directories for the supported tool chains.

For debugging the J-Link for IAR and ULink-ME for KEIL/ARM is set. The user may choose a different JTAG adapter.

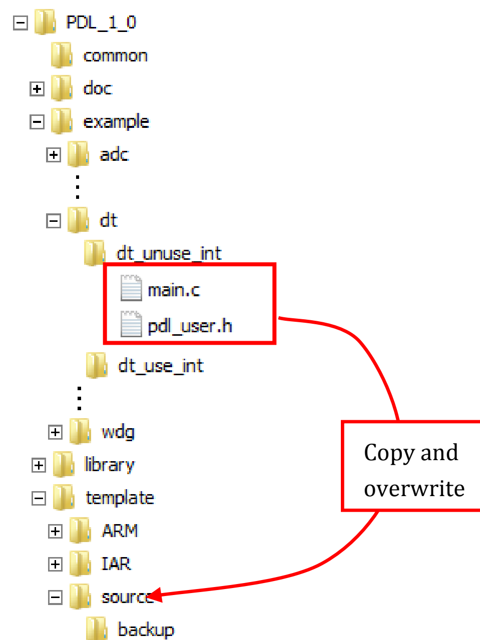
4. Using the PDL Examples

The PDL project contains a plenty of examples for the supported driver parts, such as for MFS, MFT, etc.

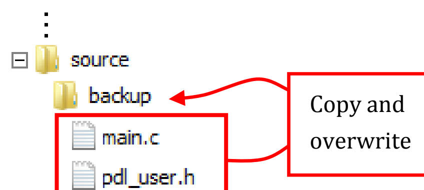
To use and build a dedicated example of the *example/<peripheral>* folders the contained files of the example (mainly *main.c* and *pdl_user.h*) should be copied to the *source* folder of the *template* directory. Overwrite *main.c* and *pdl_user.h* for this. These files can be restored by those contained in the *backup* folder.

The module *main.c* contains the example code how to use the driver. The header file *pdl_user.h* contains all necessary adjustments for using the driver peripheral and other settings.

How to use e.g. the Dual Timer's example without interrupt usage:



How to restore the original template modules:



5. Getting started

5.1 Adjustment of FM4 Device

In the file *pdl_device.h* contained in the subfolder *library/driver* there are two definitions for the used FM4 derivative and the package.

The choice list for supported devices can be found in the file *pdl.h* in the same folder, which shall not be changed by the user.

By the following user definitions the PDL extracts the following information:

- FM4 Device Type
- Correct GPIO inclusion
- Several peripheral feature support, which are different in some Device Types

5.1.1 FM4 Device Family

The definition of the user's FM4 device family looks as follows:

```
#define PDL_MCU PDL_DEVICE_TYPE_MB9BF56X
```

Note that the Flash memory size is not considered by the PDL. This is the reason for the placeholder `x` in the definition of the device.

5.1.2 FM4 Device Package

The definition of the user's FM4 device package looks as follows:

```
#define PDL_PACKAGE PDL_DEVICE_PACKAGE_R
```

The PDL checks on compile time whether the combination of the device and its package is existing. The package is very important for the GPIO usage of the PDL.

5.1.3 Possible Compile Error

If the combination of the device and its package does not exist or is not supported by the PDL, the following error message is displayed during compile time:

```
Error: Device Type not defined!
```

5.2 PDL User Settings

5.2.1 Peripheral Usage

To save Flash and RAM memory, each peripheral driver can be enabled or disabled. If a peripheral driver is disabled, no code is compiled for this peripheral. Therefore the user has to enable individually his needs of the used peripherals.

For this reason the header file *pdl_user.h* exists. This file can be found in the *template/source* directory.

Example for enabling `ADC0` and `ADC1` in *pdl_user.h*:

```
// ADC
#define PDL_PERIPHERAL_ENABLE_ADC0              PDL_ON
#define PDL_PERIPHERAL_ENABLE_ADC1              PDL_ON
#define PDL_PERIPHERAL_ENABLE_ADC2              PDL_OFF
```

The rule for enabling and disabling a peripheral instance:

- Peripheral instance enable: `PDL_ON`
- Peripheral instance disable: `PDL_OFF`

5.2.2 Interrupts

There are two steps to use the PDL's interrupts. The first is the general enable and the second the interrupt level.

Example for enabling `ADC0`'s interrupt:

```
// ADC
#define PDL_INTERRUPT_ENABLE_ADC0              PDL_ON
```

There is one exception: If the external interrupt peripheral is enabled the according interrupts are enabled automatically and do not need to be enabled by the user.

The interrupt level e.g. for `ADC0` can be set as shown below:

```
// ADC
#define PDL_IRQ_LEVEL_ADC0              7u
```

Possible values are from `0u` (highest priority) to `15u` (lowest priority).

5.3 Combination of Drivers

Some settings of the resource configuration require to activate an additional driver module such as external interrupts or DMA usage. In this case the user has to do this manually in *pdl_user.h* and has to define the configuration and has to take care of correct initialization of all drivers in his application.

5.3.1 Example: ADC and DMA

In this case the ADC has to be initialized and configured for DMA transfer. The user also has to activate manually the DMA module and to configure a desired channel to be used. The ADC driver relies on the proper DMA configuration.

5.4 Recommendations

5.4.1 Configuration

To save RAM memory it is useful to put the initialization of a peripheral with its configuration to an own function, in which this configuration is declared as a local structure.

Because this local structure is not initialized to zero at start-up, use the zero-init macro `PDL_ZERO_STRUCT(`*struct_name*`)`.

Example:

```
en_result_t function()
{
    // Local variables
    stc_adcn_t*        pstcAdc = (stc_adcn_t*)&ADC0;
    stc_adc_config_t stcAdcConfig;

    PDL_ZERO_STRUCT(stcAdcConfig);

    stcAdcConfig.u32CannelSelect.AD_CH_0 = 1;
    stcAdcConfig.bLsbAlignment              = TRUE;
    stcAdcConfig.enMode                      = SingleConversion;

    . . .

    return Adc_Init((volatile stc_adcn_t*) pstcAdc, &stcAdcConfig);
}
```

6. PDL Main Files

6.1 File: base_types.h

In the *base_types.h* file the (macro) definitions for `TRUE`, `FALSE`, `MIN(X,Y)`, `MAX(X,Y)`, etc. are done. This file should not be changed by the user.

6.1.1 Type Definitions

Additional types such as `boolean_t, float32_t`, etc. are done additionally to the CMSIS types.

6.1.2 PDL Return Value Type (en_result_t)

In the *base_types.h* file the PDL global return or result enumeration is done. For all PDL functions, which have a return value, which is not a certain value (e.g. read value or status number), this type shall be used.

Example of the result enumeration type definition:

```
typedef enum en_result
{
    Ok                          = 0,  ///< No error
    Error                       = 1,  ///< Non-specific error code
    ErrorAddressAlignment       = 2,  ///< Address alignment does not match
    ErrorAccessRights           = 3,  ///< Wrong mode (e.g. user/system) mode is set
    ErrorInvalidParameter       = 4,  ///< Provided parameter is not valid
    ErrorOperationInProgress    = 5,  ///< A conflicting or requested operation is still in
progress
    ErrorInvalidMode            = 6,  ///< Operation not allowed in current mode
    ErrorUninitialized          = 7,  ///< Module (or part of it) was not initialized properly
    ErrorBufferFull             = 8,  ///< Circular buffer can not be written because the
buffer is full
    ErrorTimeout                = 9,  ///< Time Out error occurred
    ErrorNotReady               = 10, ///< A requested final state is not reached
    OperationInProgress         = 11  ///< Indicator for operation in progress
} en_result_t;
```

6.2 File: pdl.c

This file contains all global helper functions.

Recently only a hook function is provided, which is called in any polling loop and can be used for setting user actions such as feeding a watchdog.

```
void PDL_WAIT_LOOP_HOOK(void)
{
    // Place code for triggering Watchdog counters here, if needed
}
```

Note, that for RAM code functions such as Flash programming routines the user code also has to be set in *ramcode.c's* function.

`static void PDL_RAMCODE_WAIT_LOOP_HOOK(void).`

The *pdl.c* module also contains a global function to zero-initialize local structures.

This function may also be used by the user.

```
void pdl_memclr(uint8_t* pu32Address, uint32_t u32Count)
{
    while(u32Count--)
    {
        *pu32Address++ = 0;
    }
}
```

This function is called by the PDL's macro

```
#define PDL_ZERO_STRUCT(x) pdl_memclr((uint8_t*)&(x), (uint32_t)(sizeof(x)))
```

which is set in *pdl.h*.

Example for usage:

```
function()
{
    // Local variables
    stc_adc_config_t stcAdcConfig;
         ...
    PDL_ZERO_STRUCT(stcAdcConfig);
}
```

6.3 File: pdl.h

After `PDL_ZERO_STRUCT()` definition the *pdl.h* file defines the PDL On/Off switch values, the device type, the FM4 device names, and the package code. This file ***shall not*** be changed by the user.

The PDL's On/Off definition:

```
#define PDL_ON              1u    ///< Switches a feature on
#define PDL_OFF             0u    ///< Switches a feature off
```

The Device Types are coded with the number itself:

```
#define PDL_TYPE0 0u  ///< FM4 device type0
#define PDL_TYPE1 1u  ///< FM4 device type1
#define PDL_TYPE2 2u  ///< FM4 device type2
#define PDL_TYPE3 3u  ///< FM4 device type3
                  . . .
```

After this the FM4 derivative families follows:

```
#define PDL_DEVICE_TYPE_MB9BF56X  50u
                  . . .
```

The next section defines the packages:

```
#define PDL_DEVICE_PACKAGE_K  10u
#define PDL_DEVICE_PACKAGE_L  20u
#define PDL_DEVICE_PACKAGE_M  30u
#define PDL_DEVICE_PACKAGE_N  40u
#define PDL_DEVICE_PACKAGE_R  50u
#define PDL_DEVICE_PACKAGE_S  60u
#define PDL_DEVICE_PACKAGE_T  70u
```

Now the device predefinitions are set and in the next section the *pdl_user.h* is included.

After this the device and package check is done to determine the Device Type:

```
#if (PDL_DEVICE_MB9BF56X == PDL_ON)
  #if  (PDL_PACKAGE == PDL_DEVICE_PACKAGE_M) || \
        (PDL_PACKAGE == PDL_DEVICE_PACKAGE_N) || \
        (PDL_PACKAGE == PDL_DEVICE_PACKAGE_R)
    #define PDL_DEVICE_TYPE PDL_TYPE0
  #else
    #error Device Type not defined!
  #endif
         . . .
#else
  #error Device not found!
#endif
```

Now the PDL's interrupt default level is defined (lowest priority):

```
#define PDL_DEFAULT_INTERRUPT_LEVEL 0x0Fu
```

After this the preprocessor code checks the peripheral activation, sets this activation for the driver modules and include the driver header files.

```
// Activate code in adc.c if one or more are set to PDL_ON
#if (PDL_PERIPHERAL_ENABLE_ADC0 == PDL_ON) || \
    (PDL_PERIPHERAL_ENABLE_ADC1 == PDL_ON) || \
    (PDL_PERIPHERAL_ENABLE_ADC2 == PDL_ON)
    #define PDL_PERIPHERAL_ADC_ACTIVE
    #include "adc\adc.h"
#endif
         . . .
```

The next section defines the PDL's resource driver availability, e.g.:

```
#define PDL_PERIPHERAL_ADC_AVAILABLE     PDL_ON
```

After this some user parameter checks are done:

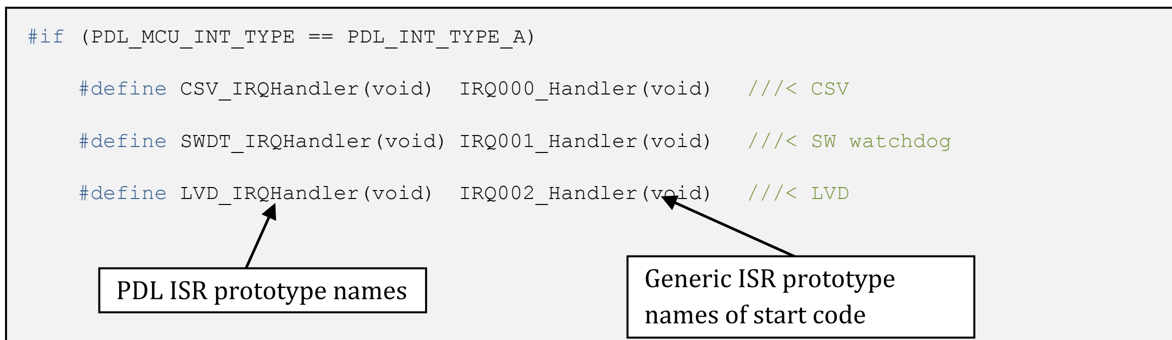
```
#if (PDL_NO_FLASH_RAMCODE != PDL_ON) && (PDL_NO_FLASH_RAMCODE != PDL_OFF)
  #error PDL_NO_FLASH_RAMCODE in pdl_user.h is not defined as either PDL_ON or PDL_OFF!
#endif

#if ((PDL_DMA_CHANNELS < 1u) || (PDL_DMA_CHANNELS > 8u))
  #error PDL_DMA_CHANNELS in pdl_user.h value out of range!
#endif
```

The next section defines the availability of Deep Standby Modes depending of the Device Type:

```
#if ((PDL_DEVICE_TYPE == PDL_TYPE0) || `
     (PDL_DEVICE_TYPE == PDL_TYPE1))
  #define PDL_DSM_SUPPORT PDL_ON
#else
  #define PDL_DSM_SUPPORT PDL_OFF
#endif
```

The start.asm code contains a generic interrupt table with a running number for the ISR prototypes.

The last section of *pdl.h* defines the interrupt table with the replaced ISR prototype names.



The intention is that this mechanism makes it easy to switch to a different interrupt table for different devices.

6.4 File: pdl_user.h

The settings in this file are discussed in the 'Getting Started' chapter.

6.5 File: interrupts.c

The *interrupts.c* file contains all first-level service routines (interrupt vectors of start code/*pdl.h*). The code also handles possible shared interrupts, check the occurrence bit(s) (`IRQMON`) first and then call the driver's own service routines. This file ***shall not*** be changed by the user.

The following code excerpt shows, how a shared interrupt is treated.

```
#if (PDL_PERIPHERAL_ENABLE_DT0 == PDL_ON) && (PDL_INTERRUPT_ENABLE_DT0 == PDL_ON)
void DT1_2_IRQHandler(void)
{
    uint32_t u32IrqMon = FM4_INTREQ->IRQ047MON;

    if (0 != (u32IrqMon & 0x00000001u))
    {
        DtIrqHandler(DtChannel0);
    }
    if (0 != (u32IrqMon & 0x00000002u))
    {
        DtIrqHandler(DtChannel1);
    }
}
#endif
```

6.6 File: interrupts.h

This file defines some IRQMON bit masks needed for shared interrupts.

7. PDL Resource Driver API (Low Level)

7.1 Preface

The following chapters explain all driver modules in alphabetical order. The title shows the module name in brackets and upper case followed by the name written out.

n.m (MODULE) Module Name

Below this a small right-aligned table shows the module's definition and configuration type and finally the address operator (if applicable).

Type Definition	`stc_module_t`
Configuration Type	`stc_module_config_t`
Address Operator	`MODULE`*n*

The description begins with the explanation of the configuration(s).

Type	Field	Possible Values	Description

`en_..._t`	`en...`	`Enum1`
		`Enum2`
		`Enum3`
			`Enum1` description
			`Enum2` description
			`Enum3` description

`...`	`...`	`...`	...

Then a short introduction of the module API functions follows. After this each API functions is described in detail and the prototype name, the arguments and the return values are explained.

Prototype	
`type Module_Function( volatile stc_module_t* pstModule, stc_module_config_t* pstcConfig, … )`	
Parameter Name	**Description**
`[in] pstcModule`	Module instance pointer
`[in] pstcConfig`	Pointer to Module instance configuration
…	…
Return Values	**Description**
`Return0`	Description of `Return0`
`Return1`	Description of `Return1`
…	…

Finally the callback functions are introduced.

Prototype
`void CallbackFunction( … )`

If a callback needs arguments, these arguments are explained.

Each API chapter ends with one or some code examples. Note, that none of the example shows the GPIO and pin relocation settings! This depends on the used device and may have the form like this:

```
#include "gpio1pin.h"
function
{
  // Init MFS0 pins for relocation '1' and UART usage
  SetPinFunc_SIN0_1();
  SetPinFunc_SOT0_1();
  // Init CAN1 pins for relocation '2'
  SetPinFunc_RX1_2();
  SetPinFunc_TX1_2();
  . . .
}
```

Note:

Carefully check the resource activation in pdl_user.h before usage, whether your device supports a peripheral and especially a certain instance. Although there are definitions for the device name and its package, the PDL is not able to check for availability.

A symbol in the following tables is concatenated by "…", if a line break occurs.

Example:

Line break concatenation	Symbol without line break
`en_clk_…` `pllowaittime_t`	`en_clk_pllowaittime_t`

7.2 (ADC) Analog Digital Converter Module

Type Definition	stc_adc_t
Configuration Type	stc_adc_config_t
Address Operator	ADC*n*

The ADC module provides all necessary configurations and functions to operate it in its full feature set. The API provides both scan and priority conversion. The module is capable to be used with DMA.

The ADC module can be used with and without interrupts.

7.2.1 Configuration Structure

An ADC instance uses the following configuration structure of the type of `stc_adc_config_t`:

Type	Field	Possible Values	Description
stc_ad… channel_list_t	u32CannelSelect	AD_CH_0 … AD_CH_31	32-Bit ADC channel bitfield. LSB represents enabling channel 0, MSB represents enabling channel 31.
boolean_t	bLsbAlignment	TRUE FALSE	Result LSB aligned Result MSB aligned
en_adc_scan_… mode_t	enScanMode	ScanSingle… Conversion ScanRepeat… Conversion	Single Scan Conversion Mode Repeat Scan Conversion Mode
stc_ad… channel_list_t	u32SamplingTimeSelect	AD_CH_0 … AD_CH_31	Selects channels for Sampling time 0 or 1 setting
en_adc_… sample_time_n_t	enSamplingTimeN0	Value1 Value4 Value8 Value16 Value32 Value64 Value128 Value256	Sampling Time N0 Set value * 1 Set value * 4 Set Value * 8 Set value * 16 Set value * 32 Set Value * 64 Set Value * 128 Set Value * 256
uint8_t	u8SamplingTime0	0…15	Sampling Time 0
en_adc_… sample_time_n_t	enSamplingTimeN1	Value1 Value4 Value8 Value16 Value32 Value64 Value128 Value256	Sampling Time N1 Set value * 1 Set value * 4 Set Value * 8 Set value * 16 Set value * 32 Set Value * 64 Set Value * 128 Set Value * 256
uint8_t	u8SamplingTime1	0…15	Sampling Time 1
uint8_t	u8SamplingMultiplier	*(see peripheral manual)*	Multiplier of N, see peripheral manual for details
uint8_t	u8EnableTime	*(see peripheral manual)*	ADC enabling time, see

Type	Field	Possible Values	Description
			peripheral manual for details
boolean_t	bScanTimerStartEnable	TRUE FALSE	Triggers scan conversion by timer Does not trigger by timer
en_adc_timer_… select_t	enScanConversion… TimerTrigger	AdcNoTimer AdcMft AdcBt0 AdcBt1 … AdcBt13	No selected timer Trigger by MFT Trigger by BT0 Trigger by BT1 … Trigger by BT13
uint8_t	u8ScanFifoDepth	0…15	Depth of scan conversion FIFO
boolean_t	bPrioExtTrigEnable	TRUE FALSE	The external trigger analog inputs are selected with an external input No external trigger
boolean_t	bPrioExtTrigStart… Enable	TRUE FALSE	Triggers Priority Conversion on falling edge of external signal Trigger on rising edge
boolean_t	bPrioTimerStartEnable	TRUE FALSE	Triggers Priority Conversion by timer No trigger by timer
en_adc_timer_… select_t	enPrioConversion… TimerTrigger	AdcNoTimer AdcMft AdcBt0 AdcBt1 … AdcBt13	No selected timer Trigger by MFT Trigger by BT0 Trigger by BT1 … Trigger by BT13
uint8_t	u8PrioLevel1Analog… ChSel	0…7	Priority Level 1 Analog Channel Selector for Channel 0…7
uint8_t	u8PrioLevel2Analog… ChSel	0…31	Priority Level 2 Analog Channel Selector for Channel 0…31
boolean_t	bComparisonEnable	TRUE FALSE	Enable comparison mode Disable comparison mode
uint16_t	u16ComparisonValue	0…4095	ADC Comparison Value (CMPD)
boolean_t	bCompareAllChannels	TRUE FALSE	Compare all selected Channels Compare CCH-Channel
uint8_t	u8ComapreChannel	0…31	CCH-Channel to be compared, if selected Ch. 0...31
boolean_t	bRangeComparison… Enable	TRUE FALSE	Enable Range Comparison Disable Range Comparison
uint16_t	u16UpperLimitRange Value…	0…4095	Upper Limit Value for Range Comparison
uint16_t	u16LowerLimitRange… Value	0…4095	Lower Limit Value for Range Comparison
uint8_t	u8RangeCountValue	1…7	Range count value 1...7
boolean_t	bOutOfRange	TRUE FALSE	Value out of range Value within range
boolean_t	bRangeCompareAll… Channels	TRUE FALSE	Range compare all selected Channels Compare WCCH-Channel

Type	Field	Possible Values	Description
`uint8_t`	`u8RangeComapreChannel`	0…31	WCCH-Channel to be Range compared, if selected Ch. 0...31

If ADC interrupts are enabled in *pdl_user.h* the configuration gets the following additional settings. Note that these settings are not available (i.e. not compiled), if ADC interrupts are disabled.

Type	Field	Possible Values	Description
`boolean_t`	`bCompIrqEqualGreater`	`TRUE` `FALSE`	Generate Interrupt, if `CMPD` most significant 10 bits >= current ADC value Generate Interrupt, if `CMPD` most significant 10 bits < current ADC value
`boolean_t`	`bScanConversion…IrqEnable`	`TRUE` `FALSE`	Enable Scan Conversion Interrupt No interrupt
`func_ptr_adc_…parg32_t`	`pfnPrioConversion`	*(see ADC API)*	Priority Conversion callback pointer
`boolean_t`	`bPrioConversionIrq…Enable`	`TRUE` `FALSE`	Enable Priority Conversion Interrupt No interrupt
`boolean_t`	`bConversionCompIrq…Enable`	`TRUE` `FALSE`	Enable Conversion Comparison Interrupt No interrupt
`boolean_t`	`bFifoOverrunIrqEnable`	`TRUE` `FALSE`	Enable FIFO Overrun interrupt Disable interrupt
`boolean_t`	`bRangeComparisonIrq…Enable`	`TRUE` `FALSE`	Enable Range Comparison interrupt Disable interrupt
`func_ptr_adc_…parg32_t`	`pfnScanConversion`	*(see ADC API)*	Scan Conversion callback pointer
`func_ptr_t`	`pfnErrorCallbackAdc`	*(see ADC API)*	Error during conversion callback pointer
`func_ptr_t`	`pfnPrioErrorCallback…Adc`	*(see ADC API)*	Priority FIFO overrun error callback pointer
`func_ptr_t`	`pfnComparisonCallback`	*(see ADC API)*	Comparison callback pointer
`func_ptr_t`	`pfnRangeCallback`	*(see ADC API)*	Range condition interrupt callback pointer

7.2.2 ADC API

7.2.2.1 Adc_Init()

This function initializes an ADC instance according the given configuration. Note that this API function does not enable the ADC operation.

Prototype	
`en_result_t  Adc_Init( volatile stc_adcn_t* pstcAdc,` `                         stc_adc_config_t*     pstcConfig );`	
Parameter Name	**Description**
[in] pstcAdc	ADC instance pointer
[in] pstcConfig	Pointer to ADC instance configuration
Return Values	**Description**
Ok	Initialization successful done
ErrorInvalidParameter	pstcAdc == NULL pstcConfig == NULL Other invalid configuration setting(s)

7.2.2.2 Adc_DeInit()

This function de-initializes an ADC instance.

Prototype	
`en_result_t  Adc_DeInit( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
[in] pstcAdc	ADC instance pointer
Return Values	**Description**
Ok	Initialization successful done

7.2.2.3 Adc_EnableWaitReady()

This function enables an ADC instance operation and waits for readiness. The timeout polling value is defined as `PDL_ADC_READY_WAIT_COUNT` and can be found in *adc.h* for possible adjustment.

Prototype	
`en_result_t  Adc_EnableWaitReady( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
[in] pstcAdc	ADC instance pointer
Return Values	**Description**
Ok	ADC instance enabled and ready
ErrorTimeout	ADC instance not ready
ErrorInvalidParameter	pstcAdc == NULL

7.2.2.4 Adc_Enable()

This function enables an ADC instance operation but does not wait for readiness.

Prototype	
`en_result_t Adc_Enable( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`Ok`	ADC instance enabled

7.2.2.5 Adc_Ready()

This function checks the readiness of an ADC instance after enabling it. The return values `Ok` and `ErrorNotReady` can be used for a user polling loop.

Prototype	
`en_result_t Adc_Ready( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`Ok`	ADC instance ready
`ErrorNotReady`	ADC instance not ready

7.2.2.6 Adc_ScanSwTrigger()

This function starts a Scan Conversion by Software trigger.

Prototype	
`en_result_t Adc_ScanSwTrigger( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`Ok`	ADC instance triggered (or re-triggered)
`ErrorInvalidParameter`	`pstcAdc == NULL`

7.2.2.7 Adc_PrioSwTrigger()

This function starts a Priority Conversion by Software trigger.

Prototype	
`en_result_t  Adc_PrioSwTrigger( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`Ok`	ADC instance triggered (or re-triggered)
`ErrorInvalidParameter`	`pstcAdc == NULL`

7.2.2.8 Adc_ForceStop()

This function requests a stop of the ADC.

Prototype	
`en_result_t  Adc_ForceSteop( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`Ok`	ADC instance stop request
`ErrorInvalidParameter`	`pstcAdc == NULL`

7.2.2.9 Adc_ScanStatus()

This function reads out the Scan Conversion Status of the ADC.

Prototype	
`en_result_t  Adc_ScanStatus( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`Ok`	ADC instance ready
`OperationInProgress`	ADC instance not ready
`ErrorInvalidParameter`	`pstcAdc == NULL`

7.2.2.10 *Adc_PrioStatus()*

This function reads out the Priority Level 1 Conversion Status of the ADC.

Prototype	
`en_result_t  Adc_PrioStatus( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`Ok`	ADC instance no priority level 1 conversion ongoing
`OperationInProgress`	ADC instance priority level 1 conversion ongoing
`ErrorInvalidParameter`	`pstcAdc == NULL`

7.2.2.11 *Adc_Prio2Status()*

This function reads out the Priority Level 2 Conversion Status of the ADC.

Prototype	
`en_result_t  Adc_Prio2Status( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`Ok`	ADC instance no priority level 2 conversion ongoing
`OperationInProgress`	ADC instance priority level 2 conversion ongoing
`ErrorInvalidParameter`	`pstcAdc == NULL`

7.2.2.12 *Adc_ScanFifoStatus()*

This function reads out the state of the Scan Conversion FIFO.

Prototype	
`en_adc_fifo_status_t Adc_ScanFifoStatus( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`AdcFifoEmpty`	Scan Conversion FIFO empty (no data)
`AdcFifoFilled`	Scan Conversion FIFO (partly) filled
`AdcFifoFull`	Scan Conversion FIFO full
`AdcFifoOverrun`	Scan Conversion FIFO overrun
`AdcFifoError`	Unknown error or `pstcAdc == NULL`

7.2.2.13 Adc_ScanFifoClear()

This function clears the Scan Conversion FIFO.

Prototype	
en_result_t Adc_ScanFifoClear(volatile stc_adcn_t* pstcAdc);	
Parameter Name	**Description**
[in] pstcAdc	ADC instance pointer
Return Values	**Description**
Ok	Scan Conversion FIFO cleared
ErrorInvalidParameter	pstcAdc == NULL

7.2.2.14 Adc_PrioFifoStatus()

This function reads out the state of the Priority Conversion FIFO.

Prototype	
en_adc_fifo_status_t Adc_PrioFifoStatus(volatile stc_adcn_t* pstcAdc);	
Parameter Name	**Description**
[in] pstcAdc	ADC instance pointer
Return Values	**Description**
AdcFifoEmpty	Priority Conversion FIFO empty (no data)
AdcFifoFilled	Sc Priority an Conversion FIFO (partly) filled
AdcFifoFull	Priority Conversion FIFO full
AdcFifoOverrun	Priority Conversion FIFO overrun
AdcFifoError	Unknown error or pstcAdc == NULL

7.2.2.15 Adc_PrioFifoClear()

This function clears the Scan Conversion FIFO.

Prototype	
en_result_t Adc_PrioFifoClear(volatile stc_adcn_t* pstcAdc);	
Parameter Name	**Description**
[in] pstcAdc	ADC instance pointer
Return Values	**Description**
Ok	Priority Conversion FIFO cleared
ErrorInvalidParameter	pstcAdc == NULL

7.2.2.16 Adc_ReadScanFifo()

This function reads out the Scan Conversion FIFO. `Adc_ScanFifoStatus()` should be called before.

Prototype	
`uint32_t Adc_ReadScanFifo( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`uint32_t`	Recent Scan Conversion FIFO value including `INVL`, `RS1`, `RS0`, and Channel data as is. If `pstcAdc == NULL 0xFFFFFFFF` is returned.

7.2.2.17 Adc_GetScanChannel()

This function returns the Channel data from input data from Scan Conversion FIFO. `Adc_ReadScanFifo()` must be called before.

Prototype	
`uint8_t Adc_GetScanChannel( uint32_t u32FifoData );`	
Parameter Name	**Description**
`[in] u32FifoData`	FIFO data (from `Adc_ReadScanFifo()`)
Return Values	**Description**
`uint8_t`	Recent Scan Conversion Channel value.

7.2.2.18 Adc_GetScanDataValid()

This function returns the Channel data from input data from FIFO. `Adc_ReadScanFifo()` must be called before.

Prototype	
`en_adc_fifo_data_valid_t Adc_GetScanDataValid( uint32_t u32FifoData );`	
Parameter Name	**Description**
`[in] u32FifoData`	FIFO data (from `Adc_ReadScanFifo()`)
Return Values	**Description**
`AdcFifoDataValid`	Recent FIFO data valid
`AdcFifoDataInvalid`	Recent FIFO data invalid

7.2.2.19 *Adc_GetScanDataCause()*

This function returns the Scan Conversion Start Cause from FIFO Data. `Adc_ReadScanFifo()` must be called before.

Prototype	
`en_adc_fifo_data_valid_t Adc_GetScanDataCause( uint32_t u32FifoData );`	
Parameter Name	**Description**
`[in] u32FifoData`	FIFO data (from `Adc_ReadScanFifo()`)
Return Values	**Description**
`AdcFifoSoftwareStart`	Recent FIFO data caused by Software Start
`AdcFifoTimerStart`	Recent FIFO data caused by Timer Start
`AdcFifoErrorStart`	Recent FIFO data caused by unknown factor

7.2.2.20 *Adc_ReadPrioFifo()*

This function reads out the Priority Conversion FIFO. `Adc_PrioFifoStatus()` should be called before.

Prototype	
`uint32_t Adc_ReadPrioFifo( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`uint32_t`	Recent Priority Conversion FIFO value including `INVL`, `RS1`, `RS0`, and Channel data as is. If `pstcAdc == NULL 0xFFFFFFFF` is returned.

7.2.2.21 *Adc_GetPrioChannel()*

This function returns the Channel data from input data from Scan Conversion FIFO. `Adc_ReadPrioFifo()` must be called before.

Prototype	
`uint8_t Adc_GetPrioChannel( uint32_t u32FifoData );`	
Parameter Name	**Description**
`[in] u32FifoData`	FIFO data (from `Adc_ReadPrioFifo()`)
Return Values	**Description**
`uint8_t`	Recent Priority Conversion Channel value.

7.2.2.22 *Adc_GetPrioDataValid()*

This function returns the Channel data from input data from FIFO. `Adc_ReadPrioFifo()` must be called before.

Prototype	
`en_adc_fifo_data_valid_t Adc_GetPrioDataValid( uint32_t u32FifoData );`	
Parameter Name	**Description**
`[in] u32FifoData`	FIFO data (from `Adc_ReadPrioFifo()`)
Return Values	**Description**
`AdcFifoDataValid`	Recent FIFO data valid
`AdcFifoDataInvalid`	Recent FIFO data invalid

7.2.2.23 *Adc_GetPrioDataCause()*

This function returns the Priority Conversion Start Cause from FIFO Data. `Adc_ReadPrioFifo()` must be called before.

Prototype	
`en_adc_fifo_data_valid_t Adc_GetPrioDataCause( uint32_t u32FifoData );`	
Parameter Name	**Description**
`[in] u32FifoData`	FIFO data (from `Adc_ReadPrioFifo()`)
Return Values	**Description**
`AdcFifoSoftwareStart`	Recent FIFO data caused by Software Start
`AdcFifoTimerStart`	Recent FIFO data caused by Timer Start
`AdcFifoExternalTrigger`	Recent FIFO data caused by External Trigger
`AdcFifoErrorStart`	Recent FIFO data caused by unknown factor

7.2.2.24 *Adc_GetRangeResult()*

This function reads out the Range Comparison Flag.

Prototype	
`en_adc_range_result_t Adc_GetRangeResultValid( volatile stc_adcn_t* pstcAdc );`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`AdcRangeResultValid`	Range result valid (occurred)
`AdcRangeResultInvalid`	Range result invalid (not occurred)
`AdcRangeResultError`	`pstcAdc == NULL`

7.2.2.25 Scan Conversion Callback

This callback function occurs at the end of a scan conversion. Note that for this function interrupts must be enabled. The pointer to this function is declared in the ADC configuration at `pfnScanConversion`.

Prototype	
void *ScanConversionCallbackName*(uint32_t*);	
Parameter Name	**Description**
[in] uint32_t*	Pointer to the Scan Conversion FIFO

7.2.2.26 Priority Conversion Callback

This callback function occurs at the end of a priority conversion. Note that for this function interrupts must be enabled. The pointer to this function is declared in the ADC configuration at `pfnPrioConversion`.

Prototype	
void *PrioConversionCallbackName*(uint32_t*);	
Parameter Name	**Description**
[in] uint32_t*	Pointer to the Priority Conversion FIFO

7.2.2.27 Scan Conversion Error Callback

This callback function is called if an error occurred during Scan Conversion. Note that for this function interrupts must be enabled. The pointer to this function is declared in the ADC configuration at `pfnErrorCallbackAdc`.

Prototype
void *ScanErrorCallbackName*(void);

7.2.2.28 Priority Conversion Error Callback

This callback function is called if an error occurred during Priority Conversion. Note that for this function interrupts must be enabled. The pointer to this function is declared in the ADC configuration at `pfnPrioErrorCallbackAdc`.

Prototype
void *PrioErrorCallbackName*(void);

7.2.2.29 Comparison Conversion Callback

This callback function is called at the end of a Comparison Conversion. Note that for this function interrupts must be enabled. The pointer to this function is declared in the ADC configuration at `pfnComparisonCallback`.

Prototype
void *ComparisonCallbackName*(void);

7.2.2.30 *Range Conversion Callback*

This callback function is called at the end of a Range Conversion. Note that for this function interrupts must be enabled. The pointer to this function is declared in the ADC configuration at `pfnRangeCallback`.

Prototype
`void` *RangeCallbackName* `( void );`

7.2.3 ADC Examples

The PDL example folder contains five ADC usage examples:

- adc_bt — ADC usage triggered by a Base Timer
- adc_dma — ADC with DMA transfer
- adc_irq — ADC with interrupts
- adc_mft — ADC triggered by a Multi Function Timer
- adc_polling — ADC in polling mode (without interrupts)

7.3 (BT) Base Timer

Type Definition	Function Type	-
Configuration Types	Init config	stc_bt_pwm_config_t stc_bt_ppg_config_t stc_bt_rt_config_t stc_bt_pwc_config_t
	Interrupt	stc_pwm_int_sel_t stc_pwm_int_cb_t stc_ppg_int_sel_t stc_ppg_int_cb_t stc_rt_int_sel_t stc_rt_int_cb_t stc_pwc_int_sel_t stc_pwc_int_cb_t
Address Operator		BT*n*

How to use PWM timer function of BT module?

Bt_ConfigIOMode() must be used to configure BT I/O mode to I/O mode 0 first.

Bt_SelTimerMode() must be used to configure BT mode to PWM function then.

Bt_Pwm_Init() is used to initialize PWM timer source clock, output polarity, operation mode and so on. Following operation mode can set:

- Continuous mode
- One-shot mode

A PWM interrupt can be enabled by the function Bt_Pwm_EnableInt(). This function can set callback function for each channel too.

With Bt_Pwm_WriteCycleVal() the PWM cycle counter is set to the value given in the parameter Bt_Pwm_WriteCycleVal#u16Cycle.

With Bt_Pwm_WriteDutyVal() the PWM duty counter is set to the value given in the parameter Bt_Pwm_WriteDutyVal#u16Duty.

Notes that PWM can be only set to 16-bit mode, so above two parameters should be 16-bit.

Bt_Pwm_EnableCount() is used to enabe PWM counter.

After above setting is done, calling Bt_Pwm_EnableSwTrig() will start PWM timer.

With Bt_Pwm_ReadCurCnt() the current PWM count can be read when PWM timer is running.

With interrupt mode, when the interrupt occurs, the interrupt flag will be cleared and run into user interrupt callback function.

With polling mode, user can use Bt_Pwm_GetIntFlag() to check if the interrupt occurs, and clear the interrupt flag by Bt_Pwm_ClrIntFlag().

When stopping the PWM timer, use Bt_Pwm_DisableCount() to disable PWM counter and Bt_Pwm_DisableInt() to disable PWM interrupt.

How to use PPG timer function of BT module?

Bt_ConfigIOMode() must be used to configure BT I/O mode to I/O mode 0 first.

Bt_SelTimerMode() must be used to configure BT mode to PPG function then.

Bt_Ppg_Init() is used to initialize PPG timer source clock, output polarity, operation mode and so on. Following operation mode can set:

- Continuous mode
- One-shot mode

A PPG interrupt can be enabled by the function Bt_Ppg_EnableInt(). This function can set callback function for each channel too.

With Bt_Ppg_WriteLowWidthVal() the PPG low width is set to the value given in the parameter Bt_Ppg_WriteLowWidthVal#u16Cycle.

With Bt_Ppg_WriteHighWidthVal() the PPG high width is set to the value given in the parameter Bt_Ppg_WriteHighWidthVal#u16Cycle.

Notes that PPG can be only set to 16-bit mode, so above two parameters should be 16-bit.

Bt_Ppg_EnableCount() is used to enabe PPG counter.

After above setting is done, calling Bt_Ppg_EnableSwTrig() will start PPG timer.

With Bt_Ppg_ReadCurCnt() the current PPG count can be read when PPG timer is running.

With interrupt mode, when the interrupt occurs, the interrupt flag will be cleared and run into user interrupt

callback function.

With polling mode, user can use Bt_Ppg_GetIntFlag() to check if the interrupt occurs, and clear the interrupt flag by Bt_Ppg_ClrIntFlag().

When stopping the PPG timer, use Bt_Ppg_DisableCount() to disable PPG counter and Bt_Ppg_DisableInt() to disable PPG interrupt.

How to use Reload Timer (RT) function of BT module?

Bt_ConfigIOMode() must be used to configure BT I/O mode to I/O mode 0 first.

Bt_SelTimerMode() must be used to configure BT mode to Reload Timer function then.

Bt_Rt_Init() is used to initialize RT source clock, output polarity, operation mode and so on. Following operation mode can set:

- Reload mode
- One-shot mode

A RT interrupt can be enabled by the function Bt_Rt_EnableInt(). This function can set callback function for each channel too.

RT can be set to 16-bit mode or 32-bit mode. In 16-bit mode, Bt_Rt_WriteCycleVal() set the RT counter cycle of according channel. In the 32-bit mode, Bt_Rt_WriteCycleVal() with even channel register pointer as paremater set the low 16-bit cycle value and Bt_Rt_WriteCycleVal() with odd channel register pointer as paremater set the high 16-bit cycle value.

Bt_Rt_EnableCount() is used to enabe RT counter.

After above setting is done, calling Bt_Rt_EnableSwTrig() will start Reload Timer.

With Bt_Rt_ReadCurCnt() the current RT count can be read when Reload Timer is running.

With interrupt mode, when the interrupt occurs, the interrupt flag will be cleared and run into user interrupt callback function.

With polling mode, user can use Bt_Rt_GetIntFlag() to check if the interrupt occurs, and clear the interrupt flag by Rt_Ppg_ClrIntFlag().

When stopping the PPG timer, use Bt_Rt_DisableCount() to disable RT counter and Bt_Rt_DisableInt() to disable RT interrupt.

How to use PWC timer function of BT module?

Bt_ConfigIOMode() must be used to configure BT I/O mode to I/O mode 0 first.

Bt_SelTimerMode() must be used to configure BT mode to PWC function then.

Bt_Pwc_Init() is used to initialize PWC timer source clock, measurement mode and so on.

Following measurement mode can set:

- High pulse width measurement
- Cycle measurement with rising edges
- Cycle measurement with falling edges
- Interval measurement between all edges
- Low pulse width measurement

Following operation mode can be set:

- Continuous mode
- One-shot mode

A PWC interrupt can be enabled by the function Bt_Pwc_EnableInt(). This function can set callback function for each channel too.

Bt_Pwc_EnableCount() is used to enabe PWC counter.

After above setting is done, when the valid edge (1st) is detected, the measurement starts, and the valid edge (2nd) is detected, the measurement ends, the interrupt request flag is set at the same time.

PWC timer can be set to 16-bit mode and 32-bit mode. In the 16-bit mode, with Bt_Pwc_Get16BitMeasureData() the measured value can be read after measurement completes, in the 32-bit mode, with Bt_Pwc_Get32BitMeasureData() the measured value can be read.

With interrupt mode, when the interrupt occurs, the interrupt flag will be cleared and the code runs into user interrupt callback function.

With polling mode, user can use Bt_Pwc_GetIntFlag() to check if the completion interrupt occurs, and clear the interrupt flag by Bt_Pwc_ClrIntFlag(). Bt_Pwc_GetErrorFlag() is used to get the error flag of PWC measurement.

When stopping the PWC timer, use Bt_Pwc_DisableCount() to disable PWC counter and Bt_Pwc_DisableInt() to disable PWC interrupt.

7.3.1 Configuration Structure

A PWM timer instance uses the following configuration structure of the type of `stc_bt_pwm_config_t`:

Type	Field	Possible Values	Description
en_pwm_clock_pres_t	enPres	 PwmPresNone PwmPres1Div4 PwmPres1Div16 PwmPres1Div128 PwmPres1Div256 PwmPres1ExtClkRising PwmPres1ExtClkFalling PwmPres1ExtClkBoth PwmPres1Div512 PwmPres1Div1024 PwmPres1Div2048	Select internal count clock or trigger event. The counter is a division of the machine clock. Original freq 1/4 freq 1/16 freq 1/128 freq 1/256 freq Ext clk (rising edge event) Ext clk (falling edge event) Ext clk (both edge event) 1/512 freq 1/1024 freq 1/2048 freq
en_pwm_restart_enable_t	enRestartEn	 PwmRestartDisable PwmRestartEnable	PWM restart option: Disable PWM restart Enable PWM restart
en_pwm_output_mask_t	enOutputMask	 PwmOutputNormal PwmOutputMask	PWM output mask setting. Normal Output PWM output fixed to LOW output
en_pwm_ext_trig_t	enExtTrig	 PwmExtTrigDisable PwmExtTrigRising PwmExtTrigFalling PwmExtTrigBoth	PWM external trigger setting: Disable external trigger Enable external trigger with rising edge Enable external trigger with falling edge Enable external trigger with both edge
en_pwm_output_polarity_t	enOutputPolarity	 PwmPolarityLow PwmPolarityHigh	PWM output polarity setting Low High
en_pwm_mode_t	enMode	PwmContinuous PwmOneshot	Continuous mode One-shot mode

A selection of the PWM interrupt instance uses the following configuration structure of the type of stc_pwm_int_sel_t:

Type	Field	Possible Values	Description
boolean_t	bPwmTrigInt	TRUE FALSE	Trigger interrupt selection: Interrupt to be operated. Interrupt not to be operated.
boolean_t	bPwmDutyMatchInt	TRUE FALSE	Duty match interrupt selection. Interrupt to be operated. Interrupt not to be operated.
boolean_t	bPwmUnderflowInt	TRUE FALSE	Underflow interrupt selection. Interrupt to be operated. Interrupt not to be operated.

A PWM interrupt callback function instance uses the following configuration structure of the type of stc_pwm_int_cb_t:

Type	Field	Possible Values	Description
func_ptr_t	pfnPwmTrigIntCallback	-	Pointer to trigger interrupt callback function.
func_ptr_t	pfnPwmDutyMatchIntCallback	-	Pointer to duty match interrupt callback function.
func_ptr_t	pfnPwmUnderflowIntCallback	-	Pointer to underflow interrupt callback function.

A PPG timer instance uses the following configuration structure of the type of stc_bt_ppg_config_t:

Type	Field	Possible Values	Description
en_ppg_clock_pres_t	enPres	PpgPresNone PpgPres1Div4 PpgPres1Div16 PpgPres1Div128 PpgPres1Div256 PpgPres1ExtClkRising PpgPres1ExtClkFalling PpgPres1ExtClkBoth PpgPres1Div512 PpgPres1Div1024 PpgPres1Div2048	Select internal count clock. The counter is a division of the machine clock. Original 1/4 freq 1/16 freq 1/128 freq 1/256 freq Ext clk (rising edge event) Ext clk (falling edge event) Ext clk (both edge event) 1/512 freq 1/1024 freq 1/2048 freq
en_ppg_restart_enable_t	enRestartEn	PpgRestartDisable PpgRestartEnable	software trigger or trigger input Restart enable Restart disable
en_ppg_output_mask_t	enOutputMask	PpgOutputNormal	PPG output mask setting. Normal Output

Type	Field	Possible Values	Description
		PpgOutputMask	PWM output fixed to LOW output
en_ppg_ext_trig_t	enExtTrig	PpgExtTrigDisable PpgExtTrigRising PpgExtTrigFalling PpgExtTrigBoth	PWM external trigger setting Disable external trigger Enable external trigger with rising edge Enable external trigger with falling edge Enable external trigger with both edge
en_ppg_output_polarity_t	enOutputPolarity	PpgPolarityLow PpgPolarityHigh	PPG output polarity setting: Low High
en_ppg_mode_t	enMode	PpgContinuous PpgOneshot	Continuous mode One-shot mode

A selection of PPG interrupt instance uses the following configuration structure of the type of stc_ppg_int_sel_t:

Type	Field	Possible Values	Description
boolean_t	bPpgTrigInt	TRUE FALSE	PPG trigger interrupt selection Interrupt to be operated. Interrupt not to be operated.
boolean_t	bPpgUnderflowInt	TRUE FALSE	PPG underflow interrupt selection. Interrupt to be operated. Interrupt not to be operated.

A PPG callback function instance uses the following configuration structure of the type of stc_ppg_int_cb_t:

Type	Field	Possible Values	Description
func_ptr_t	pfnPpgTrigIntCallback	-	Pointer to PPG trigger interrupt callback function.
func_ptr_t	pfnPpgUnderflowIntCallback	-	Pointer to PPG underflow interrupt callback function.

A reload timer instance uses the following configuration structure of the type of stc_bt_rt_config_t:

Type	Field	Possible Values	Description
en_rt_clock_pres_t	enPres	RtPresNone RtPres1Div4 RtPres1Div16 RtPres1Div128 RtPres1Div256 RtPres1ExtClkRising RtPres1ExtClkFalling RtPres1ExtClkBoth RtPres1Div512 RtPres1Div1024 RtPres1Div2048	Select internal count clock. The counter is a division of the machine clock. Original freq 1/4 freq 1/16 freq 1/128 freq 1/256 freq Ext clk (rising edge event) Ext clk (falling edge event) Ext clk (both edge event) 1/512 freq 1/1024 freq 1/2048 freq
en_rt_timer_size_t	enSize	RtSize16Bit RtSize32Bit	Select 32-bit timer function 16-bit mode 32-bit mode
en_rt_ext_trigger_t	enExtTrig	RtExtTiggerDisable RtExtTiggerRisingEdge RtExtTiggerFallingEdge RtExtTiggerBothEdge RtExtTiggerLowLevel RtExtTiggerHighLevel	Select trigger input edge and gate function level Disable external trigger Select external trigger with rising edge Select external trigger with falling edge Select external trigger with both edge Select external trigger with low level Select external trigger with high level
en_rt_output_polarity_t	enOutputPolarity	RtPolarityLow RtPolarityHigh	Reload timer output polarity setting Low High
en_rt_mode_t	enMode	RtReload RtOneshot	Reload mode One-shot mode

A selection of Reload timer interrupt instance uses the following configuration structure of the type of stc_rt_int_sel_t:

Type	Field	Possible Values	Description
boolean_t	bRtTrigInt	TRUE FALSE	Trigger interrupt selection Interrupt to be operated. Interrupt not to be operated.
boolean_t	bRtUnderflowInt	TRUE FALSE	Underflow interrupt selection. Interrupt to be operated. Interrupt not to be operated.

A Reload timer interrupt callback function instance uses the following configuration structure of the type of stc_rt_int_cb_t

Type	Field	Possible Values	Description
func_ptr_t	pfnRtTrigIntCallback	-	Pointer to trigger interrupt callback function.
func_ptr_t	pfnRtUnderflowIntCallback	-	Pointer to underflow interrupt callback function.

A PWC timer instance uses the following configuration structure of the type of stc_bt_pwc_config_t:

Type	Field	Possible Values	Description
en_pwc_clock_pres_t	enPres	PwcPresNone PwcPres1Div4 PwcPres1Div16 PwcPres1Div128 PwcPres1Div256 RtPres1Div512 RtPres1Div1024 RtPres1Div2048	Select internal count clock. The counter is a division of the machine clock. Original 1/4 freq 1/16 freq 1/128 freq 1/256 freq 1/512 freq 1/1024 freq 1/2048 freq
en_pwc_timer_size_t	enSize	PwcSize16Bit PwcSize32Bit	Select 16/32-bit timer function 16-bit mode 32-bit mode
en_pwc_measure_edge_t	enMeasureEdge	PwcMeasureRisingToFalling PwcMeasureRisingToRising PwcMeasureFallingToFalling PwcMeasureEitherToEither PwcMeasureFallingToRising	Set the measure mode of PWC timer Measure between rising edge with falling edge Measure between rising edge with rising edge Measure between falling edge with falling edge Measure between either edge with either edge Measure between falling edge with falling edge
en_pwc_mode_t	enMode	PwcContinuous PwcOneshot	Reload mode One-shot mode

A selection of PWC timer interrupt instance uses the following configuration structure of the type of stc_pwc_int_sel_t:

Type	Field	Possible Values	Description
boolean_t	bPwcMeasCmpInt	TRUE FALSE	Trigger interrupt selection. Interrupt to be operated. Interrupt not to be operated.
boolean_t	bPwcMeasOverflowInt	TRUE FALSE	Underflow interrupt selection. Interrupt to be operated. Interrupt not to be operated.

A Reload timer interrupt callback function instance uses the following configuration structure of the type of stc_rt_int_cb_t:

Type	Field	Possible Values	Description
func_ptr_t	pfnRtTrigIntCallback	-	Poiter to PWC measure completion callback function.
func_ptr_t	pfnRtUnderflowIntCallback	-	Poiter to PWC measure overflow callback function.

7.3.2 BT API

7.3.2.1 Bt_ConfigIOMode ()

This function configures BT IO mode.

Prototype	
en_result_t Bt_ConfigIOMode(volatile stc_btn_t* pstcBt, en_bt_io_mode_t enIoMode);	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
[in] enIoMode	BT IO mode.
Return Values	**Description**
Ok	BT IO mode has been set successfully.
ErrorInvalidParameter	pstcBt == NULL Other invalid configuration.

7.3.2.2 Bt_SelTimerMode ()

This function selects timer function of BT.

Prototype	
`en_result_t Bt_SelTimerMode(volatile stc_btn_t* pstcBt, en_bt_timer_mode_t enTimerMode);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] enTimerMode`	Timer mode
Return Values	**Description**
`Ok`	BT timer mode has been selected successfully

`ErrorInvalidParameter`	`pstcBt == NULL`
	`enTimerMode > BtPwcMode`
	Other invalid configuration

7.3.2.3 Bt_Pwm_Init ()

This function initializes PWM function of BT.

Prototype	
`en_result_t Bt_Pwm_Init(volatile stc_btn_t* pstcBt, stc_bt_pwm_config_t* pstcPwmConfig);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] pstcPwmConfig`	Pointer to PWM configuration.
Return Values	**Description**
`Ok`	PWM function has been configured successfully.
`ErrorInvalidParameter`	`pstcBt == NULL` `pstcPwmConfig` invalid parameter. Other invalid configuration.

7.3.2.4 Bt_Pwm_EnableCount ()

This function enables PWM timer counting.

Prototype	
`en_result_t Bt_Pwm_EnableCount(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
Return Values	**Description**
`Ok`	Enable PWM timer counting successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.5 *Bt_Pwm_DisableCount ()*

This function disables PWM timer counting.

Prototype	
`en_result_t Bt_Pwm_DisableCount(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
Return Values	**Description**
`Ok`	Disable PWM timer counting successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.6 *Bt_Pwm_EnableSwTrig ()*

This function starts PWM timer by software.

Prototype	
`en_result_t Bt_Pwm_EnableSwTrig(volatile stc_btn_t* pstcBt)`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
Return Values	**Description**
`Ok`	Start PWM timer successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.7 *Bt_Pwm_EnableInt ()*

This function enables PWM timer interrupt and set corresponding callback functions.

Prototype	
`en_result_t Bt_Pwm_EnableInt(volatile stc_btn_t* pstcBt, stc_pwm_int_sel_t* pstcIntSel, stc_pwm_int_cb_t*  pstcIntCallback);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] pstcIntSel`	Pointer to PWM timer interrupt types.
`[in] pstcIntCallback`	Pointer to interrupt callback functions.
Return Values	**Description**
`Ok`	Enable PWM timer interrupt successfully.

`ErrorInvalidParameter`	`pstcBt == NULL`
	`pstcIntSel` parameter invalid.
	`pfnIntCallback == NULL`
	Other invalid configuration.

7.3.2.8 Bt_Pwm_DisableInt ()

This function disables PWM timer interrupt and clear corresponding callback functions.

Prototype	
`en_result_t Bt_Pwm_DisableInt(volatile stc_btn_t* pstcBt, stc_pwm_int_sel_t* pstcIntSel);`	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
[in] pstcIntSel	Pointer to PWM timer interrupt type.
Return Values	**Description**
Ok	Disable PWM timer interrupt successfully
ErrorInvalidParameter	pstcBt == NULL pstcIntSel parameter invalid. Other invalid configuration.

7.3.2.9 Bt_Pwm_GetIntFlag ()

This function gets interrupt flag of PWM timer by type.

Prototype	
`en_int_flag_t Bt_Pwm_GetIntFlag(volatile stc_btn_t* pstcBt, en_bt_pwm_int_t enIntType);`	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
[in] enIntType	PWM timer interrupt type.
Return Values	**Description**
PdlSet	Interrupt flag is set.
PdlClr	Interrupt flag is clear.

7.3.2.10 Bt_Pwm_ClrIntFlag ()

This function clears interrupt flag of PWM timer.

Prototype	
`en_result_t Bt_Pwm_ClrIntFlag(volatile stc_btn_t* pstcBt, en_bt_pwm_int_t enIntType)`	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
[in] enIntType	PWM timer interrupt type.
Return Values	**Description**
Ok	Clear interrupt flag successfully
ErrorInvalidParameter	pstcBt == NULL enIntType > PwmUnderflowInt

7.3.2.11 Bt_Pwm_WriteCycleVal ()

This function writes cycle value of PWM timer.

Prototype	
`en_result_t Bt_Pwm_WriteCycleVal(volatile stc_btn_t* pstcBt, uint16_t u16Cycle);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] u16Cycle`	Cycle value.
Return Values	**Description**
`Ok`	Write Cycle value successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.12 Bt_Pwm_WriteDutyVal ()

This function writes duty value of PWM timer.

Prototype	
`en_result_t Bt_Pwm_WriteDutyVal(volatile stc_btn_t* pstcBt, uint16_t u16Duty);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] u16Duty`	Duty value.
Return Values	**Description**
`Ok`	Write duty value successfully
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.13 Bt_Pwm_ReadCurCnt ()

This function reads current count value of PWM timer.

Prototype	
`uint16_t Bt_Pwm_ReadCurCnt(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
Return Values	**Description**
`value`	Current count value.

7.3.2.14 Bt_Ppg_Init ()

This function initializes PPG function of BT.

Prototype	
`en_result_t Bt_Ppg_Init(volatile stc_btn_t* pstcBt, stc_bt_ppg_config_t* pstcPpgConfig);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] pstcPpgConfig`	Pointer to PPG configuration.
Return Values	**Description**
`Ok`	PPG function has been configured successfully.
`ErrorInvalidParameter`	`pstcBt == NULL` `pstcPpgConfig` parameter is invalid. Other invalid configuration

7.3.2.15 Bt_Ppg_EnableCount ()

This function enables PPG timer counting.

Prototype	
`en_result_t Bt_Ppg_EnableCount(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
Return Values	**Description**
`Ok`	Enable PPG timer counting successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.16 Bt_Ppg_DisableCount ()

This function disables PPG timer counting.

Prototype	
`en_result_t Bt_Ppg_DisableCount(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
Return Values	**Description**
`Ok`	Disable PPG timer counting successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.17 Bt_Ppg_EnableSwTrig ()

This function starts PPG timer by software.

Prototype	
`en_result_t Bt_Ppg_EnableSwTrig(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
Return Values	**Description**
`Ok`	Start PPG timer successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.18 Bt_Ppg_EnableInt ()

This function enables PPG timer interrupt and sets corresponding callback function.

Prototype	
`en_result_t Bt_Ppg_EnableInt(volatile stc_btn_t* pstcBt, stc_ppg_int_sel_t* pstcIntSel, stc_ppg_int_cb_t* pstcIntCallback);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] pstcIntSel`	Pointer to PPG timer interrupt types.
`[in] pstcIntCallback`	Pointer to interrupt callback functions.
Return Values	**Description**
`Ok`	Enable PPG timer interrupt successfully.

`ErrorInvalidParameter`	`pstcBt == NULL`
	`pfnIntCallback == NULL`
	Other invalid configuration.

7.3.2.19 Bt_Ppg_DisableInt ()

This function disables PPG timer interrupt and clears corresponding callback function.

Prototype	
`en_result_t Bt_Ppg_DisableInt(volatile stc_btn_t* pstcBt, stc_ppg_int_sel_t* pstcIntSel)`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] pstcIntSel`	Pointer to PPG timer interrupt types.
Return Values	**Description**
`Ok`	Disable PPG timer interrupt successfully.
`ErrorInvalidParameter`	`pstcBt == NULL` Other invalid configuration.

7.3.2.20 Bt_Ppg_GetIntFlag ()

This function gets interrupt flag of PPG timer by type.

Prototype	
`en_int_flag_t Bt_Ppg_GetIntFlag(volatile stc_btn_t* pstcBt, en_bt_ppg_int_t enIntType);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] enIntType`	PPG timer interrupt type.
Return Values	**Description**
`PdlSet`	Interrupt flag is set.
`PdlClr`	Interrupt flag is clear.

7.3.2.21 Bt_Ppg_ClrIntFlag ()

This function clears interrupt flag of PPG timer by type.

Prototype	
`en_result_t Bt_Ppg_ClrIntFlag(volatile stc_btn_t* pstcBt, en_bt_ppg_int_t enIntType);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] enIntType`	PPG timer interrupt type.
Return Values	**Description**
`Ok`	Clear interrupt flag successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.22 Bt_Ppg_WriteLowWidthVal ()

This function writes low width count value of PPG timer.

Prototype	
`en_result_t Bt_Ppg_WriteLowWidthVal(volatile stc_btn_t* pstcBt, uint16_t u16Val);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] u16Val`	Low width count value.
Return Values	**Description**
`Ok`	Write low width count value successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.23 Bt_Ppg_WriteHighWidthVal ()

This function writes high width count value of PPG timer.

Prototype	
`en_result_t Bt_Ppg_WriteHighWidthVal(volatile stc_btn_t* pstcBt, uint16_t u16Val);`	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
[in] u16Val	High width count value.
Return Values	**Description**
Ok	Write high width count value successfully.
ErrorInvalidParameter	pstcBt == NULL

7.3.2.24 Bt_Ppg_ReadCurCnt ()

This function reads current count value of PPG timer.

Prototype	
`uint16_t Bt_Pwm_ReadCurCnt(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
Return Values	**Description**
value	Current count value.

7.3.2.25 Bt_Rt_Init ()

This function initializes RT(Reload timer) function of BT.

Prototype	
`en_result_t Bt_Rt_Init(volatile stc_btn_t* pstcBt, stc_bt_rt_config_t* pstcRtConfig);`	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
[in] pstcRtConfig	Pointer to RT configuration.
Return Values	**Description**
Ok	RT timer has been configured successfully.
ErrorInvalidParameter	pstcBt == NULL pstcRtConfig parameter invalid. Other invalid configuration

7.3.2.26 Bt_Rt_EnableCount ()

This function enables RT counting.

Prototype	
`en_result_t Bt_Rt_EnableCount(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
Return Values	**Description**
`Ok`	Enable RT counting successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.27 Bt_Rt_DisableCount ()

This function disables RT counting.

Prototype	
`en_result_t Bt_Rt_DisableCount(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
Return Values	**Description**
`Ok`	Disable RT counting successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.28 Bt_Rt_EnableSwTrig ()

This function starts RT by software.

Prototype	
`en_result_t Bt_Rt_EnableSwTrig(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
Return Values	**Description**
`Ok`	Start RT successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.29 Bt_Rt_EnableInt ()

This function enables RT interrupt.

Prototype	
en_result_t Bt_Rt_EnableInt(volatile stc_btn_t* pstcBt, stc_rt_int_sel_t* pstcIntSel, stc_rt_int_cb_t* pstcIntCallback);	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
[in] pstcIntSel	Pointer to Reload timer interrupt types.
[in] pstcIntCallback	Pointer to interrupt callback functions.
Return Values	**Description**
Ok	Enable RT interrupt successfully.
ErrorInvalidParameter	pstcBt == NULL pstcIntSel parameter invalid pfnIntCallback == NULL Other invalid configuration.

7.3.2.30 Bt_Rt_DisableInt ()

This function disables RT interrupt.

Prototype	
en_result_t Bt_Rt_DisableInt(volatile stc_btn_t* pstcBt, stc_rt_int_sel_t* pstcIntSel);	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
[in] pstcIntSel	Pointer to RT interrupt type.
Return Values	**Description**
Ok	Disable RT interrupt successfully.
ErrorInvalidParameter	pstcBt == NULL pstcIntSel parameter invalid. Other invalid configuration

7.3.2.31 Bt_Rt_GetIntFlag ()

This function gets interrupt flag of RT.

Prototype	
en_int_flag_t Bt_Ppg_GetIntFlag(volatile stc_btn_t* pstcBt, en_bt_ppg_int_t enIntType);	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
[in] enIntType	RT interrupt type.
Return Values	**Description**
PdlSet	Interrupt flag is set.
PdlClr	Interrupt flag is clear.

7.3.2.32 Bt_Rt_ClrIntFlag ()

This function clears interrupt flag of RT.

Prototype	
`en_result_t Bt_Rt_ClrIntFlag(volatile stc_btn_t* pstcBt, en_bt_rt_int_t enIntType);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] enIntType`	Reload timer interrupt type.
Return Values	**Description**
`Ok`	Clear interrupt flag successfully.
`ErrorInvalidParameter`	`pstcBt == NULL` `enIntType > RtUnderflowInt`

7.3.2.33 Bt_Rt_WriteCycleVal ()

This function writes count cycle of RT.

Prototype	
`en_result_t Bt_Rt_WriteCycleVal(volatile stc_btn_t* pstcBt, uint16_t u16Val);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] u16Val`	Cycle value.
Return Values	**Description**
`Ok`	Write count cycle successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.34 Bt_Rt_ReadCurCnt ()

This function reads current count value of RT.

Prototype	
`uint16_t Bt_Rt_ReadCurCnt(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
Return Values	**Description**
`value`	Current count value.

7.3.2.35 Bt_Pwc_Init ()

This function initializes PWC function of BT.

Prototype	
`en_result_t Bt_Pwc_Init(volatile stc_btn_t* pstcBt, stc_bt_pwc_config_t* pstcPwcConfig);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] pstcPwcConfig`	Pointer to PWC configuration.
Return Values	**Description**
`Ok`	PWC function has been configured successfully.
`ErrorInvalidParameter`	`pstcBt == NULL` `pstcPwcConfig` parameter invalid Other invalid configuration.

7.3.2.36 Bt_Pwc_EnableCount ()

This function enables PWC timer counting.

Prototype	
`en_result_t Bt_Pwc_EnableCount(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a PWC timer instance.
Return Values	**Description**
`Ok`	Enable PWC timer counting successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.37 Bt_Pwc_DisableCount ()

This function disables PWC timer counting.

Prototype	
`en_result_t Bt_Pwc_DisableCount(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a PWC timer instance.
Return Values	**Description**
`Ok`	Disable PWC timer counting successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.3.2.38 *Bt_Pwc_EnableInt ()*

This function enables PWC timer interrupt.

Prototype	
en_result_t Bt_Pwc_EnableInt(volatile stc_btn_t* pstcBt, stc_pwc_int_sel_t* pstcIntSel, stc_pwc_int_cb_t* pstcIntCallback);	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
[in] pstcIntSel	Pointer to PWC timer interrupt type.
[in] pstcIntCallback	Pointer to interrupt callback function.
Return Values	**Description**
Ok	Enable PWC timer interrupt successfully.
ErrorInvalidParameter	pstcBt == NULL pstcIntSel parameter invalid pfnIntCallback == NULL Other invalid configuration

7.3.2.39 *Bt_Pwc_DisableInt ()*

This function disables PWC timer interrupt.

Prototype	
en_result_t Bt_Pwc_DisableInt(volatile stc_btn_t* pstcBt, stc_pwc_int_sel_t* pstcIntSel);	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
[in] pstcIntSel	Pointer to PWC timer interrupt type.
Return Values	**Description**
Ok	Disable PWC timer interrupt successfully.
ErrorInvalidParameter	pstcBt == NULL pstcIntSel parameter invalid. Other invalid configuration.

7.3.2.40 *Bt_Pwc_GetIntFlag ()*

This function gets interrupt flag of PWC timer by type.

Prototype	
en_int_flag_t Bt_Pwc_GetIntFlag(volatile stc_btn_t* pstcBt, en_bt_pwc_int_t enIntType);	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
[in] enIntType	PWC timer interrupt type.
Return Values	**Description**
PdlSet	Interrupt flag is set.
PdlClr	Interrupt flag is clear.

7.3.2.41 *Bt_Pwc_ClrIntFlag ()*

This function clears interrupt flag of PWC timer.

Prototype	
en_result_t Bt_Pwc_ClrIntFlag(volatile stc_btn_t* pstcBt, en_bt_pwc_int_t enIntType);	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
[in] enIntType	PWC timer interrupt type.
Return Values	**Description**
Ok	Clear interrupt flag successfully.
ErrorInvalidParameter	pstcBt == NULL enIntType > PwcMeasureOverflowInt

7.3.2.42 *Bt_Pwc_GetErrorFlag ()*

This function gets error flag of PWC timer.

Prototype	
en_stat_flag_t Bt_Pwc_GetErrorFlag(volatile stc_btn_t* pstcBt);	
Parameter Name	**Description**
[in] pstcBt	Pointer to a BT instance.
Return Values	**Description**
PdlSet	Error flag is set.
PdlClr	Error flag is clear.

7.3.2.43 Bt_Pwc_Get16BitMeasureData ()

This function gets 16 bits measure data of PWC timer.

Prototype	
`uint16_t Bt_Pwc_Get16BitMeasureData(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
Return Values	**Description**
`value`	Measurement data value.

7.3.2.44 Bt_Pwc_Get32BitMeasureData ()

This function gets 32 bits measure data of PWC timer.

Prototype	
`uint32_t Bt_Pwc_Get32BitMeasureData(volatile stc_btn_t* pstcBt);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
Return Values	**Description**
`value`	Measurement 32bit data value.

7.3.2.45 Bt_IrqHandler ()

This function implements BT interrupt service routine.

Prototype	
`void Bt_IrqHandler( volatile stc_btn_t* pstcBt,stc_bt_intern_data_t* pstcBtInternData);`	
Parameter Name	**Description**
`[in] pstcBt`	Pointer to a BT instance.
`[in] pstcBtInternData`	Pointer to BT internal data.
Return Values	**Description**
-	

7.3.2.46 Bt_SetSimultaneousStart ()

This function sets the Simultaneous Start register of Base timer.

Prototype	
`void Bt_SetSimultaneousStart(uint16_t u16Value);`	
Parameter Name	**Description**
`[in] u16Value`	Channel index.
Return Values	**Description**
-	

7.4 (CAN) Controller Area Network

Type Definition	`stc_can_t`
Configuration Type	`stc_can_config_t`
Address Operator	CAN*n*

First, to initialize a CAN module, `Can_Init()` must be called. The callback functions are optional, but recommended, otherwise there is no report to the API in case of any error.

`Can_DeInit()` has to be used if any of the settings from `Can_Init()` have to be changed (use `Can_DeInit()` and afterwards `Can_Init()`).

`Can_DeInit()` is used to completely disable the CAN module.

With `Can_DeInit()` all CAN related register values are reset to their default values. Also any pending or ongoing transmission or reception will be aborted.

Each CAN module has `CAN_MESSAGE_BUFFER_COUNT` number of message buffers which can be used either for reception or transmission of CAN messages.

Each message buffer for transmission has to be set up first by calling `Can_SetTransmitMsgBuffer()`.

For receiving CAN messages the function `Can_SetReceiveMsgBuffer()` has to be used.

It is possible to set a callback function which will be notified whenever a message has been received.

Note:

The numbers of the message buffers used in this driver are indexed from 0 to 31 although the „physical addresses" of these buffers are indexed from 1 to 32!

7.4.1 Configuration Structure

7.4.1.1 CAN overall Configuration

The CAN module uses the following configuration structure of the type `stc_can_config_t`:

Type	Field	Possible Values	Description
can_status_chg_… func_ptr_t	pfnStatusCallback	-	Callback function for CAN status changes, can be NULL.
Can_error_func_… ptr_t	pfnErrorCallback	-	Callback function for CAN related errors, can be NULL.
Boolean_t	bDisableAutomatic… Retransmission	TRUE FALSE	Automatic retransmission is disabled Automatic retransmission is enabled
stc_can_bitrate_t	stcBitrate	*(see 7.4.1.2)*	*(see 7.4.1.2)*

7.4.1.2 *Bitrate Configuration*

The Bitrate configuration has the structure type `stc_can_bitrate_t` as below:

Type	Field	Possible Values	Description
uint8_t	u8TimeSegment1	-	Range `CAN_BITRATE_TSEG1_MIN` to `CAN_BITRATE_TSEG2_MAX`
uint8_t	u8TimeSegment2	-	Range `CAN_BITRATE_TSEG2_MIN` to `CAN_BITRATE_TSEG2_MAX` (see define section in *can.h*)
uint8_t	u8SyncJumpWidth	-	Range `CAN_BITRATE_SYNC_JUMP_WIDTH_MIN` to `CAN_BITRATE_SYNC_JUMP_WIDTH_MAX` (see define section in *can.h*)
uint16_t	u16Prescaler	-	Range `PRESCALER_MIN` to `CAN_BITRATE_PRESCALER_MAX` (see define section in *can.h*, divider for the peripheral clock `CLKP2`)
en_can_… prescaler_t	enCanPrescaler	CanPreDiv11 CanPreDiv12 CanPreDiv14 CanPreDiv18 CanPreDiv23 CanPreDiv13 CanPreDiv16 CanPreDiv112 CanPreDiv15 CanPreDiv110	CAN prescaler clock: no division CAN prescaler clock: ½ CAN prescaler clock: ¼ CAN prescaler clock: 1/8 CAN prescaler clock: 2/3 CAN prescaler clock: 1/3 CAN prescaler clock: 1/6 CAN prescaler clock: 1/12 CAN prescaler clock: 1/5 CAN prescaler clock: 1/10
boolean_t	bTouchPrescaler	TRUE FALSE	Prescaler clock is initialized Prescaler clock is not initialized

Note, that the resulting prescaler frequency of the maximum of 16 MHz is checked via `PLCK2` in `Can_Init()`. `ErrorInvalidParameter` is returned if violated.

7.4.2 CAN API

7.4.2.1 *Can_Init ()*

This function initializes one specific CAN module with the parameters provided in the given configuration structure.

After initialization the CAN module has Error, Status and Module-Interrupt enabled and is ready to use.

`Can_Init()` has to be called with the parameter `pstcConfig` of type `stc_can_config_t` the basic CAN settings automatic retransmission, the CAN baudrate, and the error and status change callback function can be set.

All values in `pstcConfig` have to be in valid range (see *can.h* for allowed ranges of dedicated parameters). The error and status change callback functions can be `NULL`. In this case no information of error or status changes will be reported to the API.

To reset and disable the CAN module the function `Can_DeInit()` has to be used.

The resulting CAN prescaler value is checked, if it is within `CAN_MAX_CLK` (normally 16 MHz).

Note, if more than one CAN instance is initialized, `bTouchPrescaler` should be `TRUE` only for the first initialization, because the Prescaler Clock should only be initialized once.

Prototype	
`en_result_t Can_Init( volatile stc_cann_t* pstcCan,` `const stc_can_config_t* pstcConfig );`	
Parameter Name	**Description**
`[in] pstcCan`	CAN instance pointer
`[in] pstcConfig`	Pointer to CAN instance configuration
Return Values	**Description**
`Ok`	Initialization successfully done
`ErrorInvalidParameter`	• `pstcCan == NULL` • `pstcConfig == NULL` • `pstcCanInternData == NULL` • `pstcConfig->stcBitrate.u8TimeSegment1` out of range • `pstcConfig->stcBitrate.u8TimeSegment2` out of range • `pstcConfig->stcBitrate.u8SyncJumpWidth` out of range • `pstcConfig->stcBitrate.u16Prescaler` out of range • `pstcConfig->stcBitrate.enCanPrescaler` wrong enumerator • `pstcConfig->stcBitrate.enCanPrescaler CAN_MAX_CLK` violated

7.4.2.2 *Can_DeInit()*

This function aborts any pending transmission and reception and all CAN related registers are reset to their default values.

Prototype	
`en_result_t Can_DeInit( volatile stc_cann_t* pstcCan );`	
Parameter Name	**Description**
`[in] pstcCan`	CAN instance pointer
Return Values	**Description**
`Ok`	De-Initialization successfully done
`ErrorInvalidParameter`	• `pstcCan == NULL` • `pstcCanInternData == NULL` invalid or deactivated CAN unit (`PDL_PERIPHERAL_ENABLE_CAN`*n*)

7.4.2.3 *Can_SetTransmitMsgBuffer()*

Setting of new values is not possible if a transmission is pending, except remote transmission mode. The callback function `pfnCallback` can be `NULL`, but there will be no notification of a successful transmission. This function has to be called at least once before function `Can_UpdateAndTransmitMsgBuffer()` can be used with the same message buffer index.

With the parameter `stc_can_msg_id_t::pstcMsgId` of type `stc_can_msg_id_t` the API can set the identifier (11 bit or 29 bit length) of the CAN transmit message. It is possible to set a callback function to get notified when a transmission is successfully finished.

`Can_SetTransmitMsgBuffer()` must be called before calling `Can_UpdateAndTransmitMsgBuffer()`. Update or setting of new values of function Can_SetTransmitMsgBuffer() or Can_UpdateAndTransmitMsgBuffer() is not possible if a transmission is pending or ongoing, except remote transmission mode is used.

Prototype	
`en_result_t Can_SetTransmitMsgBuffer( volatile stc_cann_t* pstcCan, uint8_t u8MsgBuf, const stc_can_msg_id_t* pstcMsgId, can_tx_msg_func_ptr_t pfnCallback);`	
Parameter Name	**Description**
`[in] pstcCan`	CAN instance pointer
`[in] u8MsgBuf`	Message buffer index (`0 ... CAN_MESSAGE_BUFFER_COUNT - 1`)
`[in] pstcMsgId`	CAN message identifier.
`[in] pfnCallback`	Callback function to be called after successful transmission, can be `NULL`.
Return Values	**Description**
`Ok`	Message buffer has been successfully initialized.

`ErrorInvalidParameter`	`pstcConfig == NULL`
	`pstcCanInternData == NULL` Invalid or deactivated CAN unit (`PDL_PREIPHERAL_ENABLE_CAN`*n*)
	`pstcMsgId == NULL`
	`u8MsgBuf` out of range

`ErrorOperationInProgress`	A transmission is pending (either wait or call `Can_ResetMsgBuffer()`).

7.4.2.4 Can_UpdateAndTransmitMsgBuffer()

Transmits the message immediately (immediate transmission mode) or on reception of a matching remote frame (remote transmission mode).

Function `Can_SetTransmitMsgBuffer()` must be called before setup the identifier and enable this message buffer.

Prototype	
`en_result_t Can_UpdateAndTransmitMsgBuffer( volatile stc_cann_t* pstcCan, uint8_t u8MsgBuf, const stc_can_msg_data_t* pstcMsgData, en_can_tx_mode_t enTxMode );`	
Parameter Name	**Description**
`[in] pstcCan`	CAN instance pointer
`[in] u8MsgBuf`	Message buffer index (`0 ... CAN_MESSAGE_BUFFER_COUNT - 1`)
`[in] pstcMsgData`	Pointer to CAN message data
`[in] enTxMode`	• CanImmediateTransmit • CanRemoteTransmit
Return Values	**Description**
`Ok`	Message buffer has been successfully updated.

`ErrorInvalidParameter`	`pstcConfig == NULL`
	`pstcCanInternData == NULL` Invalid or deactivated CAN unit (`PDL_PREIPHERAL_ENABLE_CAN`*n*)
	`pstcMsgData == NULL`
	`u8MsgBuf` out of range

`ErrorUninitialized`	`Can_SetTransmitMsgBuffer()` was not called before.
`ErrorOperationInProgress`	A transmission is pending (either wait or call `Can_ResetMsgBuffer()`).

7.4.2.5 Can_UpdateAndTransmitFifoMsgBuffer()

This function has the same parameters and return values like `Can_UpdateAndTransmitMsgBuffer()` exept the `EOB` bit set for FIFO buffer usage.

7.4.2.6 Can_SetReceiveMsgBuffer()

Configures and enables a message buffer for reception. The acceptance filter is set by `pstcMsgBuffer->stcIdentifier` and `u32MsgIdMask`. Each „`0`" bit in `u32MsgIdMask` masks the corresponding bit of the received message ID before comparing it to the configured identifier (set by `pstcMsgBuffer->stcIdentifier`). This allows receiving messages with different identifier. Setting all bits of `u32MsgIdMask` to „`1`" will only accept messages that match the configured identifier.

If extended identifier is used, the `u32MsgIdMask` will also be interpreted as extended mask identifier. If 11 bit identifier is used, than `u32MsgIdMask` will be used as 11 bit mask identifier.

The application must provide a message buffer object (`pstcMsgBuffer`) to be filled with received data.

After reception of a message that passed the acceptance filter, the message"s identifier, data and data length is copied into the provided message buffer and its `bNew` flag is set to `TRUE`.

The message buffer has to be kept valid until this message buffer is reset (`Can_ResetMsgBuffer()`).

A mask identifier has to be set when calling `Can_SetReceiveMsgBuffer()`, the length for the mask identifier will be the same like the one used in `pstcMsgBuffer` (11-bit or 29-bit identifier mask). The extended identification mask bit and the direction mask bit are always set to „1".

The API has to check the `bNew` flag of parameter `pstcMsgBuffer` to get information about if a message has already been received or not. If a new message has been received while `bNew` flag is set (the last received message was not read out by API so far) than the `bOverflow` flag will be set. So, if callback function is not used, the API has to reset the bNew flag when the received message is read out (also the `bOverflow` flag has to be reset) and furthermore.

Prototype	
`en_result_t Can_SetReceiveMsgBuffer( volatile stc_cann_t* pstCan, uint8_t u8MsgBuf, stc_can_msg_t* pstcMsgBuffer, uint32_t u32MsgIdMask, can_rx_msg_func_ptr_t pfnCallback );`	
Parameter Name	**Description**
`[in] pstcCan`	CAN instance pointer
`[in] u8MsgBuf`	Message buffer index (`0 ... CAN_MESSAGE_BUFFER_COUNT - 1`)
`[in] pstcMsgBuffer`	Pointer to CAN message object which defines identifier for acceptance filter.
`[in] u32MsgIdMask`	Mask for identifier acceptance filter and later receives the received message (all „1" disables masking).
`[in] pfnCallback`	Callback function which is called when new CAN message was received and accepted by this message buffer.
Return Values	**Description**
`Ok`	Message buffer has been successfully updated.

`ErrorInvalidParameter`	`pstcConfig == NULL`
	`pstcCanInternData == NULL` Invalid or deactivated CAN unit (`PDL_PREIPHERAL_ENABLE_CAN`*n*)
	`pstcMsgData == NULL`
	`u8MsgBuf` out of range

7.4.2.7 *Can_SetReceiveFifoMsgBuffer()*

This function has the same parameters and return values like `Can_SetReceiveMsgBuffer()` exept the `EOB` bit set for FIFO buffer usage.

7.4.2.8 *Can_ResetMsgBuffer()*

This function stops any message buffer operation i.e. disable it.

In detail it:

- Stops pending transmission (reset `TXRQST` and `NEWDAT` flag):
- Stops reception operation (reset `MSGVAL` flag)
- Resets `RXIE` and `TXIE`
- Clears pointers to external buffers and callback functions

Prototype	
`en_result_t Can_ResetMsgBuffer( volatile stc_cann_t* pstcCan,` `uint8_t u8MsgBuf );`	
Parameter Name	**Description**
`[in] pstcCan`	CAN instance pointer
`[in] u8MsgBuf`	Message buffer index (`0 ... CAN_MESSAGE_BUFFER_COUNT - 1`)
Return Values	**Description**
`Ok`	Message buffer has been successfully updated.

`ErrorInvalidParameter`	`pstcConfig == NULL`
	`pstcCanInternData == NULL` Invalid or deactivated CAN unit (`PDL_PREIPHERAL_ENABLE_CAN`*n*)
	`pstcMsgData == NULL`
	`u8MsgBuf` out of range

7.4.2.9 *CanTxCallback()*

The callback function pointer is defined in the configuration `stc_can_interrupt_handling_t::pfnCanTxInterruptFunction`.

Prototype
`void` *CanTxCallbackName* `( uint8_t u8MsgBuf );`

7.4.2.10 *CanRxCallback()*

The callback function pointer is defined in the configuration `stc_can_interrupt_handling_t::pfnCanRxInterruptFunction`.

Prototype
`void` *CanRxCallbackName* `( uint8_t u8MsgBuf,` `stc_can_msg_t* pstcRxMsg );`

7.4.2.11 *CanStatusCallback()*

This function is called, if `stc_can_config_t::pfnStatusCallback` is defined.

Prototype
`void` *CanStatusCallbackName* `( en_can_status_t enCanStatus );`

The enumerators of en_can_status_t are:

Enumerator	Description
`CanNoError`	No error pending.
`CanStuffError`	More than 5 equal bits in a sequence have occurred in a part of a received message.
`CanFormError`	A fixed format part of a received frame has the wrong format.
`CanAckError`	The message this CAN Core transmitted was not acknowledged by another node.
`CanBit1Error`	While trying to send a recessive bit (1) a dominant level (0) was sampled.
`CanBit0Error`	While trying to send a dominant bit (0) a recessive level (1) was sampled.
`CanCRCError`	The CRC checksum was incorrect.

7.4.2.12 CanErrorCallback()

This function is called, if `stc_can_config_t::pfnErrorCallback` is defined.

Prototype
`void` *CanErrorCallbackName* `( en_can_error_t enCanError );`

The enumerators of `en_can_error_t` are:

Enumerator	Description
`CanBusOff`	The CAN module is in bus-off state.
`CanWarning`	At least one error counter has reached error warning limit of 96.

7.4.3 CAN Examples

The PDL example folder contains one CAN usage examples:

- can_simple Simple CAN communication

7.5 (CLK) Clock Module

Type Definition	–

Configuration Types	`stc_clk_main_config_t`
	`stc_clk_sub_config_t`
	`stc_clk_vbat_config_t`

The Clock Module allows the user to configure the Main Clock, Sub Clock, and the VBAT domain. Furthermore this module provides API functions to switch to different clock and power saving modes.

7.5.1 Main Clock Configuration

The Main Clock has the configuration type `stc_clk_main_config_t`.

Type	Field	Possible Values	Description

`en_clk_source_t`	`enSource`	`ClkMain`
		`ClkSub`
		`ClkHsCr`
		`ClkLsCr`
		`ClkPll`
		`CLkHsPll`
			Main Clock
			Sub Clock
			High-speed CR Clock
			Low-speed CR Clock
			PLL Clock
			High-speed PLL Clock

`boolean_t`	`bEnablePll`	`TRUE` `FALSE`	Enable PLL Disable PLL
`boolean_t`	`bEnableMain…` `Clock`	`TRUE` `FALSE`	Enable Main Clock Disable Main Clock

`en_clk_mode_t`	`enMode`	`ClkRun`
		`ClkSleep`
		`ClkTimer`
		`ClkStop`
		`ClkRtc`
		`ClkDeepRtc`
		`ClkDeepStop`
			Run modes
			Sleep modes
			Timer modes
			Stop modes
			RTC modes
			Deep Standby RTC mode
			Deep Standby Stop mode

`boolean_t`	`bLpmPortHiZ…`
	`State`
		`TRUE`
		
		`FALSE`
			Sets the status of each external port pin to high impedance (Hi-Z) in Timer or Stop mode
			No Hi-Z mode

`en_clk_…`
`baseclkdiv_t`
	`enBaseClkDiv`	`BaseClkDiv1`
		`BaseClkDiv2`
		`BaseClkDiv3`
		`BaseClkDiv4`
		`BaseClkDiv6`
		`BaseClkDiv8`
		`BaseClkDiv16`
		`BaseClkDivErr`
			HCLK division 1/1
			HCLK division 1/2
			HCLK division 1/3
			HCLK division 1/4
			HCLK division 1/6
			HCLK division 1/8
			HCLK division 1/16
			HCLK prohibited setting

`en_clk_…`
`apb0div_t`
	`enAPB0Div`	`ApdDiv1`
		`ApdDiv2`
		`ApdDiv4`
		`ApdDiv8`
			PLCK0 division 1/1
			PLCK0 division 1/2
			PLCK0 division 1/4
			PLCK0 division 1/8

`en_clk_…` `apb1div_t`	`enAPB1Div`	(*see above*)	PCLK1 division (*see above*)
`en_clk_…` `apb2div_t`	`enAPB2Div`	(*see above*)	PCLK2 division (*see above*)
`boolean_t`	`bAPB1Disable`	`TRUE` `FALSE`	Disable PCLK1 Enable PCLK1
`boolean_t`	`bAPB2Disable`	`TRUE` `FALSE`	Disable PCLK2 Enable PCLK2

`en_clk_…`
`mcowaittime_t`
	`enMCOWaitTime`	`MccWaitExp11`
		`MccWaitExp15`
		`MccWaitExp16`
		`MccWaitExp17`
		`MccWaitExp18`
		`MccWaitExp19`
		`MccWaitExp110`
		`MccWaitExp111`
		`MccWaitExp112`
		`MccWaitExp113`
		`MccWaitExp114`
			F(CH) = 4 MHz:
			2^{1} / F(CH) => ~500 ns
			2^{5} / F(CH) => ~8 µs
			2^{6} / F(CH) => ~16 µs
			2^{7} / F(CH) => ~32 µs
			2^{8} / F(CH) => ~64 µs
			2^{9} / F(CH) => ~128 µs
			2^{10} / F(CH) => ~256 µs
			2^{11} / F(CH) => ~512 µs
			2^{12} / F(CH) => ~1.0 ms
			2^{13} / F(CH) => ~2.0 ms

Type	Field	Possible Values	Description
		MccWaitExp115 MccWaitExp117 MccWaitExp119 MccWaitExp121 MccWaitExp123	2^{14} / F(CH) => ~4.0 ms 2^{15} / F(CH) => ~8.0 ms 2^{17} / F(CH) => ~33.0 ms 2^{19} / F(CH) => ~131 ms 2^{21} / F(CH) => ~524 ms 2^{23} / F(CH) => ~2.0 s
en_clk_… pllowaittime_t	enPLLOWaitTime	PlloWaitExp19 PlloWaitExp110 PlloWaitExp111 PlloWaitExp112 PlloWaitExp113 PlloWaitExp114 PlloWaitExp115 PlloWaitExp116	F(CH) = 4 MHz: 2^{9} / F(CH) => ~128 µs 2^{10} / F(CH) => ~256 µs 2^{11} / F(CH) => ~512 µs 2^{12} / F(CH) => ~1.02 ms 2^{13} / F(CH) => ~2.05 ms 2^{14} / F(CH) => ~4.10 ms 2^{15} / F(CH) => ~8.20 ms 2^{16} / F(CH) => ~16.4 ms
uint8_t	u8PllK	(*see datasheet for min/max values*)	PLL input clock frequency division ratio (PLLK)
uint8_t	u8PllM	(*see datasheet for min/max values*)	PLL VCO clock frequency division ratio (PLLM)
uint8_t	u8PllN	(*see datasheet for min/max values*)	PLL feedback frequency division ratio (PLLN)
func_ptr_t	pfnHook	-	Pointer to user hook function, when API function includes polling loops

If CLK interrupts are enabled in *pdl_user.h* the configuration gets the following additional settings. Note that these settings are not available (i.e. not compiled), if CLK interrupts are disabled.

Type	Field	Possible Values	Description
boolean_t	bPllIrq	TRUE FALSE	Enables PLL oscillation stabilization completion interrupt Disable PLL interrupt
boolean_t	bMcoIrq	TRUE FALSE	Enables Main Clock oscillation stabilization completion interrupt Disable Main Clock interrupt
func_ptr_t	pfnPllStabCb	-	Pointer to PLL stabilization callback function
func_ptr_t	pfnPllStabCb	-	Pointer to Main Clock stabilization callback function

7.5.2 Sub Clock Configuration

The Sub Clock has the configuration type `stc_clk_sub_config_t`.

Type	Field	Possible Values	Description
`boolean_t`	`bEnableSubClock`	`TRUE` `FALSE`	Enable Sub Clock Disable Sub Clock

`en_clk_…`
`scowaittime_t`
	`enSCOWaitTime`	
		`ScoWaitExp10`
		`ScoWaitExp11`
		`ScoWaitExp12`
		`ScoWaitExp13`
		`ScoWaitExp14`
		`ScoWaitExp15`
		`ScoWaitExp16`
		`ScoWaitExp17`
		`ScoWaitExp18`
		`ScoWaitExp19`
		`ScoWaitExp20`
		`ScoWaitExp21`
		`ScoWaitErr`
			F(CL) = 32768 Hz
			2^{10} / F(CL) => ~10.3 ms
			2^{11} / F(CL) => ~20.5 ms
			2^{12} / F(CL) => ~41 ms
			2^{13} / F(CL) => ~82 ms
			2^{14} / F(CL) => ~164 ms
			2^{15} / F(CL) => ~327 ms
			2^{16} / F(CL) => ~655 ms
			2^{17} / F(CL) => ~1.31 s
			2^{18} / F(CL) => ~2.62 s
			2^{19} / F(CL) => ~5.24 s
			2^{20} / F(CL) => ~10.48 s
			2^{21} / F(CL) => ~20.96 s
			Prohibited Setting

If CLK interrupts are enabled in *pdl_user.h* the configuration gets the following additional settings. Note that these settings are not available (i.e. not compiled), if CLK interrupts are disabled.

Type	Field	Possible Values	Description

`boolean_t`	`bScoIrq`	`TRUE`
		
		`FALSE`
			Enables Sub Clock oscillation stabilization completion interrupt
			Disable Sub Clock interrupt

`func_ptr_t`	`pfnScoStabCb`	-	Pointer to Sub Clock stabilization callback function

7.5.3 VBAT Domain Configuration

The Sub Clock has the configuration type `stc_clk_vbat_config_t`.

Type	Field	Possible Values	Description
boolean_t	bLinkClock	TRUE FALSE	32 kHz oscillation circuit linked with clock oscillation circuit 32 kHz oscillation circuit operates independly
uint8_t	u8VbClockDiv	(*see datasheet for min/max values*)	Value for VB_CLKDIV register
en_clk_… current_t	enClkSustain… Current	Clk0nA Clk135nA Clk195nA Clk385nA Clk445nA Clk510nA ClkErrorCurrent	< 0 nA sustain/boost, not allowed, if Sub Clock is enabled < 135 nA < 195 nA < 384 nA < 445 nA, initial value for current sustain < 510 nA, initial value for current boost Errornous setting (for read-out functions)
en_clk_… current_t	enClkBoost… Current	*(see above)*	*(see above)*
en_clk_… boost_… time_t	enClkBoost… Time	ClkBoost50ms ClkBoost63ms ClkBoost125ms ClkBoost250ms	Boost time 50 ms (initial setting) Boots time 63 ms Boots time 125 ms Boots time 250 ms
en_clk_… vbat_… pins_t	enVbatPins	ClkVbatInput ClkVbatOutputL ClkVbatOutputH	VBAT pin input function VBAT pin output "low" function VBAT pin output "high" function
boolean_t	bVbP48… Peripheral	TRUE FALSE	P48 pin (VREGCTL) as I/O of peripheral function P48 pin as GPIO
boolean_t	bVbP49… Peripheral	TRUE FALSE	P49 pin (VWAKEUP) as I/O of peripheral function P49 pin as GPIO
boolean_t	bVbP47… Peripheral	TRUE FALSE	P47 pin (X1A) as I/O of peripheral function P47 pin as GPIO
boolean_t	bVbP46… Peripheral	TRUE FALSE	P46 pin (X0A) as I/O of peripheral function P46 pin as GPIO
boolean_t	bVbP48PullUp	TRUE FALSE	Enable pull-up on P48 (VREGCTL) Disable pull-up
boolean_t	bVbP49PullUp	TRUE FALSE	Enable pull-up on P49 (VWAKEUP) Disable pull-up
boolean_t	bVbP47PullUp	TRUE FALSE	Enable pull-up on P47 (X1A) Disable pull-up
boolean_t	bVbP46PullUp	TRUE FALSE	Enable pull-up on P46 (X0A) Disable pull-up
en_clk_… vbat_… pins_ddr_t	enVbP48InOut	ClkVbatInput ClkVbatOutputL ClkVbatOutputH	VREGCTL pin input direction VREGCTL pin output “low” function VREGCTL pin output “high” function
en_clk_… vbat_… pins_ddr_t	enVbP49InOut	ClkVbatInput ClkVbatOutputL ClkVbatOutputH	VWAKEUP pin input direction VWAKEUP pin output “low” function VWAKEUP pin output “high” function
en_clk_… vbat_…	enVbP47InOut	ClkVbatInput ClkVbatOutputL	X1A pin input direction X1A pin output “low” function

Type	Field	Possible Values	Description
pins_ddr_t		ClkVbatOutputH	X1A pin output "high" function
en_clk_… vbat_… pins_ddr_t	enVbP46InOut	ClkVbatInput ClkVbatOutputL ClkVbatOutputH	X0A pin input direction X0A pin output "low" function X0A pin output "high" function
boolean_t	bVbP48Open… Drain	TRUE FALSE	P48 (VREGCTL) Pseudo Open Drain enabled Pseudo Drain disabled
boolean_t	bVbP49Open… Drain	TRUE FALSE	P49 (VWAKEUP) Pseudo Open Drain enabled Pseudo Drain disabled

7.5.4 CLK API

7.5.4.1 Clk_MainGetParameters()

This function sets the 'current' elements of the configuration according the main and PLL clock registers.

Note: This function overwrites any configuration. To avoid this, a second configuration structure like 'ConfigRecent' may be used.

Also Note: This function does not set any hook function pointer! If this function is used to get the current main and PLL clock settings as a base for new settings, a possible hook function pointer must be set explicitly after copying the configuration!

Prototype	
en_result_t Clk_MainGetParameters(stc_clk_main_config_t* pstcConfig);	
Parameter Name	**Description**
[out] pstcConfig	Pointer to Read-Out-Configuration
Return Values	**Description**
Ok	Main Clock settings has been read-out successfully
ErrorInvalidParameter	pstcConfig == NULL
ErrorInvalidMode	Illegal clock mode has been detected

7.5.4.2 Clk_SubGetParameters()

This function sets the 'current' elements of the configuration according the Sub clock registers.

Note: This function overwrites any configuration. To avoid this, a second configuration structure like 'ConfigRecent' may be used.

Also Note: This function does not set any hook function pointer! If this function is used to get the current Sub clock settings as a base for new settings, a possible hook function pointer must be set explicitly after copying the configuration!

Prototype	
en_result_t Clk_SubGetParameters(stc_clk_main_config_t* pstcConfig);	
Parameter Name	**Description**
[out] pstcConfig	Pointer to Read-Out-Configuration
Return Values	**Description**
Ok	Sub Clock settings has been read-out successfully
ErrorInvalidParameter	pstcConfig == NULL
ErrorInvalidMode	Illegal clock mode has been detected

7.5.4.3 Clk_VbatGetParameters()

This function sets the 'current' elements of the configuration according the VBAT clock registers.

Note: This function overwrites any configuration. To avoid this, a second configuration structure like 'ConfigRecent' may be used.

Also Note: This function does not set any hook function pointer! If this function is used to get the current VBAT settings as a base for new settings, a possible hook function pointer must be set explicitly after copying the configuration!

Prototype	
en_result_t Clk_VbatGetParameters(stc_clk_vbat_config_t* pstcConfig);	
Parameter Name	**Description**
[out] pstcConfig	Pointer to Read-Out-Configuration
Return Values	**Description**
Ok	VBAT settings has been read-out successfully
ErrorInvalidParameter	pstcConfig == NULL
ErrorInvalidMode	Illegal VBAT mode has been detected

7.5.4.4 Clk_SetDividers()

This function sets the clock dividers.

It is strongly recommended to disable any resource of its corresponding bus, if the bus division setting is changed! Malfunction of the resources may result (i.e. wrong baud rates, lost data, etc.).

Note: Do not access any of the resource registers, if the corresponding resource's bus is disabled! An immediate bus fault exception will occur in this case!

Prototype	
`en_result_t Clk_SetDividers(stc_clk_main_config_t* pstcConfig);`	
Parameter Name	**Description**
`[in] pstcConfig`	Clock configuration parameters
Return Values	**Description**
`Ok`	Dividers set
`ErrorInvalidParameter`	`pstcConfig == NULL` or illegal divider setting(s)

7.5.4.5 *Clk_MainSetStabilizationWaitTime()*

This function sets the stabilization wait time for the Main Clock.

Prototype	
`en_result_t Clk_MainSetStabilizationWaitTime(stc_clk_main_config_t* pstcConfig);`	
Parameter Name	**Description**
`[in] pstcConfig`	Clock configuration parameters
Return Values	**Description**
`Ok`	Dividers set
`ErrorInvalidParameter`	`pstcConfig == NULL` or illegal timing setting

7.5.4.6 *Clk_WaitForMainOscillator()*

This function waits for the Main Oscillator stabilization via polling. `PDL_WAIT_LOOP_HOOK()` is called during polling. It should be called, if the system needs a stable main clock (i.e. for communication or switching to PLL clock, etc.).

Prototype	
`en_result_t Clk_WaitForMainOscillator(uint32_t u32MaxTimeOut);`	
Parameter Name	**Description**
`[in] u32MaxTimeOut`	Time out counter start value
Return Values	**Description**
`Ok`	Main Clock stabilized
`ErrorInvalidParameter`	Main Clock not stabilized after timeout count

7.5.4.7 Clk_WaitForPllOscillator()

This function waits for the PLL Oscillator stabilization via polling. `PDL_WAIT_LOOP_HOOK()` is called during polling. It should be called, if the system needs a stable main clock (i.e. for communication or switching to PLL clock, etc.).

Prototype	
`en_result_t Clk_WaitForPllOscillator(uint32_t u32MaxTimeOut);`	
Parameter Name	**Description**
`[in] u32MaxTimeOut`	Time out counter start value
Return Values	**Description**
`Ok`	PLL Clock stabilized
`ErrorInvalidParameter`	PLL Clock not stabilized after timeout count

7.5.4.8 Clk_WaitForClockSourceReady()

This function waits for a clock source, meaning a clock source already available or a clock source transition to be expected to be ready soon or already available. `PDL_WAIT_LOOP_HOOK()` is called during polling.

Note: This function is not needed to be called, if the user has performed the stabilization wait time for the desired source clock before. For safety reasons, this function can be called anyhow with a small timeout count (<<10).

Prototype	
`en_result_t Clk_WaitForClockSourceReady(en_clk_source_t enSource,` `uint32_t u32MaxTimeOut);`	
Parameter Name	**Description**
`[in] enSource`	Clock Source to be checked
`[in] u32MaxTimeOut`	Time out counter start value
Return Values	**Description**
`Ok`	Clock Source ready
`ErrorTimeout`	Clock Source not ready within time out count
`ErrorInvalidParameter`	Not a valid Clock Mode to be checked

7.5.4.9 Clk_MainOscillatorReady()

This function checks the availability of a stable Main Clock.

Prototype	
`en_result_t Clk_MainOscillatorReady(void);`	
Return Values	**Description**
`Ok`	Main Clock stabilized
`ErrorNotReady`	Main Clock not stabilized yet

7.5.4.10 Clk_SubOscillatorReady()

This function checks the availability of a stable Sub Clock.

Prototype	
`en_result_t Clk_SubOscillatorReady(void);`	
Return Values	**Description**
`Ok`	Sub Clock stabilized
`ErrorNotReady`	Sub Clock not stabilized yet

7.5.4.11 Clk_PllOscillatorReady()

This function checks the availability of a stable PLL Clock.

Prototype	
`en_result_t Clk_PllOscillatorReady(void);`	
Return Values	**Description**
`Ok`	PLL Clock stabilized
`ErrorNotReady`	PLL Clock not stabilized yet

7.5.4.12 Clk_SetSource()

This function sets the clock source and performs transition, if wanted.

Prototype	
`en_result_t Clk_SetSource(stc_clk_main_config_t* pstcConfigMain, stc_clk_sub_config_t* pstcConfigSub);`	
Parameter Name	**Description**
`[in] pstcConfigMain`	Pointer to Main Clock configuration parameters
`[in] pstcConfigSub`	Pointer to Sub Clock configuration parameters
Return Values	**Description**
`Ok`	Clock source set
`ErrorInvalidParameter`	`pstcConfig == NULL` or Illegal mode
`ErrorInvalidMode`	Clock setting not possible

7.5.4.13 Clk_SetMode()

This function sets the clock mode and performs the transition. For individual settings (such as USB and CAN low power configuration) a hook function is called after setting `SLEEPDEEP` to `1` and final WFI instruction.

This function is only called, if the function pointer is unequal to `NULL`. Additionally the ports will go into Hi-Z state, if `stc_clk_config_t::bPortHiZState` is `TRUE`.

Prototype	
`en_result_t Clk_SetMode(stc_clk_main_config_t* pstcConfig);`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to Clock configuration parameters
Return Values	**Description**
`Ok`	Clock mode set
`ErrorInvalidParameter`	`pstcConfig == NULL` or Illegal mode

7.5.4.14 *Clk_DisableSubClock()*

This function easily disables the Sub Clock. No configuration is needed.

Prototype	
`en_result_t Clk_DisableSubClock(void);`	
Return Value	**Description**
`Ok`	Sub Clock disabled

7.5.4.15 *Clk_EnableMainClock()*

This function easily enables the Main Clock. No configuration is needed. For stabilization wait time `Clk_WaitForMainOscillator()` has to be called afterwards.

Prototype	
`en_result_t Clk_EnableMainClock (void);`	
Return Value	**Description**
`Ok`	Main Clock enabled

7.5.4.16 *Clk_EnablePllClock()*

This function easily enables the PLL Clock. It uses the PLL configuration from `stc_clk_main_config_t`.

For stabilization wait time `Clk_WaitForPllOscillator()` has to be called afterwards.

Prototype	
`en_result_t Clk_EnablePllClock(stc_clk_main_config_t* pstcConfigMain)`	
Parameter Name	**Description**
`[in] pstcConfigMain`	Pointer to Main/PLL Clock configuration parameters
Return Values	**Description**
`Ok`	PLL enabled
`ErrorInvalidParameter`	PLL settings wrong or `pstcConfigMain == NULL`
`ErrorOperationInProgress`	PLL already running

7.5.4.17 *Clk_DisableMainClock()*

This function easily disables the Main Clock. No configuration is needed.

Prototype	
`en_result_t Clk_DisableMainClock(void);`	
Return Value	**Description**
`Ok`	Main Clock disabled

7.5.4.18 *Clk_DisablePllClock()*

This function easily disables the PLL Clock. No configuration is needed.

Prototype	
`en_result_t Clk_DisablePllClock(void);`	
Return Value	**Description**
`Ok`	PLL Clock disabled

7.5.4.19 *Clk_SetIrq()*

This function enables or disables the clock stabilization interrupts according configuration. This function is only available if `PDL_INTERRUPT_ENABLE_CLK == PDL_ON`.

Prototype

```
en_result_t Clk_SetIrq( stc_clk_main_config_t* pstcConfigMain,
                        stc_clk_sub_config_t*  pstcConfigSub,
                        boolean_t              bTouchNvic );
```

Parameter Name	Description
`[in] pstcConfigMain`	Pointer to Main/PLL Clock configuration parameters
`[in] pstcConfigSub`	Pointer to Sub Clock configuration parameters
`[in] bTouchNvic`	`TRUE`: Touch NVIC registers, `FALSE`: Do not touch NVIC registers
Return Values	**Description**
`Ok`	Interrupts enabled
`ErrorInvalidParameter`	`pstcConfigMain == NULL` and/or `pstcConfigSub == NULL`

7.5.4.20 Clk_ClockVbatInit()

This function initializes the Sub Clock with VBAT setting. It does not start the clock.

Prototype	
`en_result_t Clk_ClockVbatInit( stc_clk_vbat_config_t* pstcConfig );`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to Sub Clock/VBAT configuration parameters
Return Values	**Description**
`Ok`	Interrupts enabled
`ErrorInvalidParameter`	`pstcConfig == NULL`

7.5.4.21 Clk_EnableSubClock()

This function easily enables the Sub Clock. No configuration is needed. For stabilization wait time `Clk_WaitForSubOscillator()` has to be called afterwards.

Prototype	
`en_result_t Clk_EnableSubClock( void );`	
Return Values	**Description**
`Ok`	Sub Clock enabled
`ErrorTimeout`	Data transition to VBAT domain failed

7.5.4.22 Clk_SubSetStabilizationWaitTime()

This function sets the stabilization wait time for the Sub Clock.

Prototype	
`en_result_t Clk_SubSetStabilizationWaitTime( stc_clk_sub_config_t* pstcConfig );`	
Parameter Name	**Description**
`[in] pstcConfig`	Sub Clock configuration parameters
Return Values	**Description**
`Ok`	Time set
`ErrorInvalidParameter`	`pstcConfig == NULL` or illegal timing setting

7.5.4.23 Clk_WaitForSubOscillator()

This function waits for the Sub Oscillator stabilization via polling. `PDL_WAIT_LOOP_HOOK()` is called during polling. It should be called, if the system needs a stable sub clock (i.e. for communication).

Prototype	
`en_result_t Clk_WaitForSubOscillator( uint32_t u32MaxTimeOut );`	
Parameter Name	**Description**
`[in] u32MaxTimeOut`	Time out counter start value
Return Values	**Description**
`Ok`	Clock stabilized
`ErrorTimeout`	Clock not stabilized after timeout count

7.5.4.24 Clk_RequestVccPowerDown()

This function sets the 32 kHz oscillation control disable of the RTC.

Prototype	
`en_result_t Clk_RequestVccPowerDown( void );`	
Return Value	**Description**
`Ok`	Disable set

7.5.4.25 Clk_PeripheralClockEnable()

This function sets the corresponding bit in the `CKEN`*n* register to enable the clock of a peripheral.

Prototype	
`en_result_t Clk_PeripheralClockEnable( en_clk_gate_peripheral_t enPeripheral );`	
Parameter Name	**Description**
`[in] enPeripheral`	Enumerator of a peripheral, see below for peripheral list
Return Values	**Description**
`Ok`	Peripheral clock enabled
`ErrorInvalidParameter`	Peripheral enumerator does not exist

List of peripheral clock gate enumerators of type of `en_clk_gate_peripheral_t`:

Enumerator	Description
ClkGateGpio	GPIO clock gate
ClkGateExtif	External bus interface clock gate
ClkGateDma	DMA clock gate
ClkGateAdc0	ADC0 clock gate
ClkGateAdc1	ADC1 clock gate
ClkGateAdc2	ADC2 clock gate
ClkGateAdc3	ADC3 clock gate
ClkGateMfs0	MFS0 clock gate
ClkGateMfs1	MFS1 clock gate
ClkGateMfs2	MFS2 clock gate
ClkGateMfs3	MFS3 clock gate
ClkGateMfs4	MFS4 clock gate
ClkGateMfs5	MFS5 clock gate
ClkGateMfs6	MFS6 clock gate
ClkGateMfs7	MFS7 clock gate
ClkGateMfs8	MFS8 clock gate
ClkGateMfs9	MFS9 clock gate
ClkGateMfs10	MFS10 clock gate
ClkGateMfs11	MFS11 clock gate
ClkGateMfs12	MFS12 clock gate
ClkGateMfs13	MFS13 clock gate
ClkGateMfs14	MFS14 clock gate
ClkGateMfs15	MFS15 clock gate
ClkGateQprc0	QPRC0 clock gate
ClkGateQprc1	QPRC1 clock gate
ClkGateQprc2	QPRC2 clock gate
ClkGateQprc3	QPRC3 clock gate
ClkGateMft0	MFT0, PPG0/2/4/6 clock gate
ClkGateMft1	MFT1, PPG8/10/12/14 clock gate
ClkGateMft2	MFT2, PPG16/18/20/22 clock gate
ClkGateMft3	MFT3, PPG24/26/28/30 clock gate
ClkGateBt0	BT0/1/2/3 clock gate
ClkGateBt4	BT4/5/6/7 clock gate
ClkGateBt8	BT8/9/10/11 clock gate
ClkGateSdIf	SD Card I/F clock gate
ClkGateCan0	CAN0 clock gate
ClkGateCan1	CAN1 clock gate
ClkGateUsb0	USB0 clock gate
ClkGateUsb1	USB1 clock gate

7.5.4.26 Clk_PeripheralGetClockState()

This function reads out the corresponding bit in the CKEN*n* register.

Prototype	
`boolean_t Clk_PeripheralGetClockState( en_clk_gate_peripheral_t enPeripheral );`	
Parameter Name	**Description**
`[in] enPeripheral`	Enumerator of a peripheral, see above for peripheral list (7.5.4.25)
Return Values	**Description**
`TRUE`	Peripheral clock enabled
`FALSE`	Peripheral clock not enabled or peripheral not existing

7.5.4.27 Clk_PeripheralClockDisable()

This function clears the corresponding bit in the CKEN*n* register to enable the clock of a peripheral

Prototype	
`en_result_t Clk_PeripheralClockDisable( en_clk_gate_peripheral_t enPeripheral );`	
Parameter Name	**Description**
`[in] enPeripheral`	Enumerator of a peripheral, see above for peripheral list (7.5.4.25)
Return Values	**Description**
`Ok`	Peripheral clock disabled
`ErrorInvalidParameter`	Peripheral enumerator does not exist

7.5.4.28 Clk_PeripheralSetReset()

This function sets the corresponding bit in the MRST*n* register to set a peripheral in reset state.

Prototype	
`en_result_t Clk_PeripheralSetReset( en_clk_reset_peripheral_t enPeripheral );`	
Parameter Name	**Description**
`[in] enPeripheral`	Enumerator of a peripheral, see below for peripheral list
Return Values	**Description**
`Ok`	Peripheral in reset state
`ErrorInvalidParameter`	Peripheral enumerator does not exist

List of peripheral reset enumerators of type of `en_clk_reset_peripheral_t`:

Enumerator	Description
ClkGateExtif	External bus interface clock gate
ClkGateDma	DMA clock gate
ClkGateAdc0	ADC0 clock gate
ClkGateAdc1	ADC1 clock gate
ClkGateAdc2	ADC2 clock gate
ClkGateAdc3	ADC3 clock gate
ClkGateMfs0	MFS0 clock gate
ClkGateMfs1	MFS1 clock gate
ClkGateMfs2	MFS2 clock gate
ClkGateMfs3	MFS3 clock gate
ClkGateMfs4	MFS4 clock gate
ClkGateMfs5	MFS5 clock gate
ClkGateMfs6	MFS6 clock gate
ClkGateMfs7	MFS7 clock gate
ClkGateMfs8	MFS8 clock gate
ClkGateMfs9	MFS9 clock gate
ClkGateMfs10	MFS10 clock gate
ClkGateMfs11	MFS11 clock gate
ClkGateMfs12	MFS12 clock gate
ClkGateMfs13	MFS13 clock gate
ClkGateMfs14	MFS14 clock gate
ClkGateMfs15	MFS15 clock gate
ClkGateQprc0	QPRC0 clock gate
ClkGateQprc1	QPRC1 clock gate
ClkGateQprc2	QPRC2 clock gate
ClkGateQprc3	QPRC3 clock gate
ClkGateMft0	MFT0, PPG0/2/4/6 clock gate
ClkGateMft1	MFT1, PPG8/10/12/14 clock gate
ClkGateMft2	MFT2, PPG16/18/20/22 clock gate
ClkGateMft3	MFT3, PPG24/26/28/30 clock gate
ClkGateBt0	BT0/1/2/3 clock gate
ClkGateBt4	BT4/5/6/7 clock gate
ClkGateBt8	BT8/9/10/11 clock gate
ClkGateSdIf	SD Card I/F clock gate
ClkGateCan0	CAN0 clock gate
ClkGateCan1	CAN1 clock gate
ClkGateUsb0	USB0 clock gate
ClkGateUsb1	USB1 clock gate

7.5.4.29 Clk_PeripheralClearReset()

This function clears the corresponding bit in the MRST*n* register to release a peripheral from reset state.

Prototype	
`en_result_t Clk_PeripheralClearReset( en_clk_reset_peripheral_t enPeripheral );`	
Parameter Name	**Description**
`[in] enPeripheral`	Enumerator of a peripheral, see above for peripheral list (7.5.4.28)
Return Values	**Description**
`Ok`	Peripheral reset released
`ErrorInvalidParameter`	Peripheral enumerator does not exist

7.5.4.30 *Clk_SwitchToMainClock()*

This function sets `HCLK` to Main Clock oscillation.

Prototype	
`en_result_t Clk_SwitchToMainClock( void );`	
Return Values	**Description**
`Ok`	`HCLK` is Main Clock oscillation
`ErrorInvalidMode`	Main clock not available or not enabled
`ErrorNotReady`	Main clock enabled but oscillation not stable yet

7.5.4.31 *Clk_SwitchToMainPllClock()*

This function sets `HCLK` to PLL Clock oscillation. The function expects enabled Main and PLL Clock oscillation.

Prototype	
`en_result_t Clk_SwitchToMainPllClock( void );`	
Return Values	**Description**
`Ok`	`HCLK` is PLL Clock oscillation
`ErrorInvalidMode`	Main/PLL clock not available or not enabled
`ErrorNotReady`	Main/PLL clock enabled but oscillation not stable yet

7.5.4.32 *Clk_SwitchToSubClock()*

This function sets `HCLK` to Sub Clock oscillation. The function expects enabled Sub Clock oscillation.

Prototype	
`en_result_t Clk_SwitchToSubClock( void );`	
Return Values	**Description**
`Ok`	`HCLK` is Sub Clock oscillation
`ErrorInvalidMode`	Sub Clock not available or not enabled

7.5.4.33 *Clk_SwitchToLsCrClock()*

This function sets `HCLK` to Low Speed CR Clock oscillation.

Prototype	
`en_result_t Clk_SwitchToLsCrClock( void );`	
Return Values	**Description**
`Ok`	`HCLK` is Low Speed CR Clock oscillation

7.5.4.34 *Clk_SwitchToHsCrClock()*

This function sets `HCLK` to High Speed CR Clock oscillation.

Prototype	
`en_result_t Clk_SwitchToHsCrClock( void );`	
Return Values	**Description**
`Ok`	`HCLK` is High Speed CR Clock oscillation

7.5.4.35 *Clk_SwitchToHsCrPllClock()*

This function sets HCLK to High Speed CR PLL Clock oscillation. Note: If PLL is enabled by this function no wait for stabilization is performed.

Prototype	
`en_result_t Clk_SwitchToHsCrPllClock( void );`	
Return Values	**Description**
`Ok`	`HCLK` is High Speed CR PLL Clock oscillation
`ErrorInvalidMode`	High Speed CR PLL Clock not available or not enabled

7.5.5 CLK Examples

The PDL example folder contains two CLK usage examples:

- clk_gating How to use clock gating
- clk_init Example for alternative start-up code

7.6 (CR) CR Clock Trimming

Type Definition	–
Configuration Types	–
Address Operator	–

The high-speed CR trimming function consitsts of the frequency trimming setup unit and temperature trimming setup unit.

For frequency trimming, call Cr_SetFreqTrimmingData() to set the trimming value. And get the value by calling Cr_SetFreqTrimmingData().

For temperature trimming, call Cr_SetTempTrimmingData() to set the trimming value. And get the value by calling Cr_GetTempTrimmingData().

Call Cr_SetFreqDiv() to set the frequency division of high-speed CR oscillation and it can be a input of base timer.

7.6.1 Configuration Structure

None

7.6.2 CR API

7.6.2.1 Cr_SetFreqDiv ()

This function sets the frequency division of CR output to Base timer.

Prototype	
`en_result_t Cr_SetFreqDiv(en_cr_freq_div_t enCrDiv);`	
Parameter Name	**Description**
`[in] enCrDiv`	CR division param.
Return Values	**Description**
`Ok`	Division set done.
`ErrorInvalidParameter`	`enCrDiv > CrFreqDivBy512`

7.6.2.2 Cr_SetTempTrimmingData ()

This function sets CR temperature trimming register.

Prototype	
`en_result_t Cr_SetTempTrimmingData(uint8_t u8Data);`	
Parameter Name	**Description**
`[in] u8Data`	Temperature trimming value, only Bit[4:0] is valid.
Return Values	Description
`Ok`	Set value done.

7.6.2.3 *Cr_GetTempTrimmingData ()*

This function gets CR temperature trimming register.

Prototype	
`uint8_t Cr_GetTempTrimmingData(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
`value`	Temperature trimming value.

7.6.2.4 *Cr_SetFreqTrimmingData ()*

This function sets CR frequency trimming register.

Prototype	
`en_result_t Cr_SetFreqTrimmingData(uint16_t u16Data);`	
Parameter Name	**Description**
`[in] u16Data`	Temperature trimming value, only Bit[9:0] is valid.
Return Values	**Description**
`Ok`	Set value done.

7.6.2.5 *Cr_GetFreqTrimmingData ()*

This function gets frequency trimming register.

Prototype	
`uint16_t Cr_GetFreqTrimmingData(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
`value`	Temperature trimming value.

7.7 (CRC) Cyclic Redundancy Check

Type Definition	-
Configuration Type	`stc_crc_config_t`
Address Operator	-

The CRC APIs provide a set of functions for accessing the CRC block.

Note: The user need to take care of the endianess of the data.

7.7.1 CRC Configuration Structure

The argument of `Crc_Init()` is a pointer to a structure of the CRC configuration. The type of the structure is `stc_crc_config_t`. The members of `stc_crc_config_t` are:

Type	Field	Possible Values	Description
en_crc_mode_t	enMode	Crc16 Crc32	CCITT CRC16 standard IEEE-802.3 CRC3 Ethernet standard
boolean_t	bUseDma	TRUE FALSE	DMA usage needs DMA driver Push functions used
boolean_t	bFinalXor	TRUE FALSE	CRC result as XOR value CRC result not as XOR value
boolean_t	bResultLsbFirst	TRUE FALSE	Result bit order: LSB first Result bit order: MSB first
boolean_t	bResultLittleEndian	TRUE FALSE	Result byte order: Little endian Result byte order: Big endian
boolean_t	bDataLsbFirst	TRUE FALSE	Feed data bit order: LSB first Feed data bit order: MSB first
boolean_t	bDataLittleEndian	TRUE FALSE	Feed data byte order: Little endian Feed data byte order: Big endian
uint32_t	u32CrcInitValue	0…0xFFFFFFFF	Initial CRC value

7.7.2 API Reference

7.7.2.1 *Crc_Init()*

Initializes the CRC block.

Prototype	
en_result_t Crc_Init(stc_crc_config_t* pstcConfig)	
Parameter Name	**Description**
[in] pstcConfig	A pointer to a structure of the CRC configuration
Return Values	**Description**
Ok	Initialization ended with no error
ErrorInvalidParameter	pstcConfig == NULL The parameter is out of range

7.7.2.2 *Crc_DeInit()*

Deinitializes the CRC block.

Prototype
void Crc_DeInit(void)

7.7.2.3 Crc_Push8()

Pushes 8-bit data to the CRC block.

Prototype	
`void Crc_Push8(uint8_t u8DataToPush)`	
Parameter Name	**Description**
`[in] u8DataToPush`	8-Bit data to push to the CRC block

7.7.2.4 Crc_Push16()

Pushes 16-bit data to the CRC block.

Prototype	
`void Crc_Push16(uint16_t u16DataToPush)`	
Parameter Name	**Description**
[in] `u16DataToPush`	16-Bit data to push to the CRC block

7.7.2.5 Crc_Push32()

Pushes 32-bit data to the CRC block:

Prototype	
`void Crc_Push32(uint32_t u32DataToPush)`	
Parameter Name	**Description**
[in] `u32DataToPush`	32-Bit data to push to the CRC block

7.7.2.6 Crc_ReadResult()

Returns a 32-bit CRC calcuration result.

Prototype	
`uint32_t Crc_ReadResult(void)`	
Return Values	**Description**
`uint32_t`	CRC calcuration result

7.7.3 Example Code

The example software is in *\example\crc\crc_16_32*. The example software does not use DMA.

```
#include "crc/crc.h"

...
static const uint32_t au32ConstData[64] = {
  0x6393B370, 0xF2BB4FC0, 0x6D793D2C, 0x508B2092,
   ...
  0x71F342AA, 0x9BAFA978, 0xDE2F8EFA, 0xC3FF71FE
};
...
function
{
  stc_crc_config_t       stcCrcConfig;
  uint8_t                u8Count;
  ...
  stcCrcConfig.enMode                = Crc32;          // CRC32 used
  stcCrcConfig.bUseDma               = FALSE;          // No DMA used
  stcCrcConfig.bFinalXor             = FALSE;          // No final XOR
  stcCrcConfig.bResultLsbFirst       = FALSE;          // Result in MSB first
  stcCrcConfig.bResultLittleEndian   = TRUE;           // Result little endian used
  stcCrcConfig.bDataLsbFirst         = FALSE;          // Data MSB first
  stcCrcConfig.bDataLittleEndian     = TRUE;           // Data little endian used
  stcCrcConfig.u32CrcInitValue       = 0xFFFFFFFF;     // Start value
  ...
  ...
  if (Ok != Crc_Init(&stcCrcConfig))
  {
    // some code here ...
    while(1);
  }
 // Test 32-bit pushing
 for (u8Count = 0; u8Count < 64; u8Count++)
  {
    Crc_Push32(au32ConstData[u8Count]);
  }
  // Check result
  u32CrcResult = Crc_ReadResult();
  if (0x6AEA5AF1 != u32CrcResult)
  {
    // some code here ...
  }
  Crc_DeInit();
  ...
  // Test CRC-16
  stcCrcConfig.enMode = Crc16;
  if (Ok != Crc_Init(&stcCrcConfig))
  {
    // some code here ...
    while(1);
  }
 // Test 32-bit pushing
 for (u8Count = 0; u8Count < 64; u8Count++)
  {
    Crc_Push32(au32ConstData[u8Count]);
  }
  / Check result
  u32CrcResult = Crc_ReadResult();
  if (0x14B40000 != u32CrcResult)
  {
    // some code here ...
  }
  Crc_DeInit();
  ...
}
```

7.8 (CSV) Clock Supervisor

Type Definition	-
Configuration Types	stc_csv_status_t
Address Operator	-

What is CSV?

The CSV module includes CSV and FCS function. CSV uses the high and low CR to monitor main and sub clock, if the abnormal state is detected, a reset occurs. FCS uses high-speed CR to monitor the main clock, a range can be set in advance. If the monitor cycle exeeds the preset range, either interrupt or reset will occurs.

How to use CSV module with the APIs provided?

First, Enable the CSV function with Csv_EnableMainCsv() or Csv_EnableSubCsv().When the abnormal status is detected, a CSV reset occurs, then read the CSV failure cause by Csv_GetCsvFailCause().

Disable the CSV function with Csv_DisableMainCsv() or Csv_DisableSubCsv().

How to use FCS module with the APIs provided?

First, set the CR dividor with Csv_SetFcsCrDiv() and set the expected range of main clock cycle with Csv_SetFcsDetectRange().

Second, enable the FCS interrupt with Csv_EnableFcsInt() or enable FCS reset with Csv_EnableFcsReset().

Then start FCS function with Csv_EnableFcs().

When abnormal frequency is detected, an interrupt occurs when FCS interrupt is enabled, then read the interrupt flag by Csv_GetFcsIntFlag() and clear the interrupt flag by Csv_ClrFcsIntFlag().

When abnormal frequency is detected, a reset issues when FCS reset is enabled.

When abnormal frequency is detected, current main clock cycle can be read by Csv_GetFcsDetectCount().

Disable FCS by Csv_DisableFcs(), disable FCS reset by Csv_DisableFcsReset() and disable FCS interrupt by Csv_DisableFcsInt().

7.8.1 Configuration Structure

A CSV status instance uses the following configuration structure of the type of `stc_csv_status_t`:

Type	Field	Possible Values	Description
boolean_t	bCsvMainClockStatus	TRUE	A main clock failure has been detected.
		FALSE	No main clock failure has been detected.
boolean_t	bCsvSubClockStatus	TRUE	A sub clock failure has been detected.
		FALSE	No sub clock failure has been detected.

7.8.2 CSV API

7.8.2.1 Csv_EnableMainCsv ()

This function enables main CSV function.

Prototype	
`void Csv_EnableMainCsv(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	

7.8.2.2 Csv_DisableMainCsv ()

This function disables main CSV function.

Prototype	
`void Csv_EnableMainCsv(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	

7.8.2.3 Csv_EnableSubCsv ()

This function enables sub CSV function.

Prototype	
`void Csv_EnableSubCsv(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	

7.8.2.4 *Csv_DisableSubCsv ()*

This function disables sub CSV function.

Prototype	
`void Csv_DisableSubCsv(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	

7.8.2.5 *Csv_GetCsvFailCause ()*

This function gets CSV status.

Prototype	
`uint8_t Csv_GetCsvFailCause(stc_csv_status_t* pstcCsvStatus);`	
Parameter Name	**Description**
`[out] pstcCsvStatus`	Pointer to status information struture of CSV.
Return Values	**Description**
`Ok`	Get status done and set value to `pstcCsvStatus`.

7.8.2.6 *Csv_EnableFcs ()*

This function enables FCS function.

Prototype	
`void Csv_EnableFcs(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	

7.8.2.7 *Csv_DisableFcs ()*

This function disables FCS function.

Prototype	
`void Csv_DisableFcs(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	

7.8.2.8 Csv_EnableFcsReset ()

This function enables FCS reset.

Prototype	
`void Csv_EnableFcsReset(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	

7.8.2.9 Csv_DisableFcsReset ()

This function disables FCS reset.

Prototype	
`void Csv_DisableFcsReset(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	

7.8.2.10 Csv_EnableFcsInt ()

This function enables FCS interrupts.

Prototype	
`en_result_t Csv_EnableFcsInt(fn_fcs_int_callback* pfnIntCallback);`	
Parameter Name	**Description**
`[in] pfnIntCallback`	Pointer to interrupt callback function.
Return Values	**Description**
`Ok`	Enable FCS interrupt done.
`ErrorInvalidParameter`	`PfnIntCallback == NULL`

7.8.2.11 Csv_DisableFcsInt ()

This function disables FCS interrupts.

Prototype	
`void Csv_DisableFcsInt(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	

7.8.2.12 Csv_ClrFcsIntFlag ()

This function clears the FCS interrupt cause.

Prototype	
`void Csv_ClrFcsIntFlag(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	

7.8.2.13 Csv_GetFcsIntFlag ()

This function gets Anomalous frequency detection interrupt status.

Prototype	
`uint8_t Csv_GetFcsIntFlag(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
0	No FCS interrupt has been asserted.
1	An FCS interrupt has been asserted.

7.8.2.14 Csv_SetFcsCrDiv ()

This function sets FCS count cycle.

Prototype	
en_result_t Csv_SetFcsCrDiv(en_fcs_cr_div_t enDiv);	
Parameter Name	**Description**
[in] enDiv	Count cycle setting value.
Return Values	**Description**
Ok	Count cycle value set done.
ErrorInvalidParameter	enDiv is invalid.

7.8.2.15 Csv_SetFcsDetectRange ()

This function sets frequency lower detection window.

Prototype	
void Csv_SetFcsDetectRange(uint16_t u16LowerVal, uint16_t u16UpperVal);	
Parameter Name	**Description**
[in] u16LowerVal	Lower value.
[in] u16UpperVal	Limit value.
Return Values	**Description**
-	

7.8.2.16 Csv_GetFcsDetectCount ()

This function gets the counter value of frequency detection using the main clock.

Prototype	
uint16_t Csv_GetFcsDetectCount(void);	
Parameter Name	**Description**
-	
Return Values	**Description**
value	Frequency detection counter value.

7.9 (DAC) Digital Analog Converter

Type Definition	stc_dacn_t
Configuration Types	stc_dac_config_t
Address Operator	DAC*n*

The DAC module provides simple API functions for easy usage. Each of the two DAC channels have own API

functions except `Dac_Init()` and `Dac_DeInit()`.

7.9.1 Configuration Structure

A DAC instance uses the following configuration structure of the type of `stc_dac_config_t`:

Type	Field	Possible Values	Description
boolean_t	bDac12Bit0	TRUE FALSE	DAC ch. 0 in 12-bit mode DAC ch. 0 in 10-bit mode
boolean_t	bDac10RightAlign0	TRUE FALSE	DAC ch. 0 10-bit data aligned to DA[9:0] DAC ch. 0 10-bit data aligned to DA[11:2]
boolean_t	bDac12Bit1	TRUE FALSE	DAC ch. 1 in 12-bit mode DAC ch. 1 in 10-bit mode
boolean_t	bDac10RightAlign1	TRUE FALSE	DAC ch. 1 10-bit data aligned to DA[9:0] DAC ch. 1 10-bit data aligned to DA[11:2]

7.9.2 DAC API

7.9.2.1 Dac_Init()

This function initializes a DAC instance according the given configuration. Note that this function does not enable the DAC operation.

Prototype	
`en_result_t Dac_Init( stc_dacn_t* pstcDac,` `stc_dac_config_t* pstcConfig );`	
Parameter Name	**Description**
[in] pstcDac	Pointer to DAC instance
[in] pstcConfig	Pointer to DAC configuration structure
Return Values	**Description**
Ok	DAC instance successfully initialized
ErrorInvalidParameter	pstcDac == NULL or pstcConfig == NULL

7.9.2.2 Dac_DeInit()

This function de-initializes a DAC instance.

Prototype	
`en_result_t Dac_DeInit( stc_dacn_t* pstcDac );`	
Parameter Name	**Description**
[in] pstcDac	Pointer to DAC instance
Return Values	**Description**
Ok	DAC instance successfully de-initialized
ErrorInvalidParameter	pstcDac == NULL

7.9.2.3 *Dac_SetValue0()*

This function sets DAC0 12-bit value.

Prototype	
`en_result_t Dac_SetValue0( stc_dacn_t* pstcDac, uint16_t u16DacValue );`	
Parameter Name	**Description**
`[in] pstcDac`	Pointer to DAC instance
`[in] u16DacValue`	Value to be set
Return Values	**Description**
`Ok`	Value set

7.9.2.4 *Dac_SetValue1()*

This function sets DAC1 12-bit value.

Prototype	
`en_result_t Dac_SetValue1( stc_dacn_t* pstcDac, uint16_t u16DacValue );`	
Parameter Name	**Description**
`[in] pstcDac`	Pointer to DAC instance
`[in] u16DacValue`	Value to be set
Return Values	**Description**
`Ok`	Value set

7.9.2.5 *Dac_Enable0 ()*

This function enables the DAC channel 0 operation.

Prototype	
`en_result_t Dac_Enable0( stc_dacn_t* pstcDac );`	
Parameter Name	**Description**
`[in] pstcDac`	Pointer to DAC instance
Return Values	**Description**
`Ok`	DAC channel 0 operation enabled

7.9.2.6 *Dac_Enable1 ()*

This function enables the DAC channel 1 operation.

Prototype	
`en_result_t Dac_Enable0( stc_dacn_t* pstcDac );`	
Parameter Name	**Description**
`[in] pstcDac`	Pointer to DAC instance
Return Values	**Description**
`Ok`	DAC channel 1 operation enabled

7.9.2.7 *Dac_Disable0 ()*

This function disables the DAC channel 0 operation.

Prototype	
`en_result_t Dac_Enable0( stc_dacn_t* pstcDac );`	
Parameter Name	Description
`[in] pstcDac`	Pointer to DAC instance
Return Values	Description
`Ok`	DAC channel 0 operation disabled

7.9.2.8 *Dac_Enable1 ()*

This function disables the DAC channel 1 operation.

Prototype	
`en_result_t Dac_Enable0( stc_dacn_t* pstcDac );`	
Parameter Name	Description
`[in] pstcDac`	Pointer to DAC instance
Return Values	Description
`Ok`	DAC channel 1 operation disabled

7.9.3 DAC Example

The PDL example folder contains a DAC usage example:

- dac_sine_wave Outputs sine wave sound on the DAC pins.

7.10 (DMA) Direct Memory Access

Type Definition	-
Configuration Types	`stc_dma_config_t`
Address Operator	-

The DMA is configured by `Dma_Init_Channel()` but not started then. With the function `Dma_Set_Channel()` the enable, pause and/or trigger bits can be set. `Dma_Enable()` enables globally the DMA and `Dma_Disable()` disables DMA globally. `Dma_DeInit_Channel()` clears a channel for a possible new configuration.

Once a DMA channel was setup by `Dma_Init_Channel()` it cannot be re-initialized by this function (with a new configuration) anymore. `OperationInProgress` is returned in this case. `Dma_DeInit_Channel()` has to be called before to unlock the channel for a new configuration.

`Dma_Set_ChannelParam()` and `Dma_DeInit_ChannelParam()` perform the same functionality as `Dma_Set_Channel()` and `Dma_DeInit_Channel()` instead of configuration usage. Here direct arguments are used.

Note:

Set `stc_dma_config_t::u16TransferCount` to “Number of Transfers – 1”!

7.10.1 Configuration Structure

Each DMA channel has the same configuration structure of the type of `stc_dma_config_t`:

Type	Field	Possible Values	Description
boolean_t	bEnable	TRUE FALSE	Enable DMA channel Disable DMA channel
boolean_t	bPause	TRUE FALSE	Pause of a DMA channel Normal function / resume
boolean_t	bSoftwareTrigger	TRUE FALSE	Trigger bit for software transfer No trigger
uint8_t	u8DmaChannel	0…DMA_{MAXCH} - 1	DMA channel for recent configuration
en_dma_… idreq_t	enDmaIdrq	*(see below)*	ID Request number *(see below)*
uint8_t	u8BlockCount	-	Block counter
uint16_t	u16TransferCount	-	Transfer counter
en_dma_… transfer… mode_t	enTransferMode	DmaBlockTransfer DmaBurstTransfer DmaDemandTransfer	Block transfer Burst transfer Transfer by peripheral interrupt
en_dma_… transfer… width_t	enTransferWdith	Dma8Bit Dma16Bit Dma32Bit	Transfer 8-bit width Transfer 16-bit width Transfer 32-bit width
uint32_t	u32SourceAddress	-	Source address for transfer
uint32_t	u32Destination… Address	-	Destination address for transfer
boolean_t	bFixedSource	TRUE FALSE	Source address not incremented Source address incremented
boolean_t	bFixedDestination	TRUE FALSE	Destination address not incremented Destination address incremented
boolean_t	bReloadCount	TRUE FALSE	Count is reloaded at end of DMA Count is not reloaded
boolean_t	bReloadSource	TRUE FALSE	Source address is reloaded at end of DMA Source address is not reloaded
boolean_t	bReload… Destination	TRUE FALSE	Destination address is reloaded at end of DMA Destination address is not reloaded
boolean_t	bErrorInterrupt… Enable	TRUE FALSE	Interrupt at error occurence No interrupt
boolean_t	bCompletion… InterruptEnable	TRUE FALSE	Interrupt at end of DMA transfer No interrupt
boolean_t	bEnableBitMask	TRUE FALSE	Clear EB (bEnable) bit on completion (mandatory for transfer end!) Do not clear EB
func_ptr_t	pfnCallback	-	Function pointer to DMA completion callback function
func_ptr_… arg1_t	pfnErrorCallback	-	Function pointer to DMA error callback function
uint8_t	u8ErrorStopStatus	-	Error code from Stop Status

7.10.2 DMA API

7.10.2.1 *Dma_Init()*

Sets up an DMA channel without starting immediate DMA transfer. `Enable_Dma_Channel()` is used for starting a DMA transfer.

Prototype	
`en_result_t Dma_Init_Channel(volatile stc_dma_config_t* pstcConfig);`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to DMA configuration
Return Values	**Description**
`Ok`	DMA init successful
`ErrorInvalidParameter`	`pstcAdc == NULL` or other invalid configuration
`OperationInProgress`	DMA channel already in use

7.10.2.2 *Dma_Set_Channel()*

This function enables, disables, pauses or triggers a DMA transfer according to the settings in the configuration bits for `EB` (Enable), `PB` (Pause) and `ST` (Software Trigger).

Prototype	
`en_result_t Dma_Set_Channel(volatile stc_dma_config_t* pstcConfig);`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to DMA configuration
Return Values	**Description**
`Ok`	Setting finished

7.10.2.3 *Dma_Enable()*

Enable DMA globally.

Prototype	
`en_result_t Dma_Enable(void);`	
Return Values	**Description**
`Ok`	DMA globally enabled

7.10.2.4 Dma_Disable()

Disable DMA globally.

Prototype	
`en_result_t Dma_Disable(void);`	
Return Values	**Description**
`Ok`	DMA globally disabled

7.10.2.5 Dma_DeInit_Channel ()

This function clears a DMA channel.

Prototype	
`en_result_t Dma_DeInit_Channel(volatile stc_dma_config_t* pstcConfig);`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to DMA configuration
Return Values	**Description**
`Ok`	De-init successful
`ErrorInvalidParameter`	`pstcConfig == NULL` or other invalid configuration

7.10.2.6 Dma_Set_ChannelParam ()

This function enables, disables, pauses or triggers a DMA transfer according to the settings in the parameter list for `EB` (Enable), `PB` (Pause) and `ST` (Software Trigger).

Prototype	

`en_result_t Dma_Set_ChannelParam( uint8_t u8DmaChannel,`
`boolean_t bEnable,`
`boolean_t bPause,`
`boolean_t bSoftwareTrigger );`	

Parameter Name	**Description**
`[in] u8DmaChannel`	DMA channel number
`[in] bEnable`	`TRUE`: Enable channel, `FALSE`: Disable channel
`[in] bPause`	`TRUE`: Channel paused, `FALSE`: Channel working
`[in] bSoftwareTrigger`	`TRUE`: Trigger transfer, `FALSE`: No trigger
Return Values	**Description**
`Ok`	Setting finished

7.10.2.7 *Dma_DeInit_ChannelParam ()*

De-Initializes a DMA channel via channel parameter

Prototype	
`en_result_t Dma_DeInit_ChannelParam(uint8_t u8DmaChannel);`	
Parameter Name	**Description**
`[in] u8DmaChannel`	DMA channel number
Return Values	**Description**
`Ok`	De-init successful

7.10.3 DMA Example

The PDL example folder contains a DMA usage example:

- dma_software   DMA software triggered transfer.

7.11 (DSTC) Descriptor System Data Transfer Controller

Type Definition	-
Configuration Types	`stc_dstc_config_t`
Address Operator	-

The following functions are provided:

`Dstc_ReleaseStandBy()` "wakes up" the DSTC from standby mode.

`Dsct_Init()` initializes the DSTC with the user configuration (Priority Settings, Pointers to (Error) Callback Functions, etc.)

`Dstc_ReleaseStandBy()` is called automatically via initialization.

`Dsct_ChannelInit()` initializes a channel with `DES` area start address.

`Dstc_ReadHwdesp()` returns the `DES` area start address of a dedicated channel.

`Dstc_SetCommand()` sets a command to the `CMD` register.

`Dstc_SwTrigger()` performs a software trigger for a configured transfer.

`Dstc_SwTrqansferStartStatus()` returns the status of a triggered Software transfer.

`Dstc_SetDreqenb()` sets the `DRQENB` register set via the structure `stc_dstc_dreqenb_t`.

`Dstc_ReadDreqenb()` returns the value of the `DRQENB` register to the given structure of the type of `stc_dstc_dreqenb_t`.

`Dstc_SetDreqenbBit()` sets a bit in the 256-Bitfield of `DRQENB` register.

`Dstc_ClearDreqenbBit()` clears a bit in the 256-Bitfield of `DRQENB` register.

`Dstc_ReadHwint()` returns the contents of the `HWINT` register into a strucutre of the type of `stc_dstc_hwint_t`.

`Dstc_SetHwintclr()` sets clear bits in the `HWINTCLR` register via the structure of the type of `stc_dstc_hwintclr_t`.

`Dstc_SetHwintclrbBit()` sets clear bits in the 256-Bitfield of the `HWINTCLR` register.

`Dstc_ReadDqmsk()` reads the contents of the `DQMSK` register into a structure of the type of `stc_dstc_dqmsk_t`.

`Dstc_SetDqmskclr()` sets clear bits in the `DQMSKCLR` register via the structure of the type of `stc_dstc_dqmskclr_t`.

`Dstc_SetDqmskclrbBit()` sets a clear bit in the 256-Bitfield of the `DQMSKCLR` register.

In *dstc.h* there are predefined `DES` structure types for every combination of the descriptors. Note that for `DES1` two different structures are existing for mode0 and mode1. The user should use these structures within an enclosing structure, so that the descriptors are located on consecutive addresses. The address of this structure then should be used for `stc_dstc_config_t::u32Destp`.

Note that any peripheral transfer for DSTC has to be defined *in pdl_user.h*, i.e
`PDL_DSTC_ENABLE_ADC0_PRIO == PDL_ON`.

7.11.1 Configuration Structure

The DSTC configuration structure has the type `stc_dstc_config_t`:

Type	Field	Possible Values	Description
uint32_t	u32Destp	-	Start Address of DES Area (must be aligned to 32 Bit!)
boolean_t	bSwInterrupt… Enable	TRUE FALSE	Software Interrupt enabled Software Interrupt disabled
boolean_t	bErInterrupt… Enable	TRUE FALSE	Error Interrupt enabled Error Interrupt disabled
boolean_t	bReadSkipBuffer… Disable	TRUE FALSE	Read Skip Buffer disabled Read Skip Buffer enabled
boolean_t	bErrorStopEnable	TRUE FALSE	Enables Error Stop Disables Error Stop
en_dstc_… swpr_t	enSwTransfer… Priority	PriorityHighest Priority1_2 Priority1_3 Priority1_7 Priority1_15 Priority1_31 Priority1_63 PriorityLowest	Highest priority Priority 1/2 transfer right Priority 1/3 transfer right Priority 1/7 transfer right Priority 1/15 transfer right Priority 1/31 transfer right Priority 1/63 transfer right Lowest priority
func_ptr_t	pfnNotifySw… Callback	-	Notification SW callback function pointer
func_ptr_… dstc_… args_t	pfnErrorCallback	-	Error status callback pointer

The following configuration callback function pointer depend on enabling a DSTC channel in *user_pdl.h*, e.g. `PDL_DSTC_ENABLE_ADC0_PRIO == PDL_ON`. They all have the type of `func_ptr_t`.

- pfnDstcAdc0PrioCallback
- pfnDstcAdc0ScanCallback

. . .

- pfnDstcAdc2PrioCallback
- pfnDstcAdc2ScanCallback
- pfnDstcBt0Irq0Callback

. . .

- pfnDstcBt15Irq1Callback
- pfnDstcExint0Callback

. . .

- pfnDstcExint31Callback
- pfnDstcMfs0RxCallback
- pfnDstcMfs0TxCallback

. . .

- pfnDstcMfs15RxCallback
- pfnDstcMfs15TxCallback
- pfnDstcMft0Frt0PeakCallback
- pfnDstcMft0Frt0ZeroCallback

. . .

- pfnDstcMft0Frt2PeakCallback
- pfnDstcMft0Frt2ZeroCallback
- pfnDstcMft0Icu0Callback

. . .

- pfnDstcMft0Icu3Callback
- pfnDstcMft0Ocu0Callback

. . .

- pfnDstcMft0Ocu5Callback
- pfnDstcMft0Wfg10Callback
- pfnDstcMft0Wfg32Callback
- pfnDstcMft0Wfg54Callback

- pfnDstcMft1… *(see MFT0 above)*
- pfnDstcMft2… *(see MFT0 above)*
- pfnDstcPpg0Callback
- pfnDstcPpg2Callback

  . . .
- pfnDstcPpg18Callback
- pfnDstcPpg20Callback
- pfnDstcQprc0CountInversionCallback
- pfnDstcQprc0OutOfRangeCallback
- pfnDstcQprc0PcMatchCallback
- pfnDstcQprc0PcMatchRcMatchCallback
- pfnDstcQprc0PcRcMatchCallback
- pfnDstcQprc0UflOflZCallback
- pfnDstcQprc1… *(see QPRC0 above)*
- pfnDstcQprc2… *(see QPRC0 above)*
- pfnDstcQprc3… *(see QPRC0 above)*
- pfnDstcUsb0Ep1Callback

  . . .
- pfnDstcUsb0Ep5Callback
- pfnDstcUsb1… *(see USB0 above)*
- pfnDstcWcCallback

7.11.2 DSTC DES Structures

The PDL provides for all combinations of the DES descriptors own structures. If a composition of these descriptors is used the user shall declare this as a big comprehensive structure to fulfill the DSTC requirement of continuous increasing addresses with no gaps or incorrect address order.

7.11.2.1 DES0

The DES0 structure of type of stc_dstc_des0_t has the following format:

Structure element	Bit Width
uint32_t DV	2
uint32_t ST	2
uint32_t MODE	1
uint32_t ORL	3
uint32_t TW	2
uint32_t SAC	3
uint32_t DAC	3
uint32_t CHRS	6
uint32_t DMSET	1
uint32_t CHLK	1
uint32_t ACK	2
uint32_t *RESERVED*	2
uint32_t PCHK	4

7.11.2.2 DES1 – Mode 0

The DES1 in mode 0 has the following structure of type of stc_dstc_des1_mode0_t:

Structure element	Bit Width
uint32_t IIN	16
uint32_t ORM	16

7.11.2.3 DES1 – Mode 1

The DES1 in mode 1 has the following structure of type of stc_dstc_des1_mode1_t:

Structure element	Bit Width
uint32_t IIN	8
uint32_t IRM	8
uint32_t ORM	16

7.11.2.4 DES0 – DES3 Combination

To fulfill the requirement of continuous increasing DES addresses the PDL provides the combination of DES0-DES1-DES2-DES3. The corresponding structure of type of stc_dstc_des0123_t is:

Structure element	Bit Width
stc_dstc_des0_t DES0	32
Union: stc_dstc_des1_mode0_t DES1_mode0 stc_dstc_des1_mode0_t DES1_mode1	32
uint32_t DES2	32
uint32_t DES3	32

7.11.2.5 DES0 – DES4 Combination

To fulfill the requirement of continuous increasing `DES` addresses the PDL provides the combination of `DES0-DES1-DES2-DES3-DES4`. The corresponding structure of type of `stc_dstc_des01234_t` is:

Structure element	Bit Width
stc_dstc_des0_t DES0	32
Union: stc_dstc_des1_mode0_t DES1_mode0 stc_dstc_des1_mode0_t DES1_mode1	32
uint32_t DES2	32
uint32_t DES3	32
Union: stc_dstc_des1_mode0_t DES4_mode0 stc_dstc_des1_mode0_t DES4_mode1	32

7.11.2.6 DES0 – DES5 Combination

To fulfill the requirement of continuous increasing `DES` addresses the PDL provides the combination of `DES0-DES1-DES2-DES3-DES4-DES5`. The corresponding structure of type of `stc_dstc_des012345_t` is:

Structure element	Bit Width
stc_dstc_des0_t DES0	32
Union: stc_dstc_des1_mode0_t DES1_mode0 stc_dstc_des1_mode0_t DES1_mode1	32
uint32_t DES2	32
uint32_t DES3	32
Union: stc_dstc_des1_mode0_t DES4_mode0 stc_dstc_des1_mode0_t DES4_mode1	32
uint32_t DES5	32

7.11.2.7 DES0 – DES6 Combination

To fulfill the requirement of continuous increasing `DES` addresses the PDL provides the combination of `DES0-DES1-DES2-DES3-DES4-DES5-DES6`. The corresponding structure of type of `stc_dstc_des0123456_t` is:

Structure element	Bit Width
`stc_dstc_des0_t DES0`	32

Union:
`stc_dstc_des1_mode0_t DES1_mode0`
`stc_dstc_des1_mode0_t DES1_mode1`
	32

`uint32_t DES2`	32
`uint32_t DES3`	32

Union:
`stc_dstc_des1_mode0_t DES4_mode0`
`stc_dstc_des1_mode0_t DES4_mode1`
	32

`uint32_t DES5`	32
`uint32_t DES6`	32

7.11.2.8 DES0– DES3, DES5 Combination

To fulfill the requirement of continuous increasing `DES` addresses the PDL provides the combination of `DES0-DES1-DES2-DES3-DES5`. The corresponding structure of type of `stc_dstc_des01235_t` is:

Structure element	Bit Width
`stc_dstc_des0_t DES0`	32

Union:
`stc_dstc_des1_mode0_t DES1_mode0`
`stc_dstc_des1_mode0_t DES1_mode1`
	32

`uint32_t DES2`	32
`uint32_t DES3`	32
`uint32_t DES5`	32

7.11.2.9 DES0 – DES3, DES6 Combination

To fulfill the requirement of continuous increasing `DES` addresses the PDL provides the combination of `DES0-DES1-DES2-DES3-DES5`. The corresponding structure of type of `stc_dstc_des01236_t` is:

Structure element	Bit Width
`stc_dstc_des0_t DES0`	32

Union:
`stc_dstc_des1_mode0_t DES1_mode0`
`stc_dstc_des1_mode0_t DES1_mode1`
	32

`uint32_t DES2`	32
`uint32_t DES3`	32
`uint32_t DES6`	32

7.11.2.10 DES0 – DES4, DES6 Combination

To fulfill the requirement of continuous increasing DES addresses the PDL provides the combination of DES0-DES1-DES2-DES3-DES5. The corresponding structure of type of stc_dstc_des012346_t is:

Structure element	Bit Width
stc_dstc_des0_t DES0	32
Union: stc_dstc_des1_mode0_t DES1_mode0 stc_dstc_des1_mode0_t DES1_mode1	32
uint32_t DES2	32
uint32_t DES3	32
Union: stc_dstc_des1_mode0_t DES4_mode0 stc_dstc_des1_mode0_t DES4_mode1	32
uint32_t DES6	32

7.11.2.11 DES0 – DES3, DES5 – DES6 Combination

To fulfill the requirement of continuous increasing DES addresses the PDL provides the combination of DES0-DES1-DES2-DES3-DES5. The corresponding structure of type of stc_dstc_des012356_t is:

Structure element	Bit Width
stc_dstc_des0_t DES0	32
Union: stc_dstc_des1_mode0_t DES1_mode0 stc_dstc_des1_mode0_t DES1_mode1	32
uint32_t DES2	32
uint32_t DES3	32
uint32_t DES5	32
uint32_t DES6	32

7.11.3 DSTC API

7.11.3.1 Dstc_ReleaseStandBy()

This function releases the DSTC from standby mode to enable the operation.

Prototype	
en_result_t Dstc_ReleaseStandBy(void);	
Return Values	**Description**
Ok	DSTC released from standby mode
ErrorNotReady	DSTC standby release failed

7.11.3.2 Dstc_Init()

This function initializes the DSTC according the configuration.

Prototype	
`en_result_t Dstc_Init( stc_dstc_config_t* pstcConfig );`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to DSTC configuration
Return Values	**Description**
`Ok`	DSTC initialized
`ErrorNotReady`	DSTC standby initialization failed
`ErrorInvalidParameter`	`pstcConfig == NULL` or other configuration wrong
`ErrorAddressAlignment`	`DES` Base Address not aligned to 32 Bit

7.11.3.3 Dstc_DeInit()

This function de-initializes the DSTC.

Prototype	
`en_result_t Dstc_Init( stc_dstc_config_t* pstcConfig );`	
Return Values	**Description**
`Ok`	DSTC de-initialized and in standby mode

7.11.3.4 Dstc_SetHwdesp()

This function initializes a DSTC channel.

Prototype	
`en_result_t Dstc_Init( stc_dstc_config_t* pstcConfig );`	
Parameter Name	**Description**
`[in] u8Channel`	DSTC channel to be initialized
`[in] u16HwDesp`	Address offset to channel configuration
Return Values	**Description**
`Ok`	DSTC channel initialized
`ErrorInvalidParameter`	`DES` Address > 16K Bytes
`ErrorAddressAlignment`	`DES` Base Address not aligned to 32 Bit

7.11.3.5 Dstc_SetHwdesp()

This function reads out the HWDESP address offset of a channel.

Prototype	
uint16_t Dstc_ReadHwdesp(uint8_t u8Channel);	
Parameter Name	**Description**
[in] u8Channel	DSTC channel of HWDESP
Return Values	**Description**
uint16_t	Current HWDESP DES address offset

7.11.3.6 Dstc_SetCommand()

This function sets a command to the CMD register.

Prototype	
en_result_t Dstc_SetCommand(en_dstc_cmd_t enCommand);	
Parameter Name	**Description**
[in] enCommand	DSTC command *(see command list below)*
Return Values	**Description**
Ok	Command set
ErrorInvalidParameter	Wrong command enumerator used

List of commands (en_dstc_cmd_t):

Command	Description
CmdStandyRelease	Instructs DSTC to return from standby state into normal state
CmdStandyTransition	Instructs DSTC to standby state
CmdSwclr	Clears SWTR:SWST to '0'; negates SWINT interrupt signal
CmdErclr	Clears MONERS:EST, MONERS:DER, and MONERS:ESTOP
CmdRbclr	Clears DESP to which DSTP refers in previous transfer; Sets next DES; Clears HWDESP[n]
CmdMkclr	Clears all DQMSK[n]

7.11.3.7 *Dstc_SwTrigger()*

This function sets `SWDESP` offset and trigger SW transfer.

Prototype	
`en_result_t Dstc_SwTrigger( uint16_t u16SwDesPointer );`	
Parameter Name	**Description**
`[in] u8Swdesp`	Address offset to `DES`
Return Values	**Description**
`Ok`	`SWDESP` set
`ErrorInvalidParameter`	Offset > 16K Bytes
`ErrorAddressAlignment`	Offset not 32-bit aligned

7.11.3.8 *Dstc_SwTrqansferStartStatus()*

This function checks the Software Trigger start state.

Prototype	
`en_result_t Dstc_SwTrqansferStartStatus( void );`	
Return Values	**Description**
`Ok`	Transfer finished
`OperationInProgress`	Transfer pending

7.11.3.9 *Dstc_SetDreqenb()*

This function sets the `DREQENB` register with a structure of type of `stc_dstc_dreqenb_t`.

Prototype	
`en_result_t Dstc_SetDreqenb( stc_dstc_dreqenb_t* pstcDreqenb );`	
Parameter Name	**Description**
`[in] pstcDreqenb`	Pointer to `DREQENB` structure
Return Values	**Description**
`Ok`	`DREQENB` set

Structure of type of `stc_dstc_dreqenb_t`:

Structure element	Bitposition of DREQENB
uint32_t u32Dreqenb0	[31:0]
uint32_t u32Dreqenb1	[63:32]
uint32_t u32Dreqenb2	[95:64]
uint32_t u32Dreqenb3	[127:96]
uint32_t u32Dreqenb4	[159:128]
uint32_t u32Dreqenb5	[191:160]
uint32_t u32Dreqenb6	[223:192]
uint32_t u32Dreqenb7	[255:334]

7.11.3.10 Dstc_ReadDreqenb()

This function reads-out the `DREQENB` register into a structure of type of `stc_dstc_dreqenb_t`. The structure is the same as above (7.11.3.9).

Prototype	
`en_result_t Dstc_ReadDreqenb( stc_dstc_dreqenb_t* pstcDreqenb );`	
Parameter Name	**Description**
[out] pstcDreqenb	Pointer to DREQENB structure
Return Values	**Description**
Ok	DREQENB read-out

7.11.3.11 Dstc_SetDreqenbBit()

This function sets a single bit of `DREQENB` register.

Prototype	
`en_result_t Dstc_SetDreqenbBit( uint8_t u8BitPos );`	
Parameter Name	**Description**
[in] u8BitPos	Bit position of DREQENB register [0…255]
Return Values	**Description**
Ok	DREQENB bit set

7.11.3.12 Dstc_ClearDreqenbBit()

This function clears a single bit of DREQENB register.

Prototype	
en_result_t Dstc_ClearDreqenbBit(uint8_t u8BitPos);	
Parameter Name	**Description**
[in] u8BitPos	Bit position of DREQENB register [0…255]
Return Values	**Description**
Ok	DREQENB bit cleared

7.11.3.13 Dstc_ReadHwint()

This function sets the HWINT register with a structure of type of stc_dstc_hwint_t.

Prototype	
en_result_t Dstc_SetDreqenb(stc_dstc_dreqenb_t* pstcDreqenb);	
Parameter Name	**Description**
[in] pstcHwint	Pointer to HWINT structure
Return Values	**Description**
Ok	HWINT set

Structure of type of stc_dstc_hwint_t:

Structure element	Bitposition of HWINT
uint32_t u32Hwint0	[31:0]
uint32_t u32Hwint1	[63:32]
uint32_t u32Hwint2	[95:64]
uint32_t u32Hwint3	[127:96]
uint32_t u32Hwint4	[159:128]
uint32_t u32Hwint5	[191:160]
uint32_t u32Hwint6	[223:192]
uint32_t u32Hwint7	[255:334]

7.11.3.14 *Dstc_SetHwintclr()*

This function sets the `HWINTCLR` register with a structure of type of `stc_dstc_hwintclr_t`. The corresponding `HWINT` bit(s) are cleared by this function.

Prototype	
`en_result_t Dstc_SetHwintclr( stc_dstc_hwintclr_t* pstcHwintclr );`	
Parameter Name	**Description**
`[in] pstcHwintclr`	Pointer to `HWINTCLR` structure
Return Values	**Description**
`Ok`	`HWINTCLR` set

Structure of type of `stc_dstc_hwintclr_t`:

Structure element	Bitposition of `HWINT`
`uint32_t u32Hwintclr1`	`[31:0]`
`uint32_t u32Hwintclr2`	`[63:32]`
`uint32_t u32Hwintclr3`	`[95:64]`
`uint32_t u32Hwintclr4`	`[127:96]`
`uint32_t u32Hwintclr5`	`[159:128]`
`uint32_t u32Hwintclr6`	`[191:160]`
`uint32_t u32Hwintclr7`	`[223:192]`
`uint32_t u32Hwintclr0`	`[255:334]`

7.11.3.15 *Dstc_ReadHwintBit ()*

This function reads a single bit of `HWINT` register.

Prototype	
`boolean_t Dstc_ReadHwintBit( uint8_t u8BitPos );`	
Parameter Name	**Description**
`[in] u8BitPos`	Bit position of `HWINT` register `[0…255]`
Return Values	**Description**
`TRUE`	`HWINT` bit set
`FALSE`	`HWINT` bit not set

7.11.3.16 Dstc_ReadHwintBit ()

This function sets a single clear bit of HWINTCLR register.

Prototype	
en_result_t Dstc_SetHwintclrBit(uint8_t u8BitPos);	
Parameter Name	**Description**
[in] u8BitPos	Bit position of HWINTCLR register [0…255]
Return Values	**Description**
Ok	HWINTCLR clear bit set

7.11.3.17 Dstc_ReadDqmsk()

This function reads-out the DQMSK register into a structure of type of stc_dstc_dqmsk_t.

Prototype	
en_result_t Dstc_ReadDqmsk(stc_dstc_dqmsk_t* pstcDqmsk);	
Parameter Name	**Description**
[out] pstcDqmsk	Pointer to DQMSK structure
Return Values	**Description**
Ok	DQMSK read-out

Structure of type of stc_dstc_dqmsk_t:

Structure element	Bitposition of HWINT
uint32_t u32Dqmsk0	[31:0]
uint32_t u32Dqmsk1	[63:32]
uint32_t u32Dqmsk2	[95:64]
uint32_t u32Dqmsk3	[127:96]
uint32_t u32Dqmsk4	[159:128]
uint32_t u32Dqmsk5	[191:160]
uint32_t u32Dqmsk6	[223:192]
uint32_t u32Dqmsk7	[255:334]

7.11.3.18 *Dstc_SetDqmskclr()*

This function sets the `DQMSKCLR` register with a structure of type of `stc_dstc_dqmskclr_t`. The corresponding `DQMSK` bit(s) are cleared by this function.

Prototype	
`en_result_t Dstc_SetDqmskclr( stc_dstc_dqmskclr_t* pstcDqmskclr );`	
Parameter Name	**Description**
`[in] pstcDqmskclr`	Pointer to `DQMSKCLR` structure
Return Values	**Description**
`Ok`	`DQMSKCLR` bit(s) set

Structure of type of `stc_dstc_dqmskclr_t`:

Structure element	Bitposition of `HWINT`
`uint32_t u32Dqmskclr0`	`[31:0]`
`uint32_t u32Dqmskclr1`	`[63:32]`
`uint32_t u32Dqmskclr2`	`[95:64]`
`uint32_t u32Dqmskclr3`	`[127:96]`
`uint32_t u32Dqmskclr4`	`[159:128]`
`uint32_t u32Dqmskclr5`	`[191:160]`
`uint32_t u32Dqmskclr6`	`[223:192]`
`uint32_t u32Dqmskclr7`	`[255:334]`

7.11.3.19 *Dstc_SetDqmskclrBit()*

This function sets a single bit of `DQMSKCLR` register. The corresponding `DQMSK` bit is cleared by this function.

Prototype	
`en_result_t Dstc_SetDqmskclrBit( uint8_t u8BitPos );`	
Parameter Name	**Description**
`[in] u8BitPos`	Bit position of `DQMSKCLR` register `[0…255]`
Return Values	**Description**
`Ok`	`DQMSKCLR` bit cleared

7.11.3.20 *Hardware Transfer Callback Functions*

For the hardware transfer callback functions refer to the configuration function pointer list in paragraph 7.11.1. These functions are only available if the corresponding DSTC peripheral channel is enabled in *pdl_user.h*, such as e.g. `PDL_DSTC_ENABLE_ADC0_PRIO == PDL_ON`.

7.11.3.21 *ErrorCallback()*

The error callback function has the following format:

Prototype	
`void DstcErrorCallbackName( en_dstc_est_error_t enEstError, uint16_t u16ErrorChannel, uint16_t u16ErrorDesPointer, boolean_t bSoftwareError, boolean_t bDoubleError, boolean_t bErrorStop);`	
Parameter	**Description**
enEstError	Error code *(see below)*
u16ErrorChannel	Channel which reports the error
u16ErrorDesPointer	Address offset to the erroneous DES
bSoftwareError	TRUE: Software error occurred, FALSE: Hardware error (MONRES:EHS)
bDoubleError	TRUE: Double error occurred, FALSE: No double error (MONRES:DER)
bErrorStop	TRUE: Transfer stopped by error, FALSE: No transfer stop (MONRES:ESTOP)

The enumerators of `en_dstc_est_error_t` are:

Enumerator	Description
NoError	No error (will not occur)
SourceAccessError	Source address error occurred
DestinationAccessError	Destination address error occurred
ForcedTransferStop	Transfer has been stopped compulsorily
DesAccessError	DES access error
DesOpenError	DES open error
UnknownError	Undefined state, should never happen

7.11.4 DSTC Examples

The PDL example folder contains two DSTC usage example:

- dstc_hw_transfer DSTC transfer triggered by Base Timer 0.
- dstc_sw_transfer DSTC transfer triggered by software.

7.12 (DT) Dual Timer

Type Definition	-
Configuration Type	stc_dt_channel_config_t
Address Operator	-

The DT APIs provide a set of functions for accessing the DT block.

`Dt_Init()` initializes one of the Dual Timer (DT) block channels.

`Dt_EnableCount()` enables one of the channels of the DT block.

`Dt_EnableInt()` enables the interrupt. This API sets the callback function. It is recommended that This API is called before `Dt_EnableCount()` enables the DT block.

`Dt_WriteLoadVal()` sets the value given in the parameter `Dt_WriteLoadVal()::u32LoadVal` to the DT block counter in any of the three operation modes.

`Dt_WriteBgLoadVal()` sets the background reload value to the DT block. The reload value is loaded to the DT block counter after the DT block counter reaches to the next 0.

`Dt_ReadCurCntVal()` returns the current DT block counter value.

[Note] Before deinitialization of the DT block by `Dt_DeInit()`, it is recommended to disable all channels with `Dt_DisableCount()` and `Dt_DisableInt()`, to avoid unneccesary interruptions.

7.12.1 DT Configuration Structure

The argument of `Dt_Init()` is a pointer to a structure of the DT configuration. The type of the structure is `stc_dt_channel_config_t`. The members of `stc_dt_channel_config_t` are:

Type	Field	Possible Values	Description

`uint8_t`	`u8Mode`	`DtFreeRun`
		`DtPeriodic`
		`DtOneShot`
			Free-run mode
			Periodic mode
			One-shot mode

`uint8_t`	`u8PrescalerDiv`	`DtPrescalerDiv1`
		`DtPrescalerDiv16`
		`DtPrescalerDiv256`
			Prescaler divider 1
			Prescaler divider 16
			Prescaler divider 256

`uint8_t`	`u8CounterSize`	`DtCounterSize16` `DtCounterSize32`	16 Bit counter size 32 Bit counter size

7.12.2 API Reference

7.12.2.1 Dt_Init()

Initializes the specified channel of the DT block.

Prototype	
`en_result_t Dt_Init(stc_dt_channel_config_t* pstcConfig, uint8_t u8Ch)`	
Parameter Name	**Description**
[in] `pstcConfig`	A pointer to a structure of the DT configuration
[in] `u8Ch`	The instance number
Return Values	**Description**
`Ok`	Initialization ended with no error
`ErrorInvalidParameter`	`pstcConfig == NULL` `u8Ch >= DtMaxChannels`(*)

(*) DtMaxChannels is now defined as "2" in *dt.h*.

7.12.2.2 Dt_DeInit()

Deinitializes the specified channel of the DT block.

Prototype	
en_result_t Dt_DeInit(uint8_t u8Ch)	
Parameter Name	**Description**
[in] u8Ch	The instance number
Return Values	**Description**
Ok	Clearing register values and internal data ended with no error
ErrorInvalidParameter	u8Ch >= DtMaxChannels

7.12.2.3 Dt_EnableCount()

Enables the DT block counter.

Prototype	
en_result_t Dt_EnableCount(uint8_t u8Ch)	
Parameter Name	**Description**
[in] u8Ch	The instance number
Return Values	**Description**
Ok	Starting the DT block ended with no error
ErrorInvalidParameter	u8Ch >= DtMaxChannels

7.12.2.4 Dt_DisableCount()

Disables the DT counter.

Prototype	
en_result_t Dt_DisableCount(uint8_t u8Ch)	
Parameter Name	**Description**
[in] u8Ch	The instance number
Return Values	**Description**
Ok	Stopping the DT block ended with no error
ErrorInvalidParameter	u8Ch >= DtMaxChannels

7.12.2.5 Dt_EnableInt()

Enables interrupts and registers the callback function.

Prototype	
`en_result_t Dt_EnableInt(dt_cb_func_ptr_t pfnIntCallback, uint8_t u8Ch)`	
Parameter Name	**Description**
[in] `pfnIntCallback`	A pointer to a callback function This parameter can be set to NULL
[in] `u8Ch`	The instance number
Return Values	**Description**
`Ok`	Interrupts were enabled and the callback function was registered
`ErrorInvalidParameter`	`u8Ch >= DtMaxChannels`

7.12.2.6 Dt_DisableInt()

Disables interrupts and clears the callback function.

Prototype	
`en_result_t Dt_DisableInt(uint8_t u8Ch)`	
Parameter Name	**Description**
[in] `u8Ch`	The instance number
Return Values	**Description**
`Ok`	Interrupts were disabled and the callback function is not registered
`ErrorInvalidParameter`	`u8Ch >= DtMaxChannels`

7.12.2.7 Dt_GetIntFlag()

Returns interrupts status.

Prototype	
`boolean_t Dt_GetIntFlag(uint8_t u8Ch)`	
Parameter Name	**Description**
[in] `u8Ch`	The instance number
Return Values	**Description**
`boolean_t`	Interrupts status `TRUE`: Interrupts occurred `FALSE`: Interrupts did not occur

7.12.2.8 Dt_GetMaskIntFlag()

Returns masked interrupts.

Prototype	
boolean_t Dt_GetIntFlag(uint8_t u8Ch)	
Parameter Name	**Description**
[in] u8Ch	The instance number
Return Values	**Description**
boolean_t	The logical AND value of interrupt flags and the timer Interrupt enable bit IntEnable of the register TimerXControl TRUE: Interrupt occurred and interrupt was enabled FALSE: Interrupt did not occur or interrupts was disabled

7.12.2.9 Dt_ClrIntFlag()

Clears the interrupt status.

Prototype	
en_result_t Dt_ClrIntFlag(uint8_t u8Ch)	
Parameter Name	**Description**
[in] u8Ch	The instance number
Return Values	**Description**
Ok	Clearig interrupt sources with no error
ErrorInvalidParameter	u8Ch >= DtMaxChannels

7.12.2.10 Dt_WriteLoadVal()

Writes the load value to the load register.

Prototype	
en_result_t Dt_WriteLoadVal(uint32_t u32LoadVal, uint8_t u8Ch)	
Parameter Name	**Description**
[in] u32LoadVal	The load value to be set to the register TimerXLoad
[in] u8Ch	The instance number
Return Values	**Description**
Ok	Loading value to TimerXLoad ended with no error
ErrorInvalidParameter	u8Ch >= DtMaxChannels

7.12.2.11 *Dt_WriteBgLoadVal()*

Writes the load value to the back-ground load register.

Prototype	
`en_result_t Dt_WriteBgLoadVal(uint32_t u32BgLoadVal, uint8_t u8Ch)`	
Parameter Name	**Description**
[in] `u32BgLoadVal`	The load value to set to the register `TimerXBgLoad`
[in] `u8Ch`	The instance number
Return Values	**Description**
`Ok`	Loading value to `TimerXBgLoad` ended with no error
`ErrorInvalidParameter`	`u8Ch >= DtMaxChannels`

7.12.2.12 *Dt_ReadCurCntVal()*

Reads the value from the value register.

Prototype	
`uint32_t Dt_ReadCurCntVal(uint8_t u8Ch)`	
Parameter Name	**Description**
[in] `u8Ch`	The instance number
Return Values	**Description**
`uint32_t`	The value of `TimerXValue`

7.12.2.13 *Callback Function*

`Dt_EnableInt()` registers the callback function. The callback function is called in the interrupt handler when the DT block generates interrupts.

Prototype
`void (*dt_cb_func_prt_t)(void)`

7.12.3 Example Code

The example software is in *\example\dt*.

Folder	Summary
\example\dt\dt_unuse_int	The DT block example without interrupt (by polling)
\example\dt\dt_use_int	The DT block example with interrupt

7.12.3.1 Configuraton

The following example configuration may be used for the example code.

```
// Configuration for channel 0
static const stc_dt_channel_config_t stcDtChannelConfig0 = {
    DtPeriodic,              // Periodic mode
    DtPrescalerDiv256,       // Prescaler dividor f/256
    DtCounterSize32          // 32bits counter size
};

// Configuration for channel 1
static const stc_dt_channel_config_t stcDtChannelConfig1 = {
    DtOneShot,               // One-shot mode
    DtPrescalerDiv256,       // Prescaler dividor f/256
    DtCounterSize32          // 32bits counter size
};
```

7.12.3.2 DT Without Interrupt

```
function
{
  ...
  // Initialize dual timer channel 0
  if (Ok != Dt_Init((stc_dt_channel_config_t*)&stcDtChannelConfig0,
DtChannel0))
  {
    // some code here ...
    while(1);
  }

  // Initialize dual timer channel 1
  if (Ok != Dt_Init((stc_dt_channel_config_t*)&stcDtChannelConfig1,
DtChannel1))
  {
    // some code here ...
    while(1);
  }
  ...
  // Write load value for channel 0 (0.5sec interval)
  Dt_WriteLoadVal(156250, DtChannel0);
  // Write background load value for channel 0 (0.5sec -> 1sec)
  Dt_WriteBgLoadVal(312500, DtChannel0);
  // Start count for channel 0
  Dt_EnableCount(DtChannel0);

  // Write load value for channel 1 (1sec until overflow)
  Dt_WriteLoadVal(312500, DtChannel1);
  // Start count for channel 1
  Dt_EnableCount(DtChannel1);

  while(1)
  {
    // Check interrupt for channel 0
    if (TRUE == Dt_GetIntFlag(DtChannel0))
    {
      Dt_ClrIntFlag(DtChannel0);    // Clear Irq
      // some code here ...
    }
    // Check interrupt for channel 1
    if (TRUE == Dt_GetIntFlag(DtChannel1))
    {
      Dt_ClrIntFlag(DtChannel1);    // Clear Irq
      // some code here ...
    }
  }
```

7.12.3.3 DT with Interrupt

```
static void Dt0Callback(void)
{
  Dt_ClrIntFlag(DtChannel0);     // Clear Irq
  // some code here ...
}

static void Dt1Callback(void)
{
  Dt_ClrIntFlag(DtChannel1);     // Clear Irq
  // some code here ...
}
...
function
{
  ...
  // Initialize dual timer channel 0
  if (Ok != Dt_Init((stc_dt_channel_config_t*)&stcDtChannelConfig0,
DtChannel0))
  {
    // some code here ...
    while(1);
  }

  // Initialize dual timer channel 1
  if (Ok != Dt_Init((stc_dt_channel_config_t*)&stcDtChannelConfig1,
DtChannel1))
  {
    // some code here ...
    while(1);
  }
  ...
  // Write load value for channel 0 (0.5sec interval)
  Dt_WriteLoadVal(156250, DtChannel0);
  // Write background load value for channel 0 (0.5sec -> 1sec)
  Dt_WriteBgLoadVal(312500, DtChannel0);
  // Enable interrupt for channel 0
  Dt_EnableInt(Dt0Callback, DtChannel0);
  // Start count for channel 0
  Dt_EnableCount(DtChannel0);

  // Write load value for channel 1 (1sec until overflow)
  Dt_WriteLoadVal(312500, DtChannel1);
  // Enable interrupt for channel 1
  Dt_EnableInt(Dt1Callback, DtChannel1);
  // Start count for channel 1
  Dt_EnableCount(DtChannel1);

  while(1)
  {
    // some code here ...
  }
}
```

7.13 (EXINT) External Interrupts / NMI

Type Definition	-
Configuration Types **External Interrupts** **NMI**	`stc_exint_config_t` `stc_exint_nmi_config_t`
Address Operator	-

This driver module provides configuration and API functions for using external interrupts and the non-maskable interrupt (NMI). The channel corresponding interrupts do not need to be activated in *pdl_user.h*. It is only necessary to enable a channel by e.g. `PDL_PERIPHERAL_ENABLE_EXINT0 == PDL_ON`.

7.13.1 Configuration Structure (External Interrupts)

The EXINT configuration structure has the type `stc_exint_config_t`:

Type	Field	Possible Values	Description
`boolean_t`	`abEnable[32u]`	`TRUE` `FALSE`	Channel *n* enabled Channel *n* disabled

`en_exint_level_t`	`aenLevel[32u]`	`ExIntLowLevel`
		`ExIntHighLevel`
		`ExIntRisingEdge`
		`ExIntFallingEdge`
			Channel *n* low level
			Channel *n* high level
			Channel *n* rising edge
			Channel *n* falling edge

`func_ptr_t`	`apfnExint…` `Callback[32u]`	-	Callback function pointers

7.13.2 Configuration Structure (NMI)

The NMI configuration structure has the type `stc_nmi_config_t`:

Type	Field	Possible Values	Description
`boolean_t`	`bTouchNVIC`	`TRUE` `FALSE`	NVIC is initialized by NMI NVIC is initialized by other peripheral
`func_ptr_t`	`pfnNMICallback`	-	Callback function pointer

7.13.3 EXINT API

The following API functions are used for enabling, disabling, etc. the external interrupts and the NMI.

7.13.3.1 Exint_Init()

This function initializes the external interrupt channel specified in the configuration structure for their levels and callback functions. The NVIC is initialized too.

Prototype	
`en_result_t Exint_Init( stc_exint_config_t* pstcConfig );`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to configuration structure
Return Values	**Description**
`Ok`	EXINT channels configured successfully
`ErrorInvalidParameter`	`pstcConfig == NULL` or Illegal parameter

7.13.3.2 *Exint_DeInit()*

This function disables the external interrupts and the NVIC globally.

Prototype	
`en_result_t Exint_DeInit( void );`	
Return Values	**Description**
`Ok`	EXINT channels disabled successfully

7.13.3.3 *Exint_EnableChannel ()*

This function enables a single EXINT channel.

Prototype	
`en_result_t Exint_EnableChannel( uint8_t u8Channel );`	
Parameter Name	**Description**
`[in] u8Channel`	EXINT channel number
Return Values	**Description**
`Ok`	EXINT channel enabled
`ErrorInvalidParameter`	Invalid channel number

7.13.3.4 *Exint_DisableChannel ()*

This function disables a single EXINT channel.

Prototype	
`en_result_t Exint_DisableChannel( uint8_t u8Channel );`	
Parameter Name	**Description**
`[in] u8Channel`	EXINT channel number
Return Values	**Description**
`Ok`	EXINT channel disabled
`ErrorInvalidParameter`	Invalid channel number

7.13.3.5 *Exint_Nmi_Init ()*

This function initializes and enables the NMI according the user configuration.

Prototype	
`en_result_t Exint_Nmi_Init( stc_exint_nmi_config_t* pstcConfig );`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to configuration structure
Return Values	**Description**
`Ok`	NMI initialized
`ErrorInvalidParameter`	`pstcConfig == NULL`

7.13.3.6 *Exint_Nmi_DeInit ()*

This function initializes and enables the NMI according the user configuration. The configuration is needed to tell this function whether to de-initialize the NVIC or not.

Prototype	
`en_result_t Exint_Nmi_DeInit( stc_exint_nmi_config_t* pstcConfig );`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to configuration structure
Return Values	**Description**
`Ok`	NMI de-initialized
`ErrorInvalidParameter`	`pstcConfig == NULL`

7.13.3.7 *Callback functions*

The callback functions are collected in an array according the EXINT channel number. They have no arguments.

Prototype
`void` *ExintnCallbackName* `( void );`

7.13.4 EXINT Examples

The PDL example folder contains an EXINT usage example:

- exint_simple External interrupt by port trigger.

7.14 (EXTIF) External Bus Interface

Type Definition	-
Configuration Type	stc_extif_area_config_t
Address Operator	-

This driver provides API functions and configuration for each possible external bus interface memory area.

7.14.1 Configuration Structure

The EXTIF configuration structure has the type `stc_extif_config_t`:

Type	Field	Possible Values	Description
en_extif_width_t	enWidth	Extif8Bit Extif16Bit	Data bus 8 bit wide Data bus 16 bit wide
boolean_t	bReadByteMask	TRUE FALSE	Read byte mask enabled Read byte mask disabled
boolean_t	bWriteEnable… Off	TRUE FALSE	Write enable disabled Write enable possible
boolean_t	bNandFlash	TRUE FALSE	NAND Flash mode activated NAND Flash mode disabled
boolean_t	bPageAccess	TRUE FALSE	NOR Flash Page access enabled NOR Flash Page access disabled
boolean_t	bRdyOn	TRUE FALSE	RDY mode enabled RDY mode disabled
boolean_t	bStopDataOut… AtFirstIdle	TRUE FALSE	Stop to write data output at first idle cycle Extends to write data output to the last idle cycle
boolean_t	bMultiplexMode	TRUE FALSE	Multiplex mode enabled Non-Multiplex mode enabled
boolean_t	bAleInvert	TRUE FALSE	ALE signal inverted (negative) ALE signal not inverted (positive)
boolean_t	bAddrOnData… LinesOff	TRUE FALSE	Do not output address to data lines (Hi-Z during ALC cycle period) No Hi-Z during ALC cycle period
boolean_t	bMpxcsOff	TRUE FALSE	Do not assert MCSX in ALC cycle period MCSX in ALC ALC asserted
boolean_t	bMoexWidthAs… Fradc	TRUE FALSE	MOEX width is set with FRADC MOEX width is set with RACC-RADC
en_extif_cycle_t	enReadAccess… Cycle*	*(see below)**	Read access cycles*
en_extif_cycle_t	enReadAddress… SetupCycle*	*(see below)**	Read address setup cycles*
en_extif_cycle_t	enFirstRead… AddressCycle*	*(see below)**	First read address cycles*
en_extif_cycle_t	enReadIdle… Cycle*	*(see below)**	Read idle cycles*

Type	Field	Possible Values	Description
en_extif_cycle_t	enWriteAccess… Cycle	*(see below)**	Write access cycles*
en_extif_cycle_t	enWrite… AddressSetup… Cycle	*(see below)**	Write address setup cycles*
en_extif_cycle_t	enWriteEnable… Cycle	*(see below)**	Write enable cycles*
en_extif_cycle_t	enWriteIdle… Cycle	*(see below)**	Write idle cycles*
uint8_t	u8AreaAddress	-	Area address bits [27:20]
en_extif_mask_t	enAreaMask	Extif1MB Extif2MB Extif4MB . . . Extif64MB Extif128MB	EXTIF area 1 MByte EXTIF area 2 MByte EXTIF area 4 MByte ... EXTIF area 64 MByte EXTIF area 128 MByte
en_extif_cycle_t	enAddress… LatchCycle	*(see below)**	Address latch cycles*
en_extif_cycle_t	enAddress… LatchSetup… Cycle	*(see below)**	Address latch setup cycles*
en_extif_cycle_t	enAddress… LatchWidth… Cycle	*(see below)**	Address latch width cycle
boolean_t	bSdramEnable	TRUE FALSE	Enables SDRAM functionality (only possible for area 8) No SDRAM functionality
boolean_t	bSdramPower… DownMode	TRUE FALSE	Enables SDRAM Power Down Mode (only possible for area 8) No SDRAM functionality
boolean_t	bSdramRefresh… Off	TRUE FALSE	Enables SDRAM Power Down Mode (only possible for area 8) No SDRAM functionality
en_extif_cas_t	enCasel	ExtifCas16Bit ExtifCas32Bit	Column address select: MAD[9:0] = Internal address [10:1], 16-Bit width MAD[9:0] = Internal address [11:2], 32-Bit width
en_extif_ras_t	enRasel	ExtifRas_19_6 ExtifRas_20_7 ExtifRas_21_8 ExtifRas_22_9 ExtifRas_23_10 ExtifRas_24_11 ExtifRas_25_12	Row address select: MAD[13:0] = Internal address [19:6] MAD[13:0] = Internal address [20:7] MAD[13:0] = Internal address [21:8] MAD[13:0] = Internal address [22:9] MAD[13:0] = Internal address [23:10] MAD[13:0] = Internal address [24:11] MAD[13:0] = Internal address [25:12]
en_extif_bas_t	enBasel	ExtifBas_20_19 ExtifBas_21_20 ExtifBas_22_21 ExtifBas_23_22 ExtifBas_24_23 ExtifBas_25_24 ExtifBas_26_25	Bank address select: MAD[13:0] = Internal address [20:19] MAD[13:0] = Internal address [21:20] MAD[13:0] = Internal address [22:21] MAD[13:0] = Internal address [23:22] MAD[13:0] = Internal address [24:23] MAD[13:0] = Internal address [25:24] MAD[13:0] = Internal address [26:25]

Type	Field	Possible Values	Description
uint16_t	u16PowerDown… Count	-	Power Down Count in Cycles (only possible for area 8)
en_extif_cycle_t	enSdramCas… LatencyCycle	*(see below)**	SDRAM CAS latency cycles*
en_extif_cycle_t	enSdramRas… Cycle	*(see below)**	SDRAM RAS cycles*
en_extif_cycle_t	enSdramRas… PrechargeCycle	*(see below)**	SDRAM precharge cycles*
en_extif_cycle_t	enSdramRasCas… DelayCycle	*(see below)**	SDRAM RAS-CAS delay cycles*
en_extif_cycle_t	enSdramRas… ActiveCycle	*(see below)**	SDRAM RAS active cycles*
en_extif_cycle_t	enSdram… RefreshCycle	*(see below)**	SDRAM refresh cycles*
en_extif_cycle_t	enSdram… PrechargeCycle	*(see below)**	SDRAM precharge cycles*
boolean_t	bSdramError… Interrupt… Enable	TRUE FALSE	Enable SDRAM error interrupt Disable SDRAM error interrupt
boolean_t	bSramFlash… Error… Interrupt… Enable	TRUE FALSE	Enable SRAM/Flash error interrupt Disable SRAM/Flash error interrupt
func_ptr_t	pfnSdramError… Callback	-	Pointer to SDRAM error callback function
func_ptr_t	pfnSramFlash… ErrorCallback	-	Pointer to SRAM/Flash error callback function
boolean_t	bMclkoutEnable	TRUE FALSE	Enable MCLKOUT pin Disable MCLKOUT pin
uint8_t	u8MclkDivision	1 - 16	Division ratio for MCLK
boolean_t	bPrecedRead… Continuous… Write	TRUE FALSE	Enables preceding read and continuous write request Disable feature

* These cycle setting depend from each other. Refer to the EXTIF chapter of the peripheral manual.

Possible enumerators for `en_extif_cycle_t`:

Enumerator	Explanation
Extif0Cycle	0 cycles
Extif1Cycle	1 cycle
Extif2Cycle	2 cycles
. . .	. . .
Extif15Cycle	15 cycles
Extif16Cycle	16 cycles
ExtifDisabled	Setting disabled

7.14.2 EXTIF API

The following API functions are used for handling the external bus interface.

7.14.2.1 *Extif_InitArea()*

This function initializes an EXTIF area according the configuration.

Prototype	
`en_result_t Extif_InitArea( uint8_t u8Area, stc_extif_area_config_t* pstcConfig );`	
Parameter Name	**Description**
[in] u8Area	Area number (0…8)
[in] pstcConfig	Pointer to configuration structure
Return Values	**Description**
Ok	Area setup successful
ErrorInvalidParameter	pstcConfig == NULL or other parameter illegal
ErrorInvalidMode	SDMODE is set for area different from 8

7.14.2.2 *Extif_ReadErrorStatus()*

This function reads-out the Error Status Register.

Prototype	
`en_result_t Extif_ReadErrorStatus( void );`	
Return Values	**Description**
Ok	No error response exists
Error	Error response exists

7.14.2.3 *Extif_ReadErrorAddress()*

This function returns the address at which the error occurred.

Prototype	
`uint32_t Extif_ReadErrorAddress( void );`	
Return Values	**Description**
Ok	No error response exists
Error	Error response exists

7.14.2.4 Extif_ClearErrorStatus()

This function clears the Error Status Register.

Prototype	
`en_result_t Extif_ClearErrorStatus( void );`	
Return Values	**Description**
`Ok`	Error Status Register cleared

7.14.2.5 Extif_CheckSdcmdReady()

This function checks if the SDRAM is ready for access.

Prototype	
`en_result_t Extif_CheckSdcmdReady( void );`	
Return Values	**Description**
`Ok`	SDRAM ready
`ErrorNotReady`	SDRAM not ready

7.14.2.6 Extif_SetSdramCommand()

This function sets SDRAM commands.

Prototype	
`en_result_t Extif_SetSdramCommand( uint16_t u16Address, boolean_t bMsdwex, boolean_t bMcasx, boolean_t bMrasx, boolean_t bMcsx8, boolean_t bMadcke );`	
Parameters	**Description**
`[in] u16Address`	SDRAM address (`MAD[15:00]` pin values
`bMsdwex`	`TRUE:` `MDSWEX` pin value = 1
`bMcasx`	`TRUE:` `MCASX` pin value = 1
`bMrasx`	`TRUE:` `MRASX` pin value = 1
`bMcsx8`	`TRUE:` `MCSX8` pin value = 1
`bMadcke`	`TRUE:` `MADCKE` pin value = 1
Return Values	**Description**
`Ok`	Writing to SDCMD register was successful
`ErrorNotReady`	Access to SDCMD register was not possible

7.14.2.7 *Callback functions*

The error callback functions for SDRAM or SRAM/Flash have no arguments.

Prototype
void *ExtifErrorCallback* (void);

7.14.3 EXTIF Examples

The PDL example does not provide an EXTIF example yet. Further versions will have such an example.

7.15 (FLASH) Flash Memory

Type Definition	–
Configuration Types	–
Address Operator	–

Flash memeory including 2 parts: Main Flash and Work Flash.

7.15.1 Main Flash

Before using the Main Flash operation APIs, please make sure the code is operated in RAM area.

MFlash_ChipErase() can erase whole chip space of Main Flash, whether CR data remains after chip erase depends on the parameter bCrRemain.

MFlash_SectorErase() can erase one selected sector.

MFlash_Write() writes data into Flash area with word align, as ECC is equipped in the Flash module. Whether data verify and ECC check is done depends on the parameter bVerifyAndEccCheck.

7.15.1.1 *Configuration Structure*

NONE

7.15.1.2 *Main Flash API*

7.15.1.3 MFlash_ChipErase ()

This function erases flash chip.

Prototype	
`uint8_t MFlash_ChipErase(boolean_t bCrRemain);`	
Parameter Name	**Description**
`[in] bCrRemain`	CR remaining flag.
Return Values	**Description**
`MFLASH_RET_OK`	Erase flash successfully.
`MFLASH_RET_INVALID_PARA`	`bCrRemain` is invalid
`MFLASH_RET_ABNORMAL`	The automatic algorithm of flash memory is abnormally completed.

7.15.1.4 MFlash_SectorErase ()

This function erases flash sector.

Prototype	
`uint8_t MFlash_SectorErase(uint16_t* pu16SecAddr);`	
Parameter Name	**Description**
`[in] pu16SecAddr`	Address of flash sector.
Return Values	**Description**
`MFLASH_RET_OK`	Erase flash sector successfully.
`MFLASH_RET_INVALID_PARA`	`pu16SecAddr == 0`
`MFLASH_RET_ABNORMAL`	The automatic algorithm of flash memory is abnormally completed.

7.15.1.5 MFlash_Write ()

This function writes flash half-word with ECC.

Prototype	
`uint8_t MFlash_Write(uint16_t* pu16WriteAddr, uint16_t* pu16WriteData, uint32_t u32EvenSize, boolean_t bVerifyAndEccCheck);`	
Parameter Name	**Description**
[in] pu16WriteAddr	Pointer to the flash address to write.
[in] pu16WriteData	Pointer to the data to write.
[in] u32EvenSize	Data size, 1 indicates 1 16-bit data, always set it to even
[in] bVerifyAndEccCheck	The flag for if verify the readback data.
Return Values	**Description**
MFLASH_RET_OK	Write flash data successfully.
MFLASH_RET_INVALID_PARA	pu16WriteAddr == 0 u32EvenSize%2 != 0
MFLASH_RET_ABNORMAL	The automatic algorithm of flash memory is abnormally completed. Verify data fail. ECC check error.

7.15.2 Work Flash

Work Flash has independent area, which is sperated from Main Flash, thus the Flash operation API can be called derectly from Main Flash.

WFlash_ChipErase() can erase whole chip space of Work Flash.

WFlash_SectorErase() can erase one selected sector.

WFlash_Write() writes data into Flash area with half-word align.

7.15.2.1 Configuration Structure

NONE

7.15.2.2 Work Flash API

7.15.2.3 WFlash_ChipErase ()

This function erases flash chip.

Prototype	
uint8_t WFlash_ChipErase(void);	
Parameter Name	**Description**
-	
Return Values	**Description**
WFLASH_RET_OK	Erase flash successfully.
WFLASH_RET_ABNORMAL	The automatic algorithm of flash memory is abnormally completed.

7.15.2.4 *WFlash_SectorErase ()*

This function erases flash sector.

Prototype	
uint8_t WFlash_SectorErase(uint16_t* pu16SecAddr);	
Parameter Name	**Description**
[in] pu16SecAddr	Address of flash sector.
Return Values	**Description**
WFLASH_RET_OK	Erase flash sector successfully.
WFLASH_RET_INVALID_PARA	pu16SecAddr == 0
WFLASH_RET_ABNORMAL	The automatic algorithm of flash memory is abnormally completed.

7.15.2.5 *MFlash_Write ()*

This function writes flash half-word with ECC.

Prototype	
uint8_t WFlash_Write(uint16_t* pu16WriteAddr, uint16_t* pu16WriteData, uint32_t u32Size);	
Parameter Name	**Description**
[in] pu16WriteAddr	Pointer to the flash address to write.
[in] pu16WriteData	Pointer to the data to write.
[in] u32Size	Data size.
Return Values	**Description**
WFLASH_RET_OK	Write flash data successfully.
WFLASH_RET_INVALID_PARA	pu16WriteAddr == 0 u32Size != 0
MFLASH_RET_ABNORMAL	The automatic algorithm of flash memory is abnormally completed.

7.16 (GPIO) General Purpose I/O Ports

Type Definition	–
Configuration Type	–
Address Operator	–

The GPIO modules only consist of header files for each device. The naming is *gpio_mb9bf[0-9AB][0-9AB]x.[klmnst].h*. These particular header files are included in the common header file *gpio1pin.h*, which takes care of the device and package.

These particular header files consist of two blocks for each functional pin. There are definitions of pseudo functions for:

- Set port pins
- Set resource pins inclusive relocation

Attention:

- Carefully adjust your device and package described in chapter 4.4. Wrong pin usage may be caused, if the device and package is set wrongly!
- If External Bus Interface pins with prefix "**`_1`**" are provided by the package, the user ***must*** set the **`UERLC`** bit of `EPFR11` manually! It can be done by the following instruction:

```
bGPIO_EPFR11_UERLC = 1;
```

- Carefully check in device documentation, whether SOUBOUT pin at SOUBOUT[_n] or TIOB0 pin should be output. TIOB0-SUBOUT is not provided by this driver.
- Internal LSYN connection is not provided by this driver.

7.16.1 GPIO Macro API

The following API macros are used for handling the GPIO ports for peripheral functionality or GPIO usage.

7.16.1.1 Gpio1pin_InitIn()

This macro sets a port to digital input.

Macro		
`Gpio1pin_InitIn( p, settings );`		
Parameter Name	**Description**	
`p`	Port Pin Name	
	`GPIO1PIN_P`*xy*	Normal logic
	`GPIO1PIN_NP`*xy*	Inverted logic
`settings`	Settings	
	`Gpio1pin_InitPullup()`	Pull-up: `0` = OFF, `1` = ON

7.16.1.2 Gpio1pin_InitOut()

This macro sets a port to digital output.

<table>
<tr><th colspan="3">Macro</th></tr>
<tr><td colspan="3"><code>Gpio1pin_InitOut(p, settings);</code></td></tr>
<tr><th>Parameter Name</th><th colspan="2">Description</th></tr>
<tr><td rowspan="3"><code>p</code></td><td colspan="2">Port Pin Name</td></tr>
<tr><td><code>GPIO1PIN_P</code>xy</td><td>Normal logic</td></tr>
<tr><td><code>GPIO1PIN_NP</code>xy</td><td>Inverted logic</td></tr>
<tr><td rowspan="2"><code>settings</code></td><td colspan="2">Settings</td></tr>
<tr><td><code>Gpio1pin_InitVal()</code></td><td>Initial value <code>0</code> or <code>1</code></td></tr>
</table>

7.16.1.3 Gpio1pin_Get()

This macro reads out a GPIO port via `PDIR`.

<table>
<tr><th colspan="3">Macro</th></tr>
<tr><td colspan="3"><code>Gpio1pin_Get(p);</code></td></tr>
<tr><th>Parameter Name</th><th colspan="2">Description</th></tr>
<tr><td rowspan="3"><code>p</code></td><td colspan="2">Port Pin Name</td></tr>
<tr><td><code>GPIO1PIN_P</code>xy</td><td>Normal logic</td></tr>
<tr><td><code>GPIO1PIN_NP</code>xy</td><td>Inverted logic</td></tr>
<tr><td colspan="3">Return Values</td></tr>
<tr><td rowspan="2"></td><td colspan="2">Values</td></tr>
<tr><td colspan="2"><code>0</code> or <code>1</code></td></tr>
</table>

7.16.1.4 Gpio1pin_Put()

This macro sets a GPIO port via `PDOR`.

<table>
<tr><th colspan="3">Macro</th></tr>
<tr><td colspan="3"><code>Gpio1pin_Put(p, v);</code></td></tr>
<tr><th>Parameter Name</th><th colspan="2">Description</th></tr>
<tr><td rowspan="3"><code>p</code></td><td colspan="2">Port Pin Name</td></tr>
<tr><td><code>GPIO1PIN_P</code>xy</td><td>Normal logic</td></tr>
<tr><td><code>GPIO1PIN_NP</code>xy</td><td>Inverted logic</td></tr>
<tr><td rowspan="2"><code>v</code></td><td colspan="2">Values</td></tr>
<tr><td colspan="2"><code>0</code> or <code>1</code></td></tr>
</table>

7.16.1.5 *SetPinFunc_PINNAME()*

This macro sets the pin function to peripheral usage and adjusts the `EPFR` register for pin relocation. Additionally a possible analog functionality is switched off in the `ADE` register.

PINNAME is the package pin name followed by a possible relocation suffix.

Macro
`SetPinFunc_`*PINNAME*`[_`*n*`]();`

Example for RTO01 at pin relocation 1:

```
SetPinFunc_RTO01_1();
```

7.16.2 GPIO additional Macros

Since PDL version 1.1 additional macros has been developed to allow the user to parameterize port names and settings. Note that only positive logic can be used.

7.16.2.1 *GpioInitIn()*

This macro sets a GPIO port to input functionality.

Macro		
`GpioInitIn( port, pullup );`		
Parameter Name	**Description**	
`port`	Port Pin Name	
	`P`*xy*	Port name
`pullup`	Values	
	`0` (on) or `1` (off)	

7.16.2.2 *GpioInitOut()*

This macro sets a GPIO port to output functionality with a given value.

Macro		
`GpioInitOut( port, value );`		
Parameter Name	**Description**	
`port`	Port Pin Name	
	`P`*xy*	Port name
`value`	Values	
	`0` (VSS) or `1` (VCC)	

7.16.2.3 *GpioGet()*

This macro reads out a port state. The corresponding port has to be set to input before.

Macro		
`GpioGet( port );`		
Parameter Name	**Description**	
`port`	Port Pin Name	
	`P`*xy*	Port name

7.16.2.4 *GpioPut()*

This macro sets a given value to a GPIO port. The corresponding port has to be set to output before.

Macro		
`GpioPut( port, value );`		
Parameter Name	**Description**	
`port`	Port Pin Name	
	`P`*xy*	Port name
`value`	Values	
	`0` (VSS) or `1` (VCC)	

7.16.3 GPIO Examples

The PDL example folder contains two GPIO usage examples:

- gpio_ports — Examples for GPIO macro usage.
- gpio_parameterized_ports — Example of additional GPIO macros

7.17 (HWWDG) Hardware Watch Dog

Type Definition	-
Configuration Type	`stc_hwwdg_config_t`
Address Operator	-

The HWWDG APIs have dedicated interrupt callback functions, in which the user need to feed the HWWDG block.

`Hwwdg_Init()` sets the interval time. `Hwwdg_Feed()` resets the HWWDG block timer by a function call. `Hwwdg_QuickFeed()` does the same, but the code is inline expanded. For example, it is for the time-critical polling loops.

`Hwwdg_Init()` sets the Hardware Watchdog interval.

`Hwwdg_DeInit()` stops and disables the HWWDG block.

`Hwwdg_Start()` starts the HWWDG block counter.

`Hwwdg_Stop()` stops the HWWDG block counter..

`Hwwdg_WriteWdgLoad()` writes the load value for the HWWDG block counter.

`Hwwdg_ReadWdgValue()` reads the value of the HWWDG block counter.

`Hwwdg_Feed()` and `Hwwdg_QuickFeed()` do the same as their correspondig functions for the Software Watchdog, but here are two parameter needed, the 2nd one the inverted value of the 1st.

Notes:

- The HWWDG block shares the interrupt vector with the NMI.
- The HWWDG block is switched off with `System_Init()` in *system_mb9[ab]xyz.c*.
- If setting the definition for `HWWD_DISABLE` to 0 in *system_mb9[ab]xyz.h,* the HWWDG block runs during the start-up phase.

7.17.1 HWWDG Configuration Structure

The argument of `Hwwdg_Init()` is a pointer to a structure of the HWWDG configuration. The type of the structure is `stc_hwwdg_config_t`. The members of `stc_hwwdg_config_t` are:

Type	Field	Possible Values	Description
`uint32_t`	`u32LoadValue`	`0x00000001 - 0xFFFFFFFF`	Interval value
`boolean_t`	`bResetEnable`	`TRUE` `FALSE`	Enables Hardware watchdog reset Disables Hardware watchdog reset

7.17.2 API reference

7.17.2.1 *Hwwdg_Init()*

Initializes the HWWDG block.

Prototype	
`en_result_t Hwwdg_Init(stc_hwwdg_config_t* pstcConfig)`	
Parameter Name	**Description**
[in] `pstcConfig`	A pointer to a structure of the HWWDG configuration
Return Values	**Description**
`Ok`	Initialization ended with no error
`ErrorInvalidParameter`	`pstcConfig == NULL`

7.17.2.2 *Hwwdg_DeInit()*

Deinitializes the HWWDG block when the first argument is `0xC72E51A3` and the second argument is `0x89DB2E43`.

Prototype	
`en_result_t Hwwdg_DeInit(uint32_t u32MagicWord1, uint32_t u32MagicWord2)`	
Parameter Name	**Description**
[in] `u32MagicWord1`	The 1st Magic word (`0xC72E51A3`)
[in] `u32MagicWord2`	The 2nd Magic word (`0x89DB2E43`)
Return Values	**Description**
`Ok`	The interrupt is desabiled and the HWWDG block stops
`ErrorInvalidParameter`	The magic word(s) is(are) incorrect

7.17.2.3 *Hwwdg_Start()*

Starts the HWWDG block.

Prototype	
`en_result_t Hwwdg_Start(func_ptr_t pfnHwwdgCb)`	
Parameter Name	**Description**
[in] `pfnHwwdgCb`	A pointer to a callback function (can be set to NULL)
Return Values	**Description**
`Ok`	The HWWDG modle starts with no error
`ErrorOperationInProgress`	The HWWDG modle is already in motion

7.17.2.4 Hwwdg_Stop()

Stops the HWWDG block.

Prototype
`void Hwwdg_Stop(void)`

7.17.2.5 Hwwdg_WriteWdgLoad()

Sets the HWWDG block timer load value.

Prototype	
`void Hwwdg_WriteWdgLoad(uint32_t u32LoadValue)`	
Parameter Name	**Description**
[in] `u32LoadValue`	The load value

7.17.2.6 Hwwdg_ReadWdgValue()

Reterns the timer value from the HWWDG block.

Prototype	
`uint32_t Hwwdg_ReadWdgValue(void)`	
Return Values	**Description**
`uint32_t`	The value of the value register

7.17.2.7 Hwwdg_Feed()

Feeds the HWWDG block with unlock, feed, and lock sequence.

Prototype	
`void Hwwdg_Feed(uint8_t u8ClearPattern1, uint8_t u8ClearPattern2)`	
Parameter Name	**Description**
[in] `u8ClearPattern1`	The arbitrary value
[in] `u8ClearPattern2`	The inverted arbitrary value

7.17.2.8 Hwwdg_EnableDbgBrkWdgCtl()

Enable counting of the HWWDG block timer during a tool break.

Prototype
`void Hwwdg_EnableDbgBrkWdgCtl(void)`

7.17.2.9 Hwwdg_DisableDbgBrkWdgCtl()

Disables counting of the HWWDG block timer during a tool break (default).

Prototype
`void Hwwdg_DisableDbgBrkWdgCtl(void)`

7.17.2.10 Static Inline Function

For the quick feed, the HWWDG API provides a feed function which is defined as the static inline in ***hwwdg.h***.

Hwwdg_QuickFeed()

Feeds the Hardware watchdog with unlock, feed, and lock sequence.

Prototype	
`static __INLINE void Hwwdg_Feed( uint8_t u8ClearPattern1, uint8_t u8ClearPattern2)`	
Parameter Name	**Description**
[in] `u8ClearPattern1`	The arbitrary value
[in] `u8ClearPattern2`	The inverted arbitrary value

7.17.2.11 Callback Function

The callback function is registered by `Crc_Start()`.The callback function is called in the interrupt handler when the HWWDG generates interrupts.

Prototype
`void (*func_prt_t)(void)`

7.17.3 Example Code

The example software is in *\example\wdg\hwwdg*.
This code excerpt shows how to use the HWWDG APIs.

```
#include "wdg/hwwdg.h"

static void WdgHwCallback(void)
{
  Hwwdg_Feed(0x55, 0xAA); // Only for example! Do not use this in your
                          //   application!
  // some code here ...
}

function
{
  stc_hwwdg_config_t stcHwwdgConfig;

  ...
  stcHwwdgConfig.u32LoadValue = 100000;       // Interval:1s
(@CLKLC:100kHz)
  stcHwwdgConfig.bResetEnable = TRUE;         // Enables Hardware
watchdog reset
  // Initialize hardware watchdog
  if (Ok != Hwwdg_Init(&stcHwwdgConfig))
  {
    // some code here ...
  }
  else
  {
    // Start hardware watchdog
    Hwwdg_Start(WdgHwCallback);
  }

  // wait for interrupts
  while (1);
}
```

7.18 (LPM) Low Power Modes

Type Definition	-
Configuration Type	-
Address Operator	-

This module provides API functions for Low Power Modes.

Note: This driver module requires the CLK and EXINT module to be activated. For accessing the Back-up Registers only these two additional modules do not need to be activated.

The LPM driver does not need any configuration.

7.18.1 LPM API

The following API functions are used for handling the Low Power Modes.

7.18.1.1 Lpm_SetHsCrSleep()

This API function transits the MCU to the High-Speed CR Clock Sleep Mode.

Prototype	
`en_result_t Lpm_SetHsCrSleep( boolean_t bDisablePllMainClock, boolean_t bDisableSubClock );`	
Parameter Name	**Description**
`[in] bDisablePllMainClock`	`TRUE`: Disable PLL Clock and Main Clock
`[in] bDisableSubClock`	`TRUE`: Disable Sub Clock
Return Values	**Description**
`Ok`	Returned from HS-CR Sleep Mode

7.18.1.2 Lpm_SetMainSleep()

This API function transits the MCU to the Main Clock Sleep Mode.

Prototype	
`en_result_t Lpm_SetMainSleep( boolean_t bDisablePllClock, boolean_t bDisableSubClock );`	
Parameter Name	**Description**
`[in] bDisablePllClock`	`TRUE`: Disable PLL Clock
`[in] bDisableSubClock`	`TRUE`: Disable Sub Clock
Return Values	**Description**
`Ok`	Returned from Main Clock Sleep Mode

7.18.1.3 Lpm_SetPllSleep()

This API function transits the MCU to the PLL Clock Sleep Mode.

Prototype	
`en_result_t Lpm_SetMainSleep( boolean_t bDisablePllClock, boolean_t bDisableSubClock );`	
Parameter Name	**Description**
`[in] bDisableSubClock`	`TRUE`: Disable Sub Clock
Return Values	**Description**
`Ok`	Returned from PLL Clock Sleep Mode

7.18.1.4 Lpm_SetLsCrSleep()

This API function transits the MCU to the Low-Speed CR Clock Sleep Mode.

Prototype	
`en_result_t Lpm_SetLsCrSleep( boolean_t bDisablePllMainClock, boolean_t bDisableSubClock );`	
Parameter Name	**Description**
[in] bDisablePllMainClock	TRUE: Disable PLL Clock and Main Clock
[in] bDisableSubClock	TRUE: Disable Sub Clock
Return Values	**Description**
Ok	Returned from LS-CR Sleep Mode

7.18.1.5 Lpm_SetLsCrSleep()

This API function transits the MCU to the Low-Speed CR Clock Sleep Mode.

Prototype	
`en_result_t Lpm_SetLsCrSleep( boolean_t bDisablePllMainClock, boolean_t bDisableSubClock );`	
Parameter Name	**Description**
[in] bDisablePllMainClock	TRUE: Disable PLL Clock and Main Clock
[in] bDisableSubClock	TRUE: Disable Sub Clock
Return Values	**Description**
Ok	Returned from LS-CR Sleep Mode

7.18.1.6 Lpm_SetLsCrSleep()

This API function transits the MCU to Sub Sleep Mode.

Prototype	
`en_result_t Lpm_SetSubSleep( boolean_t bDisablePllMainClock );`	
Parameter Name	**Description**
[in] bDisablePllMainClock	TRUE: Disable PLL Clock and Main Clock
[in] bDisableSubClock	TRUE: Disable Sub Clock
Return Values	**Description**
Ok	Returned from LS-CR Sleep Mode

7.18.1.7 Lpm_SetPllTimer ()

This API function transits the MCU to PLL Timer Mode. The system has to be in stabilized PLL mode before calling this function.

Prototype	
`en_result_t Lpm_SetPllTimer( boolean_t bDisableSubClock );`	
Parameter Name	**Description**
`[in] bDisableSubClock`	`TRUE`: Disable Sub Clock
Return Values	**Description**
`Ok`	Returned from PLL Timer Mode
`ErrorNotReady`	PLL clock not available

7.18.1.8 Lpm_SetHsCrTimer ()

This API function transits the MCU to High-Speed CR clock Timer Mode. The system has to be in stabilized PLL mode before calling this function.

Prototype	
`en_result_t Lpm_SetHsCrTimer( boolean_t bDisablePllMainClock,` `boolean_t bDisableSubClock );`	
Parameter Name	**Description**
`[in] bDisablePllMainClock`	`TRUE`: Disable PLL Clock and Main Clock
`[in] bDisableSubClock`	`TRUE`: Disable Sub Clock
Return Values	**Description**
`Ok`	Returned from HS-CR Timer Mode

7.18.1.9 Lpm_SetMainTimer ()

This API function transits the MCU to Main clock Timer Mode.

Prototype	
`en_result_t Lpm_SetMainTimer( boolean_t bDisablePllClock,` `boolean_t bDisableSubClock );`	
Parameter Name	Description
[in] bDisablePllClock	**TRUE: Disable PLL Clock**
`[in] bDisableSubClock`	`TRUE`: Disable Sub Clock
Return Values	**Description**
`Ok`	Returned from Main Timer Mode

7.18.1.10 Lpm_SetLsCrTimer ()

This API function transits the MCU to Low-Speed CR clock Timer Mode.

Prototype	
`en_result_t Lpm_SetLsCrTimer( boolean_t bDisablePllMainClock, boolean_t bDisableSubClock );`	
Parameter Name	**Description**
`[in] bDisablePllMainClock`	`TRUE`: Disable PLL Clock and Main Clock
`[in] bDisableSubClock`	`TRUE`: Disable Sub Clock
Return Values	**Description**
`Ok`	Returned from LS-CR Timer Mode

7.18.1.11 Lpm_SetSubTimer ()

This API function transits the MCU to Sub clock Timer Mode.

Prototype	
`en_result_t Lpm_SetSubTimer( boolean_t bDisablePllMainClock );`	
Parameter Name	**Description**
`[in] bDisablePllMainClock`	`TRUE`: Disable PLL Clock and Main Clock
Return Values	**Description**
`Ok`	Returned from Sub Clock Timer Mode

7.18.1.12 Lpm_SetRtcMode ()

This API function transits the MCU to RTC Mode. This function requires a properly operating RTC.

Prototype	
`en_result_t Lpm_SetRtcMode( void );`	
Return Values	**Description**
`Ok`	Returned from RTC Mode

7.18.1.13 Lpm_SetStopMode ()

This API function transits the MCU to Stop Mode.

Prototype	
`en_result_t Lpm_SetStopMode( void );`	
Return Values	**Description**
`Ok`	Returned from Stop Mode

7.18.1.14 Lpm_SetDeepRtcMode ()

This API function transits the MCU to Deep RTC Mode. This function requires a properly operating RTC.

Prototype	
`en_result_t Lpm_SetDeepRtcMode( void );`	
Return Values	**Description**
Ok	Returned from Deep RTC Mode

7.18.1.15 Lpm_SetDeepStopMode ()

This API function transits the MCU to Deep Stop Mode.

Prototype	
`en_result_t Lpm_SetDeepStopMode( void );`	
Return Values	**Description**
Ok	Returned from Deep Stop Mode

7.18.1.16 Lpm_ReadBackupRegisters()

This API function does not require the CLK and EXINT driver activated. It writes the contents of the Back UP Registers into a structure of the type of `stc_backupreg_t` .

Prototype	
`en_result_t Lpm_ReadBackupRegisters( stc_backupreg_t* stcBackUpReg );`	
Parameter Name	**Description**
[out] stcBackUpReg	Pointer to structure to be filled with Back Up Register contents
Return Values	**Description**
Ok	Successfully read

Structure of the type of `stc_backupreg_t` .

Element Type	Element Name
uint8_t	u8BREG00
uint8_t	u8BREG01
. . .	. . .
uint8_t	u8BREG1F

7.18.1.17 Lpm_WriteBackupRegisters()

This API function does not require the CLK and EXINT driver activated. It writes the contents of the structure of the type of `stc_backupreg_t` to the Back Up Registers. The structure is the same as described in paragraph above (7.18.1.16).

Prototype	
`en_result_t Lpm_WriteBackupRegisters( stc_backupreg_t* stcBackUpReg );`	
Parameter Name	**Description**
`[in] stcBackUpReg`	Pointer to structure which fills the Back Up Register
Return Values	**Description**
`Ok`	Successfully written

7.18.1.18 Lpm_Read_u8DataBackupRegister()

This API function does not require the CLK and EXINT driver activated. It reads a single byte from a Back Up Registers.

Prototype	
`en_result_t Lpm_Read_u8DataBackupRegister( uint8_t  u8AddressOffset, uint8_t* u8Data );`	
Parameter Name	**Description**
`[in] u8AddressOffset`	Address offset to register (0…31)
`[out] u8Data`	Pointer to data byte, which is filled by the Back Up Register content
Return Values	**Description**
`Ok`	Successfully read
`ErrorInvalidParameter`	Address offset out of range

7.18.1.19 Lpm_Write_u8DataBackupRegister()

This API function does not require the CLK and EXINT driver activated. It writes a single byte to a Back Up Registers.

Prototype	
`en_result_t Lpm_Write_u8DataBackupRegister( uint8_t u8AddressOffset, uint8_t u8Data );`	
Parameter Name	**Description**
`[in] u8AddressOffset`	Address offset to register (0…31)
`[out] u8Data`	Data byte to be written to a Back Up Register
Return Values	**Description**
`Ok`	Successfully written
`ErrorInvalidParameter`	Address offset out of range

7.18.1.20 Lpm_Read_u16DataBackupRegister()

This API function does not require the CLK and EXINT driver activated. It reads a 16-bit word from a Back Up Registers.

Prototype	
`en_result_t Lpm_Read_u16DataBackupRegister( uint8_t u8AddressOffset, uint16_t* u16Data );`	
Parameter Name	**Description**
[in] u8AddressOffset	Address offset to register (0…31)
[out] u16Data	Pointer to 16-bit word, which is filled by the Back Up Register content
Return Values	**Description**
Ok	Successfully read
ErrorInvalidParameter	Address offset out of range
ErrorAddressAlignment	Address not aligned to 16-bit

7.18.1.21 Lpm_Write_u16DataBackupRegister()

This API function does not require the CLK and EXINT driver activated. It writes a 16-bit word to a Back Up Registers.

Prototype	
`en_result_t Lpm_Write_u16DataBackupRegister( uint8_t u8AddressOffset, uint16_t u16Data );`	
Parameter Name	**Description**
[in] u8AddressOffset	Address offset to register (0…31)
[out] u16Data	16-bit word to be written to a Back Up Register
Return Values	**Description**
Ok	Successfully written
ErrorInvalidParameter	Address offset out of range
ErrorAddressAlignment	Address not aligned to 16-bit

7.18.1.22 *Lpm_Read_u32DataBackupRegister()*

This API function does not require the CLK and EXINT driver activated. It reads a 32-bit word from a Back Up Registers.

Prototype	
`en_result_t Lpm_Read_u32DataBackupRegister( uint8_t u8AddressOffset, uint32_t* u32Data );`	
Parameter Name	**Description**
`[in] u8AddressOffset`	Address offset to register (0…31)
`[out] u32Data`	Pointer to 32-bit word, which is filled by the Back Up Register content
Return Values	**Description**
`Ok`	Successfully read
`ErrorInvalidParameter`	Address offset out of range
`ErrorAddressAlignment`	Address not aligned to 32-bit

7.18.1.23 *Lpm_Write_u32DataBackupRegister()*

This API function does not require the CLK and EXINT driver activated. It writes a 32-bit word to a Back Up Registers.

Prototype	
`en_result_t Lpm_Write_u32DataBackupRegister( uint8_t u8AddressOffset, uint32_t u32Data );`	
Parameter Name	**Description**
`[in] u8AddressOffset`	Address offset to register (0…31)
`[out] u32Data`	32-bit word to be written to a Back Up Register
Return Values	**Description**
`Ok`	Successfully written
`ErrorInvalidParameter`	Address offset out of range
`ErrorAddressAlignment`	Address not aligned to 32-bit

7.18.2 LPM Example

The PDL example folder contains an LPM usage example:

- lpm_bu_reg Access to the Back Up Registers.
- lpm_sleep_stop Entering sleep and stop mode (requires external wake-up source)

7.19 (LVD) Low Voltage Detection

Type Definition	-
Configuration Type	`stc_lvd_config_t`
Address Operator	-

`Lvd_Init()` initializes the LVD block with the given voltage threshold. If the callback function is not specified when `Lvd_Init()` is called, NVIC is not enabled. In this case the user needs to check the interrupt by `Lvd_GetIntStatus()` and clear the interrupt by `Lvd_ClearIntStatus()`.

`Lvd_DeInit()` disables the LVD block.

`Lvd_GetIntOperationStatus()` is used for checking the operation status of the interrupt.

7.19.1 LVD Configuration Structure

The argument of `Lvd_Init()` is a pointer to a structure of the LVD configuration. The type of the structure is `stc_lvd_config_t`. The members of `stc_lvd_config_t` are:

Type	Field	Possible Values	Description
`en_lvd_irq_voltage_t`	`enIrqVoltage`	See 10.1.1	The threshold Voltage for the interrupt
`func_ptr_t`	`pfnCallback`	-	A pointer to an interrupt callback function

7.19.1.1 Interrupt Voltage Enumerators

The following enumerators are used for `en_lvd_irq_voltqage_t enIrqVoltage`.

Enumerator	Description
`LvdIrqVoltage220`	The interrupt occurs when voltage is arround 2.20 volts
`LvdIrqVoltage240`	The interrupt occurs when voltage is arround 2.40 volts
`LvdIrqVoltage245`	The interrupt occurs when voltage is arround 2.45 volts
`LvdIrqVoltage260`	The interrupt occurs when voltage is arround 2.60 volts
`LvdIrqVoltage270`	The interrupt occurs when voltage is arround 2.70 volts
`LvdIrqVoltage280`	The interrupt occurs when voltage is arround 2.80 volts
`LvdIrqVoltage290`	The interrupt occurs when voltage is arround 2.90 volts
`LvdIrqVoltage300`	The interrupt occurs when voltage is arround 3.00 volts
`LvdIrqVoltage320`	The interrupt occurs when voltage is arround 3.20 volts
`LvdIrqVoltage350`	The interrupt occurs when voltage is arround 3.50 volts
`LvdIrqVoltage360`	The interrupt occurs when voltage is arround 3.60 volts
`LvdIrqVoltage370`	The interrupt occurs when voltage is arround 3.70 volts
`LvdIrqVoltage380`	The interrupt occurs when voltage is arround 3.80 volts
`LvdIrqVoltage390`	The interrupt occurs when voltage is arround 3.90 volts
`LvdIrqVoltage400`	The interrupt occurs when voltage is arround 4.00 volts
`LvdIrqVoltage410`	The interrupt occurs when voltage is arround 4.10 volts
`LvdIrqVoltage420`	The interrupt occurs when voltage is arround 4.20 volts
`LvdIrqVoltage430`	The interrupt occurs when voltage is arround 4.30 volts
`LvdIrqVoltage452`	The interrupt occurs when voltage is arround 4.52 volts

7.19.2 API Reference

7.19.2.1 Lvd_Init()

Initializes the LVD block. The user sets the threshold voltage for the interrupt and set the callback function to the interrupt.

Prototype	
`en_result_t Crc_Init(stc_lvd_config_t* pstcConfig)`	
Parameter Name	**Description**
[in] `pstcConfig`	A pointer to a structure of the LVD configuration
Return Values	**Description**
`Ok`	Initialization ended with no error
`ErrorInvalidParameter`	`pstcConfig == NULL` The parameter was out of range

7.19.2.2 Lvd_DeInit()

Disables interrupts and deinitializes the LVD block.

Prototype	
`en_result_t Lvd_DeInit(void)`	
Return Values	**Description**
`Ok`	Disabling interrupt and deinitializing the LVD block ended with no error

7.19.2.3 Lvd_GetIntStatus()

Returns interrupts (low-voltage detection). This function can be called only when interrupts are disabled.

Prototype	
`boolean_t Lvd_GetIntStatus(void)`	
Return Values	**Description**
`boolean_t`	The status of the low-voltage detection `TRUE`: The low-voltage was detected. `FALSE`: The low-voltage was not detected.

7.19.2.4 Lvd_ClrIntStatus()

Clears the interrupt status.

Prototype
`void Lvd_ClearIntStatus(void)`

7.19.2.5 *Lvd_GetIntStatus()*

Returns the operation status.

Prototype	
`boolean_t Lvd_GetIntOperationStatus(void)`	
Return Values	**Description**
`boolean_t`	The operation status `TRUE`: In monitoring state `FALSE`: In stabilization wait state or monitoring stop state

7.19.2.6 *Callback Function*

`Crc_Init()` registers the callback function.
The callback function is called in the interrupt handler when the interrupts are generated.

Prototype
`void (*func_prt_t)(void)`

7.19.3 Example Code

The example software is in *\example\lvd*.

Folder *\example\lvd\...*	Summary
`lvd_unuse_int_27`	Low voltage detection at vicinity of 2.7 volts un-using interrupt
`lvd_unuse_int_28`	Low voltage detection at vicinity of 2.8 volts un-using interrupt
`lvd_unuse_int_30`	Low voltage detection at vicinity of 2.7 volts un-using interrupt
`lvd_use_int_26`	Low voltage detection at vicinity of 2.6 volts using interrupt
`lvd_use_int_28`	Low voltage detection at vicinity of 2.8 volts using interrupt
`lvd_use_int_32`	Low voltage detection at vicinity of 3.2 volts using interrupt

7.19.3.1 *Configuration*

The following example configuration may be used for the example code.

```
#include "lvd/lvd.h"

function
{
  // Interrupt when voltage is vicinity of 2.8 volts
  stcLvdConfig.enIrqVoltage = LvdIrqVoltage280;
  // Clear callback (Un-use interrupt)
  stcLvdConfig.pfnCallback = NULL;
}
```

7.19.3.2 LVD Without Interrupt

```
#include "lvd/lvd.h"

function
{
  ...
  // Interrupt when voltage is vicinity of 2.8 volts
  stcLvdConfig.enIrqVoltage = LvdIrqVoltage280;
  // Clear callback (Un-use interrupt)
  stcLvdConfig.pfnCallback = NULL;

  // Initialize LVD
  if (Ok != Lvd_Init(&stcLvdConfig))
  {
    // some code here ...
    while(1);
  }

  // Wait to change status
  while (FALSE == Lvd_GetIntOperationStatus());
  ...
  while(1)
  {
    // If LVD is detected...
    if (TRUE == Lvd_GetIntStatus())
    {
      // Clear interrupt status
      Lvd_ClearIntStatus();
      // some code here ...
    }
  }
}
```

7.19.3.3 LVD With Interrupt

```
#include "lvd/lvd.h"

static void LvdCallback(void)
{
  // Clear interrupt status
  Lvd_ClearIntStatus();
  // some code here...
}

function
{
  ...
  // Interrupt when voltage is vicinity of 2.8 volts
  stcLvdConfig.enIrqVoltage = LvdIrqVoltage280;
  // Set callback (Use interrupt)
  stcLvdConfig.pfnCallback = LvdCallback;

  // Initialize LVD
  if (Ok != Lvd_Init(&stcLvdConfig))
  {
    // some code here ...
    while(1);
  }

  // Wait to change status
  while (FALSE == Lvd_GetIntOperationStatus());
  ...
  while(1)
  {
    // some code here...
  }
}
```

7.20 (MFT) Multi Function Timer

The multifunction timer is a function block that enables three-phase motor control. In conjunction with a PPG and an A/D converter (called "ADC" hereafter), it can provide a variety of motor controls.

MFT (1 unit) consists of the following function blocks:

FRT (Free-run Timer) Unit

An FRT is a timer function block that outputs counter values for the operational criteria of the function blocks in the MFT. The MFT employs 3 channels, which can operate independently from one another. Inside MFT, the output of the FRT counter value is connected to the OCU, ICU and ADCMP.

OCU (Output Compare Unit)

An OCU is a function block that generates and outputs PWM signals on the basis of the counter values of the FRT. The OCU employs 6 channels (2 channels × 3 units).

ICU (Input Capture) Unit

An ICU is a function block that captures the FRT count value and generates an interrupt in the CPU when a valid edge is detected in an external input pin signal. The ICU employs 4 channels (2 channels × 2 units).

ADCMP (ADC Start Compare Unit)

An ADCMP is a function block that generates AD conversion start signals on the basis of the FRT counter value. The ADCMP employs 6 channels.

7.20.1 FRT (Free-run Timer Unit)

Type Definition	-
Configuration Types	stc_mft_frt_config_t stc_frt_int_sel_t stc_frt_int_cb_t stc_mft_frt_intern_data_t stc_mft_frt_instance_data_t
Address Operator	-

Mft_Frt_Init() must be used for configuration of a Free-Run timer (FRT) channel with a structure of the type stc_mft_frt_config_t.

Mft_Frt_SetMaskZeroTimes() is used to set the mask times of Zero match interrupt.
Mft_Frt_GetMaskZeroTimes() is used to get the mask times of peak match interrupt.

Mft_Frt_SetMaskPeakTimes() is used to set the mask times of peak match interrupt.
Mft_Frt_GetMaskPeakTimes() is used to get the mask times of peak match interrupt.

A FRT interrupt can be enabled by the function Mft_Frt_EnableInt(). This function can set callback function for each channel too.

With Mft_Frt_SetCountCycle() the FRT cycle is set to the value given in the parameter Mft_Frt_SetCountCycle#u16Cycle. And the initial count value can be modified by Mft_Frt_SetCountVal().

After above setting, calling Mft_Frt_Start() will start FRT.

With Mft_Frt_GetCurCount() the current FRT count can be read when FRT is running.

With interrupt mode, when the interrupt occurs, the interrupt flag will be cleared and run into user interrupt callback function.

With polling mode, user can use Mft_Frt_GetIntFlag() to check if the

interrupt occurs, and clear the interrupt flag by Mft_Frt_ClrIntFlag().

When stopping the FRT, use Mft_Frt_Stop() to disable FRT and Mft_Frt_DisableInt() to disable FRT interrupt.

7.20.1.1 Configuration Structure

A FRT configure instance uses the following configuration structure of the type of stc_mft_frt_config_t:

Type	Field	Possible Values	Description
en_mft_frt_clock_t	enFrtClockDiv	 FrtPclkDiv1 FrtPclkDiv2 FrtPclkDiv4 FrtPclkDiv8 FrtPclkDiv16 FrtPclkDiv32 FrtPclkDiv64 FrtPclkDiv128 FrtPclkDiv256 FrtPclkDiv512 FrtPclkDiv1024	FRT clock divide PCLK PCLK/2 PCLK/4 PCLK/8 PCLK/16 PCLK/32 PCLK/64 PCLK/128 PCLK/256 PCLK/512 PCLK/1024
en_mft_frt_mode_t	enFrtMode	 FrtUpCount FrtUpDownCount	FRT count mode FRT up-count mode FRT up-/down-count mode
boolean_t	bEnBuffer	 TRUE FALSE	enable buffer
boolean_t	bEnExtClock	 TRUE FALSE	enable external clock

A FRT interrupt selection instance uses the following configuration structure of the type of stc_frt_int_sel_t:

Type	Field	Possible Values	Description
boolean_t	bFrtZeroMatchInt	TRUE FALSE	zero match interrupt selection select not select
boolean_t	bFrtPeakMatchInt	TRUE FALSE	peak match interrupt selection select not select

A FRT interrupt callback function instance uses the following configuration structure of the type of stc_frt_int_cb_t:

Type	Field	Possible Values	Description
func_ptr_t	pfnFrtZeroCallback	-	match zero interrupt callback function
func_ptr_t	pfnFrtPeakCallback	-	match peak interrupt callback function

An FRT internal data instance uses the following configuration structure of the type of stc_mft_frt_intern_data_t:

Type	Field	Possible Values	Description
func_ptr_t	pfnFrt0PeakCallback	-	pointer to FRT0 peak detection interrupt callback function.
func_ptr_t	pfnFrt1PeakCallback	-	pointer to FRT1 peak detection interrupt callback function.
func_ptr_t	pfnFrt2PeakCallback	-	pointer to FRT2 peak detection interrupt callback function.
func_ptr_t	pfnFrt0ZeroCallback	-	pointer to FRT0 zero detection interrupt callback function.
func_ptr_t	pfnFrt1ZeroCallback	-	pointer to FRT1 zero detection interrupt callback function.
func_ptr_t	pfnFrt2ZeroCallback	-	pointer to FRT2 zero detection interrupt callback function.

A FRT data type instance uses the following configuration structure of the type of stc_mft_frt_instance_data_t:

Type	Field	Possible Values	Description
volatile stc_mftn_frt_t*	pstcInstance	-	pointer to registers of an instance.
stc_mft_frt_intern_data_t	stcInternData	-	module internal data of instance.

7.20.1.2 MFT FRT API

7.20.1.3 Mft_Frt_Init ()

This function initializes FRT module.

Prototype	
`en_result_t Mft_Frt_Init(volatile stc_mftn_frt_t *pstcMft, uint8_t u8Ch, stc_mft_frt_config_t* pstcFrtConfig);`	
Parameter Name	**Description**
[in] pstcMft	Pointer to a MFT instance.
[in] u8Ch	Channel of Free run timer.
[in] pstcFrtConfig	Pointer to a FRT configure.
Return Values	**Description**
Ok	Initialization successful done
ErrorInvalidParameter	pstcMft == NULL pstcFrtConfig parameter invalid Other invalid configuration setting(s)

7.20.1.4 Mft_Frt_SetMaskZeroTimes ()

This function sets FRT mask zero times.

Prototype	
`en_result_t Mft_Frt_SetMaskZeroTimes(volatile stc_mftn_frt_t*pstcMft,uint8_t u8Ch, uint8_t u8Times);`	
Parameter Name	**Description**
[in] pstcMft	Pointer to a MFT instance.
[in] u8Ch	Channel of Free run timer.
[in] u8Times	Mask times.
Return Values	**Description**
Ok	Set mask done.
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_FRT_MAX_CH u8Times > 15

7.20.1.5 Mft_Frt_GetMaskZeroTimes ()

This function gets FRT mask zero times.

Prototype	
`uint8_t Mft_Frt_GetMaskZeroTimes(volatile stc_mftn_frt_t*pstcMft, uint8_t u8Ch);`	
Parameter Name	**Description**
[in] pstcMft	Pointer to a MFT instance.
[in] u8Ch	channel of Free run timer.
Return Values	**Description**
value	Read register done.
0	pstcMft == NULL u8Ch >= MFT_FRT_MAX_CH

7.20.1.6 Mft_Frt_SetMaskPeakTimes ()

This function sets mask peak times.

Prototype	
`en_result_t Mft_Frt_SetMaskPeakTimes(volatile stc_mftn_frt_t*pstcMft,uint8_t u8Ch, uint8_t u8Times);`	
Parameter Name	**Description**
[in] pstcMft	Pointer to a MFT instance.
[in] u8Ch	Channel of Free run timer.
[in] u8Times	mask times
Return Values	**Description**
Ok	Set value done.
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_FRT_MAX_CH u8Times > 15

7.20.1.7 Mft_Frt_GetMaskPeakTimes ()

This function gets mask peak times.

Prototype	
`uint8_t Mft_Frt_GetMaskPeakTimes(volatile stc_mftn_frt_t*pstcMft, uint8_t u8Ch);`	
Parameter Name	**Description**
[in] pstcMft	Pointer to a MFT instance.
[in] u8Ch	Channel of Free run timer.
Return Values	**Description**
value	Read register done.
0	pstcMft == NULL u8Ch >= MFT_FRT_MAX_CH

7.20.1.8 Mft_Frt_Start ()

This function sets FRT start.

Prototype	
en_result_t Mft_Frt_Start(volatile stc_mftn_frt_t*pstcMft, uint8_t u8Ch);	
Parameter Name	**Description**
[in] pstcMft	Pointer to a MFT instance.
[in] u8Ch	Channel of Free run timer.
Return Values	**Description**
Ok	FRT start successfully.
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_FRT_MAX_CH

7.20.1.9 Mft_Frt_Stop ()

This function sets FRT stop.

Prototype	
en_result_t Mft_Frt_Stop(volatile stc_mftn_frt_t*pstcMft, uint8_t u8Ch);	
Parameter Name	**Description**
[in] pstcMft	Pointer to a MFT instance.
[in] u8Ch	Channel of Free run timer.
Return Values	**Description**
Ok	FRT stop successfully.
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_FRT_MAX_CH

7.20.1.10 Mft_Frt_EnableInt ()

This function enables FRT interrupt.

Prototype	
en_result_t Mft_Frt_EnableInt(volatile stc_mftn_frt_t*pstcMft, uint8_t u8Ch, stc_frt_int_sel_t* pstcIntSel, stc_frt_int_cb_t* pstcFrtIntCallback);	
Parameter Name	Description
[in] pstcMft	**Pointer to a MFT instance.**
[in] u8Ch	channel of Free run timer.
[in] pstcIntSel	Pointer to the type of interrupt.
[in] pstcFrtIntCallback	Pointer to the callback function of FRT.
Return Values	**Description**
Ok	Enable FRT interrupt successfully.
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_FRT_MAX_CH pstcIntSel is invalid

7.20.1.11 Mft_Frt_DisableInt ()

This function disables FRT interrupt.

Prototype	
en_result_t Mft_Frt_DisableInt(volatile stc_mftn_frt_t*pstcMft,uint8_t u8Ch, stc_frt_int_sel_t* pstcIntSel);	
Parameter Name	**Description**
[in] pstcMft	Pointer to a MFT instance.
[in] u8Ch	Channel of Free run timer.
[in] pstcIntSel	Pointer to the type of interrupt.
Return Values	**Description**
Ok	Disable FRT interrupt successfully.
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_FRT_MAX_CH pstcIntSel is invalid

7.20.1.12 Mft_Frt_GetIntFlag ()

This function gets FRT interrupt flag.

Prototype	
en_int_flag_t Mft_Frt_GetIntFlag(volatile stc_mftn_frt_t*pstcMft, uint8_t u8Ch, en_mft_frt_int_t enIntType);	
Parameter Name	**Description**
[in] pstcMft	Pointer to a MFT instance.
[in] u8Ch	Channel of Free run timer.
[in] enIntType	Type of interrupt.
Return Values	**Description**
value	Interrupt flag according to input type.

7.20.1.13 Mft_Frt_ClrIntFlag ()

This function clears FRT interrupt flag.

Prototype	
en_result_t Mft_Frt_ClrIntFlag(volatile stc_mftn_frt_t*pstcMft, uint8_t u8Ch, en_mft_frt_int_t enIntType);	
Parameter Name	**Description**
[in] pstcMft	Pointer to a MFT instance.
[in] u8Ch	Channel of Free run timer.
[in] enIntType	Type of interrupt.
Return Values	**Description**
Ok	Clear Flag successfully.
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_FRT_MAX_CH

7.20.1.14 Mft_Frt_SetCountCycle ()

This function sets FRT cycle value.

Prototype	
en_result_t Mft_Frt_SetCountCycle(volatile stc_mftn_frt_t*pstcMft, uint8_t u8Ch, uint16_t u16Cycle);	
Parameter Name	**Description**
[in] pstcMft	Pointer to a MFT instance.
[in] u8Ch	Channel of Free run timer.
[in] u16Cycle	Cycle value.
Return Values	**Description**
Ok	Set value successfully.
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_FRT_MAX_CH

7.20.1.15 Mft_Frt_SetCountVal ()

This function set FRT count value.

Prototype	
`en_result_t Mft_Frt_SetCountVal(volatile stc_mftn_frt_t*pstcMft, uint8_t u8Ch,uint16_t u16Count);`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to a MFT instance.
`[in] u8Ch`	Channel of Free run timer.
`[in] u16Count`	Count value.
Return Values	**Description**
`Ok`	Set value successfully.
`ErrorInvalidParameter`	`pstcMft == NULL` `u8Ch >= MFT_FRT_MAX_CH`

7.20.1.16 Mft_Frt_GetCurCount ()

This function gets FRT current count.

Prototype	
`uint16_t Mft_Frt_GetCurCount(volatile stc_mftn_frt_t*pstcMft, uint8_t u8Ch);`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to a MFT instance.
`[in] u8Ch`	Channel of Free run timer.
Return Values	**Description**
`value`	FRT current count value.
`ErrorInvalidParameter`	`pstcMft == NULL` `u8Ch >= MFT_FRT_MAX_CH`

7.20.1.17 Mft_Frt_IrqHandler ()

This function does FRT interrupt handler sub function.

Prototype	
`void Mft_Frt_IrqHandler( volatile stc_mftn_frt_t*pstcMft,stc_mft_frt_intern_data_t* pstcMftFrtInternData)`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to a MFT instance.
`[in] pstcMftFrtInternData`	Pointer to a FRT callback function.
Return Values	**Description**
`-`	

7.20.2 OCU (Output Compare Unit)

Type Definition	–

Configuration Types	`stc_mft_ocu_config_t`
	`stc_even_compare_config_t`
	`stc_odd_compare_config_t`
	`stc_mft_ocu_intern_data_t`
	`stc_mft_ocu_instance_data_t`

Address Operator	–

Before using OCU, a FRT used to connect with applying OCU must be initialed.

Mft_Ocu_Init() must be used for configuration of a Output Comapre Unit (OCU) channel with a structure of the type stc_mft_ocu_config_t. A FRT should be connected with applying OCU with this function.

Mft_Ocu_WriteOcse() is used to set the compare function/mode. For the meaning of each bit in OCSE register, please refer to MFT chapter of FM4 peripheral manual.

With Mft_Ocu_WriteOccp() the OCU compare value is set to the value given in the parameter Mft_Ocu_WriteOccp#u16Cycle. Whether the compare value is modified directly depends on buffer function.

An OCU interrupt can be enabled by the function Mft_Ocu_EnableInt(). This function can set callback function for each channel too.

After above setting, calling Mft_Ocu_EnableOperation() will start OCU.

With interrupt mode, the interrupt occurs when FRT count match OCU compare value, the interrupt flag will be cleared and run into user interrupt callback function.

With polling mode, user can use Mft_Ocu_GetIntFlag() to check if the interrupt occurs, and clear the interrupt flag by Mft_Ocu_ClrIntFlag().

When stopping the OCU, use Mft_Ocu_DisableOperation() to disable OCU and Mft_Ocu_DisableInt() to disable OCU interrupt.

7.20.2.1 Configuration Structure

A MFT OCU configure instance uses the following configuration structure of the type of stc_mft_ocu_config_t:

Type	Field	Possible Values	Description
en_mft_ocu_frt_t	enFrtConnect	Frt0ToOcu Frt1ToOcu Frt2ToOcu OcuFrtToExt	select the FRT to be connected to OCU connect FRT0 to OCU connect FRT1 to OCU connect FRT2 to OCU connect FRT of an external MFT
boolean_t	bFm4	TRUE FALSE	select FM4 mode or FM3-compatible mode FM4 mode FM3-compatible mode
boolean_t	bCmod	TRUE FALSE	FM3 compatible mode In FM4 mode, ignore this. See “4.2 HWM Timer Part” for details
uint8_t	u8Mod	Ch0 .. Ch5	select OCU ch0~ch5 in FM3 compatible
en_ocu_occp_buf_t	enOccpBufMode	OccpBufDisable OccpBufTrsfByFrtZero OccpBufTrsfByFrtPeak OccpBufTrsfByFrtZeroPeak	buffer register function of OCCP disable the buffer function buffer transfer when counter value is 0x0000 buffer transfer when counter value is TCCP buffer transfer when the value is both 0 and TCCP
en_ocu_ocse_buf_t	enOcseBufMode	OcseBufDisable OcseBufTrsfByFrtZero OcseBufTrsfByFrtPeak OcseBufTrsfByFrtZeroPeak	buffer register function of OCSE disable the buffer function buffer transfer when counter value is 0x0000 buffer transfer when counter value is TCCP buffer transfer when the value is both 0 and TCCP
en_ocu_rt_out_state_t	enStatePin	RtLowLevel RtHighLevel	RT output level state output low level to RT pin output high level to RT pin

A compare configuration of odd OCU channel instance uses the following configuration structure of the type of stc_odd_compare_config_t:

Type	Field	Possible Values	Description
en_rt_odd_status_t	enFrtZeroOddMatchEvenMatchOddChRtStatus	RtOutputHold RtOutputHigh RtOutputLow RtOutputReverse	Odd channel's RT output status when even channel and odd channel match occurs at the condition of FRT count=0x0000 RT output hold RT output high RT output low RT output reverse
en_rt_odd_status_t	enFrtZeroOddMatchEvenNotMatchOddChRtStatus	Same above	Odd channel's RT output status when even channel not match and odd channel match occurs at the condition of FRT count=0x0000
en_rt_odd_status_t	enFrtZeroOddNotMatchEvenMatchOddChRtStatus	Same above	Odd channel's RT output status when even channel match and odd channel not match occurs at the condition of FRT count=0x0000
en_rt_odd_status_t	enFrtZeroOddNotMatchEvenNotMatchOddChRtStatus	Same above	Odd channel's RT output status when even channel not match and odd channel not match occurs at the condition of FRT count=0x0000
en_rt_odd_status_t	enFrtUpCntOddMatchEvenMatchOddChRtStatus	Same above	Odd channel's RT output status when even channel and odd channel match occurs at the condition of FRT is counting up
en_rt_odd_status_t	enFrtUpCntOddMatchEvenNotMatchOddChRtStatus	Same above	Odd channel's RT output status when even channel not match and odd channel match occurs at the condition of FRT is counting up
en_rt_odd_status_t	enFrtUpCntOddNotMatchEvenMatchOddChRtStatus	Same above	Odd channel's RT output status when even channel match and odd channel not match occurs at the condition of FRT is counting up
en_rt_odd_status_t	enFrtPeakOddMatchEvenMatchOddChRtStatus	Same above	Odd channel's RT output status when even channel and odd channel match occurs at the condition of FRT count=Peak
en_rt_odd_status_t	enFrtPeakOddMatchEvenNotMatchOddChRtStatus	Same above	Odd channel's RT output status when even channel not match and odd channel match occurs at the condition of FRT count=Peak
en_rt_odd_status_t	enFrtPeakOddNotMatchEvenMatchOddChRtStatus	Same above	Odd channel's RT output status when even channel match and odd channel not match occurs at the condition of FRT count=Peak
en_rt_odd_status_t	enFrtPeakOddNotMatchEvenNotMatchOddChRtStatus	Same above	Odd channel's RT output status when even channel not match and odd channel not match

Type	Field	Possible Values	Description
			occurs at the condition of FRT count=Peak
en_rt_odd_status_t	enFrtDownOddMatchEvenMatchOddChRtStatus	Same above	Odd channel's RT output status when even channel and odd channel match occurs at the condition of FRT is counting down
en_rt_odd_status_t	enFrtDownOddMatchEvenNotMatchOddChRtStatus	Same above	Odd channel's RT output status when even channel not match and odd channel match occurs at the condition of FRT is counting down
en_rt_odd_status_t	enFrtDownOddNotMatchEvenMatchOddChRtStatus	Same above	Odd channel's RT output status when even channel match and odd channel not match occurs at the condition of FRT is coutning down
en_iop_flag_odd_t	enIopFlagWhenFrtZeroOddMatch	IopFlagHold IopFlagSet	Odd channel OCU's IOP flag status when Odd channel match occurs at the condition of FRT count=0x0000
en_iop_flag_odd_t	enIopFlagWhenFrtUpCntOddMatch	Same above	Odd channel OCU's IOP flag status when Odd channel match occurs at the condition of FRT is counting up
en_iop_flag_odd_t	enIopFlagWhenFrtPeakOddMatch	Same above	Odd channel OCU's IOP flag status when Odd channel match occurs at the condition of FRT count=Peak
en_iop_flag_odd_t	enIopFlagWhenFrtDownCntOddMatch	Same above	Odd channel OCU's IOP flag status when Odd channel match occurs at the condition of FRT is counting down

An OCU internal data instance uses the following configuration structure of the type of stc_mft_ocu_intern_data_t:

Type	Field	Possible Values	Description
func_ptr_t	pfnOcu0Callback	-	Callback function pointer of OCU0 interrupt
func_ptr_t	pfnOcu1Callback	-	Callback function pointer of OCU1 interrupt
func_ptr_t	pfnOcu2Callback	-	Callback function pointer of OCU2 interrupt
func_ptr_t	pfnOcu3Callback	-	Callback function pointer of OCU3 interrupt
func_ptr_t	pfnOcu4Callback	-	Callback function pointer of OCU4 interrupt
func_ptr_t	pfnOcu5Callback	-	Callback function pointer of OCU5 interrupt

An OCU instance data type instance uses the following configuration structure of the type of stc_mft_ocu_intern_data_t:

Type	Field	Possible Values	Description
`volatile stc_mftn_ocu_t*`	`pstcInstance`	-	Pointer to registers of an instance.
`stc_mft_ocu_intern_data_t`	`stcInternData`	-	Module internal data of instance.

7.20.2.2 MFT OCU API

7.20.2.3 Mft_Ocu_Init ()

This function initializes OCU module.

Prototype	
`en_result_t Mft_Ocu_Init( volatile stc_mftn_ocu_t*pstcMft, uint8_t u8Ch, stc_mft_ocu_config_t* pstcOcuConfig);`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance.
`[in] u8Ch`	MFT OCU channel.
`[in] pstcOcuConfig`	Pointer to a configure of OCU.
Return Values	**Description**
`Ok`	Initialize done successfully.

`ErrorInvalidParameter`	`pstcMft == NULL`
	`u8Ch >= MFT_OCU_MAXCH`
	`pstcOcuConfig == NULL`
	`pstcOcuConfig param invalid`

7.20.2.4 Mft_Ocu_SetEvenChCompareMode ()

This function compares congifuration of even OCU channel.

Prototype	
`en_result_t Mft_Ocu_SetEvenChCompareMode(volatile stc_mftn_ocu_t*pstcMft,uint8_t EvenCh, stc_even_compare_config_t* pstcConfig);`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance.
`[in] EvenCh`	Even channel of OCU.
`[in] pstcConfig`	Pointer to structure of compare mode.
Return Values	**Description**
`Ok`	Compare mode set done.

`ErrorInvalidParameter`	`pstcMft == NULL`
	`EvenCh%2 != 0`
	pstcConfig param invalid.

7.20.2.5 Mft_Ocu_SetOddChCompareMode ()

This function compares congifuration of odd OCU channel.

Prototype	
en_result_t Mft_Ocu_SetOddChCompareMode (volatile stc_mftn_ocu_t*pstcMft,uint8_t OddCh, stc_odd_compare_config_t* pstcConfig);	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] OddCh	Odd channel of OCU.
[in] pstcConfig	Pointer to structure of compare mode.
Return Values	**Description**
Ok	Compare mode set done
ErrorInvalidParameter	pstcMft == NULL OddCh%2 != 0 pstcConfig param invalid.

7.20.2.6 Mft_Ocu_EnableOperation ()

This function enables OCU operation.

Prototype	
en_result_t Mft_Ocu_EnableOperation(volatile stc_mftn_ocu_t*pstcMft, uint8_t u8Ch);	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8Ch	Channel of OCU.
Return Values	**Description**
Ok	Enable operation done.
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_OCU_MAXCH

7.20.2.7 Mft_Ocu_DisableOperation ()

This function disables OCU operation.

Prototype	
en_result_t Mft_Ocu_DisableOperation(volatile stc_mftn_ocu_t*pstcMft, uint8_t u8Ch);	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8Ch	Channel of OCU.
Return Values	**Description**
Ok	Disable operation done.
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_OCU_MAXCH

7.20.2.8 Mft_Ocu_EnableInt ()

This function enables OCU interrupt.

Prototype	
en_result_t Mft_Ocu_EnableInt(volatile stc_mftn_ocu_t*pstcMft,uint8_t u8Ch, func_ptr_t pfnCallback);	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8Ch	Channel of OCU.
[in] pfnCallback	Interrupt callback function.
Return Values	**Description**
Ok	Enable OCU interrupt done
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_OCU_MAXCH

7.20.2.9 *Mft_Ocu_DisableInt ()*

This function disables OCU interrupt.

Prototype	
en_result_t Mft_Ocu_DisableInt(volatile stc_mftn_ocu_t*pstcMft,uint8_t u8Ch);	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8Ch	Channel of OCU.
Return Values	**Description**
Ok	Disable OCU interrupt done.
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_OCU_MAXCH

7.20.2.10 *Mft_Ocu_GetIntFlag ()*

This function gets OCU interrupt flag

Prototype	
en_int_flag_t Mft_Ocu_GetIntFlag(volatile stc_mftn_ocu_t*pstcMft, uint8_t u8Ch);	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8Ch	Channel of OCU.
Return Values	**Description**
value	Interrupt flag value.

7.20.2.11 Mft_Ocu_ClrIntFlag ()

This function clears OCU interrupt flag.

Prototype	
`en_result_t Mft_Ocu_ClrIntFlag(volatile stc_mftn_ocu_t*pstcMft, uint8_t u8Ch);`	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8Ch	Channel of OCU.
Return Values	**Description**
Ok	Clear OCU interrupt flag done.
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_OCU_MAXCH

7.20.2.12 Mft_Ocu_GetRtPinLevel ()

This function gets RT pin level of OCU.

Prototype	
`en_ocu_rt_out_state_t Mft_Ocu_GetRtPinLevel(volatile stc_mftn_ocu_t*pstcMft, uint8_t u8Ch);`	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8Ch	Channel of OCU.
Return Values	**Description**
value	RtLowlevel or RtHighlevel

7.20.2.13 Mft_Ocu_WriteOccp ()

This function writes OCCP register.

Prototype	
`en_result_t Mft_Ocu_WriteOccp(volatile stc_mftn_ocu_t*pstcMft,uint8_t u8Ch, uint16_t u16Occp);`	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8Ch	Channel of OCU.
[in] u16Occp	Value of occp to write.
Return Values	**Description**
Ok	Write OCCP register done
ErrorInvalidParameter	pstcMft == NULL u8Ch >= MFT_OCU_MAXCH

7.20.2.14 *Mft_Ocu_ReadOccp ()*

This function gets OCCP register value.

Prototype	
`uint16_t Mft_Ocu_ReadOccp(volatile stc_mftn_ocu_t*pstcMft, uint8_t u8Ch);`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance.
`[in] u8Ch`	Channel of OCU.
Return Values	**Description**
`value`	Return OCCP value.
`ErrorInvalidParameter`	`pstcMft == NULL` `u8Ch >= MFT_OCU_MAXCH`

7.20.2.15 *Mft_Ocu_IrqHandler ()*

This function is OCU module interrupt handler.

Prototype	
`void Mft_Ocu_IrqHandler( volatile stc_mftn_ocu_t* pstcMft, stc_mft_ocu_intern_data_t* pstcMftOcuInternData);`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance.
`[in] pstcMftOcuInternData`	Callbacks function of OCU.
Return Values	**Description**
-	

7.20.3 WFG (Waveform Genarator Unit)

Type Definition	-

Configuration Types	`stc_wfg_ctrl_bits`
	`stc_mft_wfg_intern_data_t`
	`stc_mft_wfg_instance_data_t`

Address Operator	-

Operation mode of WFG module

The WFG can be configured to following mode:

- Through mode
- RT-PPG mode
- Timer-PPG mode
- RT-dead timer mode
- PPG-dead timer mode

How to operate WFG with Through mode (one of usages).

Before using WFG, the FRT used to connect with applying OCU must be initialed first and OCU must be initialed.

Set the control bits with Mft_Wfg_ConfigCtrlBits() with a structure of the type stc_wfg_ctrl_bits

- enGtenBits = b'00
- enPselBits = b'00
- enPgenBits = b'00
- enDmodBit = b'0

Configure WFG to Through mode with Mft_Wfg_ConfigMode()

Enable OCU operation and RT signal will output to RTO pin directly. Changing the value of control bits will influence RTO output.

how to operate WFG with RT-PPG mode (one of usages).

Before using WFG, the FRT used to connect with applying OCU must be initialized first and OCU must be initialed then, PPG should be initialized at the following.

Set the control bits with Mft_Wfg_ConfigCtrlBits() with a structure of the type stc_wfg_ctrl_bits

- enGtenBits = b'00
- enPselBits = b'00 (input PPG ch.0 to WFG0)
- enPgenBits = b'11
- enDmodBit = b'0

Configure WFG to RT-PPG mode with Mft_Wfg_ConfigMode()

Start PPG0 and enable OCU operation.

In this case RTO0 will outputs the logic AND signal of RT0 signal and PPG0, RTO1 will outputs the logic AND signal of RT1 signal and PPG0. Changing the value of control bits will influence RTO output.

How to operate WFG with Timer-PPG mode (one of usages).

Before using WFG, the FRT used to connect with applying OCU must be initialized first and OCU must be initialed then, PPG and WFG timer should be initialized at the following.

Set the control bits with Mft_Wfg_ConfigCtrlBits() with a structure of the type stc_wfg_ctrl_bits

- enGtenBits = b'00
- enPselBits = b'00 (input PPG ch.0 to WFG10)
- enPgenBits = b'11
- enDmodBit = b'0

Configure WFG to Timer-PPG mode with Mft_Wfg_ConfigMode()

Start PPG0, enable OCU operation and start WFG timer.

In this case RTO0 will outputs the logic AND signal of WFG timer flag 0 and PPG0, RTO1 will outputs the logic AND signal of WFG timer flag 1 and PPG0.

Changing the value of control bits will influence RTO output.

How to operate WFG with RT-dead timer mode (one of usages).

Before using WFG, the FRT used to connect with applying OCU must be initialized first and OCU must be initialed then, WFG timer should be initialized at the following.

Configure WFG to RT-dead timer mode with Mft_Wfg_ConfigMode()

Set the control bits with Mft_Wfg_ConfigCtrlBits() with a structure of the type stc_wfg_ctrl_bits

- enGtenBits = b'00
- enPselBits = b'00
- enPgenBits = b'00
- enDmodBit = b'0

Enable OCU operation and start WFG timer.

In this case, RT(1) will trigger the WFG timer to start, WFG will output the generated non-overlap singal. Changing the value of control bits will influence RTO output.

How to operate WFG with PPG-dead timer mode (one of usages).

Before using WFG, WFG timer should be initialized at the following. For how to initialize WFG timer, see the description at the following.

Configure WFG to PPG-dead timer mode with Mft_Wfg_ConfigMode()

Set the control bits with Mft_Wfg_ConfigCtrlBits() with a structure of the type stc_wfg_ctrl_bits

- enGtenBits = b'00
- enPselBits = b'00 (input PPG0 to WFG10)
- enPgenBits = b'00
- enDmodBit = b'0

Start WFG timer.

In this case, PPG0 will trigger the WFG timer to start, WFG will output the generated non-overlap singal. Changing the value of control bits will influence RTO output.

How to use the WFG timer

Mft_Wfg_InitTimerClock() must be used for configuration of a Free-Run timer (FRT) channel with a structure of the type en_wfg_timer_clock_t.

A WFG timer interrupt can be enabled by the function Mft_Wfg_EnableTimerInt(). This function can set callback function for each channel too.

With Mft_Wfg_WriteTimerCountCycle() the WFG timer cycle is set to the value given in the parameter Mft_Wfg_WriteTimerCountCycle#u16CycleA and Mft_Wfg_WriteTimerCountCycle#u16CycleB.

After above setting, calling Mft_Wfg_SetTimerCycle() will start WFG timer.

With Mft_Wfg_GetTimerCurCycle() the current WFG timer count can be read when WFG timer is running.

With interrupt mode, when the interrupt occurs, the interrupt flag will be cleared and run into user interrupt callback function.

With polling mode, user can use Mft_Wfg_GetTimerIntFlag() to check if the interrupt occurs, and clear the interrupt flag by Mft_Wfg_ClrTimerIntFlag().

When stopping the WFG, use Mft_Wfg_DisableTimerInt() to disable WFG timer and Mft_Wfg_DisableTimerInt() to disable WFG timer interrupt.

7.20.3.1 Configuration Structure

A WFG control configure instance uses the following configuration structure of the type of stc_wfg_ctrl_bits:

Type	Field	Possible Values	Description

`en_gten_bits_t`	`enGtenBits`	`GtenBits00B`
		`GtenBits01B`
		`GtenBits10B`
		`GtenBits11B`
			select the output condition:
			(see HWM 4.3 Description of WFG Operation)

`en_psel_bits_t`	`enPselBits`	`PselBits00B`
		`PselBits01B`
		`PselBits10B`
		`PselBits11B`
			Select the PPG timer unit:
			(see HWM 4.3 Description of WFG Operation)

`en_pgen_bits_t`	`enPgenBits`	`PgenBits00B`
		`PgenBits01B`
		`PgenBits10B`
		`PgenBits11B`
			how to reflect the CH_PPG signal:
			(see HWM 4.3 Description of WFG Operation)

`en_dmod_bit_t`	`enDmodBit`	`DmodBit0B` `DmodBit0B`	polarity for RTO0 and RTO1 signal output: output RTO1 and RTO0 signals without changing the level output both RTO1 and RTO0 signals reversed

A WFG internal data instance uses the following configuration structure of the type of stc_mft_wfg_intern_data_t:

Type	Field	Possible Values	Description
func_ptr_t	pfnWfg10TimerCallback	-	Callback function pointer of WFG10 timer interrupt callback.
func_ptr_t	pfnWfg32TimerCallback	-	Callback function pointer of WFG32 timer interrupt callback.
func_ptr_t	pfnWfg54TimerCallback	-	Callback function pointer of WFG54 timer interrupt callback.
func_ptr_t	pfnWfgAnalogFilterCallback	-	Callback function pointer of analog filter interrupt callback.
func_ptr_t	pfnWfgDigtalFilterCallback	-	Callback function pointer of analog filter interrupt callback.

A WFG data instance uses the following configuration structure of the type of stc_mft_wfg_instance_data_t:

Type	Field	Possible Values	Description
volatile stc_mftn_wfg_t*	pstcInstance	-	Pointer to registers of an instance.
stc_mft_wfg_intern_data_t	stcInternData	See above	module internal data

7.20.3.2 MFT WFG API

7.20.3.3 Mft_Wfg_ConfigMode ()

This function configures WFG mode.

Prototype	
`en_result_t Mft_Wfg_ConfigMode( volatile stc_mftn_wfg_t*pstcMft, uint8_t u8CoupleCh, en_mft_wfg_mode_t enMode);`	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8CoupleCh	Channel of WFG couple.
[in] enMode	WFG mode.
Return Values	**Description**
Ok	Configure WFG mode successfully.
ErrorInvalidParameter	pstcMft == NULL u8CoupleCh >= MFT_WFG_MAXCH

7.20.3.4 Mft_Wfg_ConfigCtrlBits ()

This function configures WFG control bit.

Prototype	
en_result_t Mft_Wfg_ConfigCtrlBits(volatile stc_mftn_wfg_t*pstcMft,uint8_t u8CoupleCh, stc_wfg_ctrl_bits* pstcCtrlBits);	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8CoupleCh	Channel of WFG couple.
[in] pstcCtrlBits	Pointer to control bit instance.
Return Values	**Description**
Ok	Configure control bits successfully.
ErrorInvalidParameter	pstcMft == NULL u8CoupleCh >= MFT_WFG_MAXCH

7.20.3.5 Mft_Wfg_InitTimerClock ()

This function initializes timer clock.

Prototype	
en_result_t Mft_Wfg_InitTimerClock(volatile stc_mftn_wfg_t*pstcMft,uint8_t u8CoupleCh, en_wfg_timer_clock_t enClk);	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8CoupleCh	Channel of WFG couple.
[in] enClk	Count clock cycle to PCLK multiplied.
Return Values	**Description**
Ok	Init timer clock successfully.
ErrorInvalidParameter	pstcMft == NULL u8CoupleCh >= MFT_WFG_MAXCH

7.20.3.6 Mft_Wfg_EnableTimerInt ()

This function enables WFG timer interrupt.

Prototype	
en_result_t Mft_Wfg_EnableTimerInt(volatile stc_mftn_wfg_t*pstcMft, uint8_t u8CoupleCh, func_ptr_t pfnCallback);	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8CoupleCh	Channel of WFG couple.
[in] pfnCallback	Pointer to interrupt callback function.
Return Values	**Description**
Ok	Enable interrupt successfully.
ErrorInvalidParameter	pstcMft == NULL u8CoupleCh >= MFT_WFG_MAXCH

7.20.3.7 Mft_Wfg_DisableTimerInt ()

This function disables WFG timer interrupt.

Prototype	
en_result_t Mft_Wfg_DisableTimerInt(volatile stc_mftn_wfg_t*pstcMft, uint8_t u8CoupleCh);	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8CoupleCh	Channel of WFG couple.
Return Values	**Description**
Ok	Disable interrupt successfully.
ErrorInvalidParameter	pstcMft == NULL u8CoupleCh >= MFT_WFG_MAXCH

7.20.3.8 Mft_Wfg_StartTimer ()

This function starts WFG timer.

Prototype	
`en_result_t Mft_Wfg_StartTimer(volatile stc_mftn_wfg_t*pstcMft, uint8_t u8CoupleCh);`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance.
`[in] u8CoupleCh`	Channel of WFG couple.
Return Values	**Description**
`Ok`	Start timer successfully.
`ErrorInvalidParameter`	`pstcMft == NULL` `u8CoupleCh >= MFT_WFG_MAXCH`

7.20.3.9 Mft_Wfg_StopTimer ()

This function stops WFG timer.

Prototype	
`en_result_t Mft_Wfg_StopTimer(volatile stc_mftn_wfg_t*pstcMft, uint8_t u8CoupleCh);`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance.
`[in] u8CoupleCh`	Channel of WFG couple.
Return Values	**Description**
`Ok`	Stops timer successfully.
`ErrorInvalidParameter`	`pstcMft == NULL` `u8CoupleCh >= MFT_WFG_MAXCH`

7.20.3.10 Mft_Wfg_GetTimerIntFlag ()

This function gets WFG timer interrupt flag.

Prototype	
`en_int_flag_t Mft_Wfg_GetTimerIntFlag( volatile stc_mftn_wfg_t*pstcMft,uint8_t u8CoupleCh);`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance.
`[in] u8CoupleCh`	Channel of WFG couple.
Return Values	**Description**
`PdlSet`	Interrupt occurs.
`PdlClr`	Interrupt not occurs.

7.20.3.11 Mft_Wfg_ClrTimerIntFlag ()

This function clears WFG timer interrupt flag.

Prototype	
`en_result_t Mft_Wfg_ClrTimerIntFlag( volatile stc_mftn_wfg_t*pstcMft, uint8_t u8CoupleCh);`	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8CoupleCh	Channel of WFG couple.
Return Values	**Description**
Ok	Clear interrupt flag successfully.
ErrorInvalidParameter	pstcMft == NULL u8CoupleCh >= MFT_WFG_MAXCH

7.20.3.12 Mft_Wfg_WriteTimerCountCycle ()

This function writes timer count cycle.

Prototype	
`en_result_t Mft_Wfg_WriteTimerCountCycle( volatile stc_mftn_wfg_t*pstcMft, uint8_t u8CoupleCh, uint16_t u16CycleA, uint16_t u16CycleB);`	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8CoupleCh	Channel of WFG couple.
[in] u16CycleA	WFTA value.
[in] u16CycleB	WFTB value.
Return Values	**Description**
Ok	Write value done.
ErrorInvalidParameter	pstcMft == NULL u8CoupleCh >= MFT_WFG_MAXCH

7.20.3.13 Mft_Wfg_SetTimerCycle ()

This function sets WFG pulse counter.

Prototype	
`en_result_t Mft_Wfg_SetTimerCycle( volatile stc_mftn_wfg_t*pstcMft, uint8_t u8CoupleCh, uint16_t u16Count);`	
Parameter Name	**Description**
[in] pstcMft	Pointer to MFT instance.
[in] u8CoupleCh	Channel of WFG couple.
[in] u16Count	WFG pulse counter value.
Return Values	**Description**
Ok	Set WFG pulse counter done.

7.20.3.14 Mft_Wfg_GetTimerCurCycle ()

This function gets current pulse counter.

Prototype	
`uint16_t Mft_Wfg_GetTimerCurCycle( volatile stc_mftn_wfg_t*pstcMft, uint8_t u8CoupleCh);`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance.
`[in] u8CoupleCh`	Channel of WFG couple.
Return Values	**Description**
`value`	Value of pulse counter.

7.20.3.15 Mft_Wfg_IrqHandler ()

This function is WFG interrupt handler sub function.

Prototype	
`void Mft_Wfg_IrqHandler( volatile  stc_mftn_wfg_t*pstcMft, stc_mft_wfg_intern_data_t* pstcMftWfgInternData);`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance.
`[in] pstcMftWfgInternData`	Pointer to WFG callback function.
Return Values	**Description**
–	

7.20.4 ICU (Input Capture Unit)

Type Definition	–

Configuration Types	`stc_mft_icu_config_t`
	`stc_mft_icu_intern_data_t`
	`stc_mft_icu_instance_data_t`

Address Operator	–

Before using ICU, a FRT used to connect with applying ICU must be initialed. With Mft_Icu_SelFrt(), a FRT can be connected with ICU.

A ICU interrupt can be enabled by the function Mft_Icu_EnableInt(). This function can set callback function for each channel too.

After above setting, calling Mft_Icu_ConfigDetectMode() will select a detection mode and start ICU operation at the same time. Following detection mode cna be select:

- Disable
- Rising edge detection mode
- Falling edge detection mode
- Both edge detection mode

With interrupt mode, the interrupt occurs when valid edge is detected, the interrupt flag will be cleared and run into user interrupt callback function. In the callback fucntion, the capture value can be read with Mft_Icu_GetCaptureData().

With polling mode, user can use Mft_Icu_GetIntFlag() to check if the interrupt occurs, and clear the interrupt flag by Mft_Icu_ClrIntFlag().

when interrupt is detected, the capture value can be read with Mft_Icu_GetCaptureData().

When the valid edge is detected, Mft_Icu_GetLastEdge() can get the edge information of last valid edge.

When stopping the ICU, use Mft_Icu_ConfigDetectMode() to disable ICU and Mft_Icu_DisableInt() to disable ICU interrupt.

7.20.4.1 Configuration Structure

A MFT ICU configuration instance uses the following configuration structure of the type of stc_mft_icu_config_t:

Type	Field	Possible Values	Description
en_mft_icu_frt_t	enFrt	Frt0ToIcu Frt1ToIcu Frt2ToIcu IcuFrtToExt0 IcuFrtToExt1	Frt channel Frt channel 0 to Icu Frt channel 1 to Icu Frt channel 2 to Icu Extern Frt channel 1 to Icu Extern Frt channel 1 to Icu
en_mft_icu_mode_t	enMode	IcuDisable IcuRisingDetect IcuFallingDetect IcuBothDetect	Icu detect mode disable Icu edge detection detect Icu rising edge detect Icu falling edge detect Icu rising/falling edge
en_mft_icu_edge_t	enEdge	IcuFallingEdge IcuRisingEdge	Icu edge select Icu falling edge select Icu rising edge

An ICU internal data instance uses the following configuration structure of the type of stc_mft_icu_intern_data_t:

Type	Field	Possible Values	Description
func_ptr_t	pfnIcu0Callback	-	Pointer to an ICU0 interrupt Callback function.
func_ptr_t	pfnIcu1Callback	-	Pointer to an ICU1 interrupt Callback function.
func_ptr_t	pfnIcu2Callback	-	Pointer to an ICU2 interrupt Callback function.
func_ptr_t	pfnIcu3Callback	-	Pointer to an ICU3 interrupt Callback function.

An Mft_icu data type instance uses the following configuration structure of the type of stc_mft_icu_instance_data_t:

Type	Field	Possible Values	Description
volatile stc_mftn_icu_t*	pstcInstance	-	Pointer to registers of an instance.
stc_mft_icu_intern_data_t	stcInternData	See detail above.	module internal data of instance..

7.20.4.2 MFT ICU API

7.20.4.3 Mft_Icu_SelFrt ()

This function selects FRTx channel to connect to ICUx.

Prototype	
en_result_t Mft_Frt_Init(volatile stc_mftn_frt_t *pstcMft, uint8_t u8Ch, stc_mft_frt_config_t* pstcFrtConfig);	
Parameter Name	**Description**
[in] pstcMftIcu	Pointer to a MFT instance.
[in] u8Ch	MFT ICU channel.
[in] enFrt	FRT channel number.
Return Values	**Description**
Ok	Set channel done.
ErrorInvalidParameter	pstcMft == NULL u8Ch > MFT_ICU_CHx_MAX enFrt > IcuFrtToExt1

7.20.4.4 Mft_Icu_ConfigDetectMode ()

This function configures ICU module detection mode.

Prototype	
en_result_t Mft_Icu_ConfigDetectMode(volatile stc_mftn_icu_t* pstcMftIcu, uint8_t u8Ch, en_mft_icu_mode_t enMode);	
Parameter Name	**Description**
[in] pstcMftIcu	Pointer to a MFT instance.
[in] u8Ch	MFT ICU channel.
[in] enMode	ICU detect mode.
Return Values	**Description**
Ok	Set detect mode done.
ErrorInvalidParameter	pstcMft == NULL u8Ch > MFT_ICU_CHx_MAX enMode > IcuBothDetect

7.20.4.5 Mft_Icu_EnableInt ()

This function enables MFT ICU interrupt.

Prototype	
en_result_t Mft_Icu_EnableInt(volatile stc_mftn_icu_t*pstcMftIcu, uint8_t u8Ch, func_ptr_t pfnCallback);	
Parameter Name	**Description**
[in] pstcMftIcu	Pointer to a MFT instance.
[in] u8Ch	MFT ICU channel.
[in] pfnCallback	Pointer to interrupt callback function.
Return Values	**Description**
Ok	Enable ICU interrupt done.
ErrorInvalidParameter	pstcMft == NULL u8Ch > MFT_ICU_CHx_MAX

7.20.4.6 *Mft_Icu_DisableInt ()*

This function disables MFT ICU interrupt and release callback function.

Prototype	
en_result_t Mft_Icu_DisableInt(volatile stc_mftn_icu_t*pstcMftIcu, uint8_t u8Ch);	
Parameter Name	**Description**
[in] pstcMftIcu	Pointer to a MFT instance.
[in] u8Ch	MFT ICU channel.
Return Values	**Description**
Ok	Disable ICU interrupt done.
ErrorInvalidParameter	pstcMft == NULL u8Ch > MFT_ICU_CHx_MAX

7.20.4.7 *Mft_Icu_GetIntFlag ()*

This function gets interrupt flag.

Prototype	
en_int_flag_t Mft_Icu_GetIntFlag(volatile stc_mftn_icu_t *pstcMftIcu, uint8_t u8Ch);	
Parameter Name	**Description**
[in] pstcMftIcu	Pointer to MFT instance.
[in] u8Ch	MFT ICU channel.
Return Values	**Description**
value	interrupt flag according to channel.

7.20.4.8 Mft_Icu_ClrIntFlag ()

This function clears interrupt flag.

Prototype	
`en_result_t Mft_Icu_ClrIntFlag( volatile stc_mftn_icu_t *pstcMftIcu, uint8_t u8Ch);`	
Parameter Name	**Description**
[in] pstcMftIcu	Pointer to MFT instance.
[in] u8Ch	MFT ICU channel.
Return Values	**Description**
Ok	Clear flag done.
ErrorInvalidParameter	pstcMft == NULL u8Ch > MFT_ICU_CHx_MAX

7.20.4.9 Mft_Icu_GetLastEdge ()

This function gets the latest captured edge type.

Prototype	
`en_mft_icu_edge_t Mft_Icu_GetLastEdge( volatile stc_mftn_icu_t *pstcMftIcu, uint8_t u8Ch);`	
Parameter Name	**Description**
[in] pstcMftIcu	Pointer to MFT instance.
[in] u8Ch	MFT ICU channel.
Return Values	**Description**
value	Return detected edge type.

7.20.4.10 Mft_Icu_GetCaptureData ()

This function reads captured data value.

Prototype	
`uint16_t Mft_Icu_GetCaptureData(volatile stc_mftn_icu_t *pstcMftIcu, uint8_t u8Ch);`	
Parameter Name	**Description**
[in] pstcMftIcu	Pointer to MFT instance.
[in] u8Ch	MFT ICU channel.
Return Values	**Description**
Value	Return captured data value.
ErrorInvalidParameter	pstcMft == NULL u8Ch > MFT_ICU_CHx_MAX

7.20.4.11 Mft_Icu_IrqHandler ()

This function implements ICU interrupt service routine.

Prototype	
`void Mft_Icu_IrqHandler( volatile stc_mftn_icu_t* pstcMftIcu, stc_mft_icu_intern_data_t* pstcMftIcuInternData);`	
Parameter Name	**Description**
`[in] pstcMftIcu`	Pointer to MFT instance.
`[in] pstcMftIcuInternData`	Pointer to intern data.
Return Values	**Description**
-	

7.20.5 ADCMP (ADC Start Compare Unit)

Type Definition	-

Configuration Types	`stc_mft_adcmp_config_t`
	`stc_mft_adcmp_func_t`
	`stc_mft_adcmp_fm3_config_t`

Address Operator	-

How to use ADCMP module

Two modes including normal mode and offset mode can be configured for ADCMP. This module should be used with ADC.

Before using ADCMP, a FRT used to connect with applying ADCMP must be initialed. In the offset mode, an OCU should also be used and initialed.

Mft_Adcmp_Init() must be used for configuration of a ADCMP channel with a structure of the type stc_mft_adcmp_config_t.

With Mft_Adcmp_WriteAcmp() the ADC start compare value is set to the value given in the parameter Mft_Adcmp_WriteAcmp#u16AcmpVal. Whether the compare value is modified directly depends on buffer function.

After above setting, calling Mft_Adcmp_EnableOperation() will start ADCMP. When stopping the OCU, use Mft_Adcmp_DisableOperation() to disable ADCMP.

In addition, the module is also compatible with FM3 usage.

How to use ADCMP module with FM3 compatibility function

Before using ADCMP, a FRT used to connect with applying ADCMP must be initialed.

Mft_Adcmp_Fm3_Init() must be used for configuration of a ADCMP channel with a structure of the type stc_mft_adcmp_fm3_config_t.

With Mft_Adcmp_Fm3_WriteAccp() the ADC start up compare value is set to the value given in the parameter Mft_Adcmp_WriteAcmp#u16AccpVal. With Mft_Adcmp_Fm3_ReadAccp() the ADC start up compare value is read.

With Mft_Adcmp_Fm3_WriteAccpdn() the ADC start down compare value is set to the value given in the parameter Mft_Adcmp_WriteAcmp#u16AccpVal. With Mft_Adcmp_Fm3_ReadAccpdn() the ADC start down compare value is read.

7.20.5.1 Configuration Structure

A Mft_adcmp configure parameters instance uses the following configuration structure of the type of stc_mft_adcmp_config_t:

Type	Field	Possible Values	Description
en_adcmp_frt_t	enFrt	Frt0ToAdcmp Frt1ToAdcmp Frt2ToAdcmp AdcmpFrtToExt0 AdcmpFrtToExt1	configure Adcmp Frt channel: Frt channel 0 Frt channel 1 Frt channel 2 Extern Frt channel 0 Extern Frt channel 1
en_adcmp_buf_t	enBuf	AdcmpBufDisable AdcmpBufFrtZero AdcmpBufFrtPeak AdcmpBufFrtZeroPeak	Adcmp buffer type disable Adcmp buffer function transfer buffer when counter value of Frt connected= 0x0000 transfer buffer when counter value of Frt connected= TCCP transfer buffer both when counter value of Frt connected= 0x0000 and TCCP
en_adcmp_trig_sel_t	enTrigSel	AdcmpTrigAdc0Scan AdcmpTrigAdc0Prio AdcmpTrigAdc1Scan AdcmpTrigAdc1Prio AdcmpTrigAdc2Scan AdcmpTrigAdc2Prio	configure Adcmp Trigger type AdcmpStartTrig0 AdcmpStartTrig1 AdcmpStartTrig2 AdcmpStartTrig3 AdcmpStartTrig4 AdcmpStartTrig5
en_adcmp_mode_t	enMode	AdcmpNormalMode AdcmpOffsetMode	configure Adcmp Running mode Normal mode Offset mode
en_adcmp_occp_sel_t	enOccpSel	AdcmpSelOccp0 AdcmpSelOccp1	select Adcmp Occp channel Occp0 channel Occp1 channel

A Mft_adcmp functions instance uses the following configuration structure of the type of stc_mft_adcmp_func_t:

Type	Field	Possible Values	Description
boolean_t	bDownEn	TRUE FALSE	Adcmp Down function Enable Disable
boolean_t	bPeakEn	TRUE FALSE	Adcmp Peak function Enable Disable
boolean_t	bUpEn	TRUE FALSE	Adcmp Up function Enable Disable
boolean_t	bZeroEn	TRUE FALSE	Adcmp Zero function Enable Disable

A Mft_adcmp compatible fm3 configure parameters instance uses the following configuration structure of the type of stc_mft_adcmp_fm3_config_t:

Type	Field	Possible Values	Description
en_adcmp_fm3_frt_t	enFrt	Frt1ToAdcmpFm3 Frt2ToAdcmpFm3	Adcmp Frt channel connect Frt channel 1 to Icu connect Frt channel 2 to Icu
en_adcmp_fm3_mode_t	enMode	AdcmpFm3AccpUpAccpDown AdcmpFm3AccpUp AdcmpFm3AccpDown AdcmpFm3AccpUpAccpdnDown	compatible Fm3 mode Accp Up and Down Accp Up Accp Down Accp up adn Accpdn Down
en_adcmp_buf_t	enBuf	AdcmpBufDisable AdcmpBufFrtZero AdcmpBufFrtPeak AdcmpBufFrtZeroPeak	Adcmp Buffer transfer type disable Adcmp buffer function transfer buffer when counter value of Frt connected= 0x0000 transfer buffer when counter value of Frt connected= TCCP transfer buffer both when counter value of Frt connected= 0x0000 and TCCP
en_adcmp_trig_sel_t	enTrigSel	AdcmpTrigAdc0Scan AdcmpTrigAdc0Prio AdcmpTrigAdc1Scan AdcmpTrigAdc1Prio AdcmpTrigAdc2Scan AdcmpTrigAdc2Prio	Select trig mode AdcmpStartTrig0 AdcmpStartTrig1 AdcmpStartTrig2

Type	Field	Possible Values	Description
			AdcmpStartTrig3 AdcmpStartTrig4 AdcmpStartTrig5

7.20.5.2 MFT ADCMP API

7.20.5.3 Mft_Adcmp_Init ()

This function does device dependent initialization of Mft adcmp module.

Prototype	
`en_result_t Mft_Adcmp_Init(volatile stc_mftn_adcmp_t* pstcMftAdcmp, uint8_t u8Ch, stc_mft_adcmp_config_t* pstcConfig);`	
Parameter Name	**Description**
[in] pstcMftAdcmp	Pointer to a MFT instance.
[in] u8Ch	Mft adcmp channel.
[in] pstcConfig	Pointer to Mft adcmp config.
Return Values	**Description**
Ok	Initialization successful done.
ErrorInvalidParameter	pstcMftAdcmp == NULL pstcConfig == NULL u8Ch > MFT_ADCMP_CH_MAX Other invalid configuration setting(s)

7.20.5.4 Mft_Adcmp_EnableOperation ()

This function enables Mft Adcmp operations.

Prototype	
`en_result_t Mft_Adcmp_EnableOperation(volatile stc_mftn_adcmp_t *pstcMftAdcmp, uint8_t u8Ch, stc_mft_adcmp_func_t* pstcFunc);`	
Parameter Name	**Description**
[in] pstcMftAdcmp	Pointer to a MFT instance.
[in] u8Ch	MFT Adcmp channel.
[in] pstcFunc	MFT Adcmp function.
Return Values	**Description**
Ok	Enable adcmp opearations successfully.
ErrorInvalidParameter	pstcMftAdcmp == NULL pstcFunc == NULL u8Ch > MFT_ADCMP_CH_MAX

7.20.5.5 Mft_Adcmp_DisableOperation ()

This function disables Mft Adcmp operations.

Prototype	
`en_result_t Mft_Adcmp_DisableOperation(volatile stc_mftn_adcmp_t *pstcMftAdcmp, uint8_t u8Ch, stc_mft_adcmp_func_t* pstcFunc);`	
Parameter Name	**Description**
`[in] pstcMftAdcmp`	Pointer to a MFT instance.
`[in] u8Ch`	MFT Adcmp channel.
`[in] pstcFunc`	MFT Adcmp function.
Return Values	**Description**
`Ok`	Disable adcmp opearations successfully.

`ErrorInvalidParameter`	`pstcMftAdcmp == NULL`
	`pstcFunc == NULL`
	`u8Ch > MFT_ADCMP_CH_MAX`

7.20.5.6 Mft_Adcmp_WriteAcmp ()

This function writes compare and offset value to Mft ACMP.

Prototype	
`en_result_t Mft_Adcmp_WriteAcmp(volatile stc_mftn_adcmp_t *pstcMftAdcmp, uint8_t u8Ch, uint16_t u16AcmpVal);`	
Parameter Name	**Description**
`[in] pstcMftAdcmp`	Pointer to a MFT instance.
`[in] u8Ch`	MFT Adcmp channel.
`[in] u16AcmpVal`	Compare value.
Return Values	**Description**
`Ok`	Write value done.
`ErrorInvalidParameter`	`pstcMftAdcmp == NULL` `u8Ch > MFT_ADCMP_CH_MAX`

7.20.5.7 Mft_Adcmp_ReadAcmp ()

This function reads compare and offset value to ACMP.

Prototype	
`uint16_t Mft_Adcmp_ReadAcmp(volatile stc_mftn_adcmp_t *pstcMftAdcmp, uint8_t u8Ch);`	
Parameter Name	**Description**
[in] pstcMftAdcmp	Pointer to a MFT instance.
[in] u8Ch	Mft Adcmp channel.
Return Values	**Description**
value	Value of register ACMP.
ErrorInvalidParameter	pstcMftAdcmp == NULL u8Ch > MFT_ADCMP_CH_MAX

7.20.5.8 Mft_Adcmp_Fm3_Init ()

This function does Mft Adcmp fm3 compatible initialization.

Prototype	
`en_result_t Mft_Adcmp_Fm3_Init(volatile stc_mftn_adcmp_t *pstcMftAdcmp, uint8_t u8CoupleCh, stc_mft_adcmp_fm3_config_t *pstcConfig);`	
Parameter Name	**Description**
[in] pstcMftAdcmp	Pointer to a MFT instance.
[in] u8CoupleCh	Mft Adcmp channel.
[in] pstcConfig	Pointer to a MFT Adcmp configuration.
Return Values	**Description**
Ok	Mft Adcmp fm3 compatible init successfully.
ErrorInvalidParameter	u8CoupleCh is invalid. pstcConfig parameter invalid.

7.20.5.9 Mft_Adcmp_Fm3_EnableOperation ()

This function does Mft Adcmp fm3 compatible enable operation.

Prototype	
`en_result_t Mft_Adcmp_Fm3_EnableOperation(volatile stc_mftn_adcmp_t *pstcMftAdcmp, uint8_t u8CoupleCh);`	
Parameter Name	**Description**
[in] pstcMftAdcmp	Pointer to a MFT instance.
[in] u8CoupleCh	MFT Adcmp channel.
Return Values	**Description**
Ok	Enable operation done.
ErrorInvalidParameter	u8CoupleCh > MFT_ADCMP_CPCH_MAX pstcMftAdcmp == NULL

7.20.5.10 Mft_Adcmp_Fm3_DisableOperation ()

This function does Mft Adcmp fm3 compatible disable operation.

Prototype	
`en_result_t Mft_Adcmp_Fm3_DisableOperation(volatile stc_mftn_adcmp_t *pstcMftAdcmp, uint8_t u8CoupleCh);`	
Parameter Name	**Description**
[in] pstcMftAdcmp	Pointer to a MFT instance.
[in] u8CoupleCh	MFT Adcmp channel.
Return Values	**Description**
Ok	Disable operation done.
ErrorInvalidParameter	u8CoupleCh > MFT_ADCMP_CPCH_MAX pstcMftAdcmp == NULL

7.20.5.11 Mft_Adcmp_Fm3_WriteAccp ()

This function writes Accp register in Mft Adcmp fm3 compatible mode.

Prototype	
`en_result_t Mft_Adcmp_Fm3_WriteAccp(volatile stc_mftn_adcmp_t *pstcMftAdcmp, uint8_t u8CoupleCh, uint16_t u16AccpVal);`	
Parameter Name	**Description**
[in] pstcMftAdcmp	Pointer to a MFT instance.
[in] u8CoupleCh	MFT Adcmp channel.
[in] u16AccpVal	MFT Adcmp configuration parameter.
Return Values	**Description**
Ok	Write value successfully.
ErrorInvalidParameter	u8CoupleCh > MFT_ADCMP_CPCH_MAX pstcMftAdcmp == NULL

7.20.5.12 Mft_Adcmp_Fm3_ReadAccp ()

This function reads Accp register in Mft Adcmp fm3 compatible mode.

Prototype	
`uint16_t Mft_Adcmp_Fm3_ReadAccp(volatile stc_mftn_adcmp_t *pstcMftAdcmp, uint8_t u8CoupleCh);`	
Parameter Name	**Description**
[in] pstcMftAdcmp	Pointer to a MFT instance.
[in] u8CoupleCh	MFT Adcmp channel.
Return Values	**Description**
Value	Value in Accp register.
ErrorInvalidParameter	u8CoupleCh > MFT_ADCMP_CPCH_MAX pstcMftAdcmp == NULL

7.20.5.13 Mft_Adcmp_Fm3_WriteAccpdn ()

This function writes Accpdn register in Mft Adcmp fm3 compatible mode.

Prototype	
en_result_t Mft_Adcmp_Fm3_WriteAccpdn(volatile stc_mftn_adcmp_t *pstcMftAdcmp, uint8_t u8CoupleCh, uint16_t u16AccpdnVal);	
Parameter Name	**Description**
[in] pstcMftAdcmp	Pointer to a MFT instance.
[in] u8CoupleCh	MFT Adcmp channel.
[in] u16AccpdnVal	Write data value of Accpdn.
Return Values	**Description**
Ok	Write value successfully.
ErrorInvalidParameter	u8CoupleCh > MFT_ADCMP_CPCH_MAX pstcMftAdcmp == NULL

7.20.5.14 Mft_Adcmp_Fm3_ReadAccpdn ()

This function reads Accpdn register in Mft Adcmp fm3 compatible mode.

Prototype	
uint16_t Mft_Adcmp_Fm3_ReadAccpdn(volatile stc_mftn_adcmp_t *pstcMftAdcmp, uint8_t u8CoupleCh);	
Parameter Name	**Description**
[in] pstcMftAdcmp	Pointer to a MFT instance
[in] u8CoupleCh	MFT Adcmp channel
Return Values	**Description**
value	Value in Accpdn register.
ErrorInvalidParameter	u8CoupleCh > MFT_ADCMP_CPCH_MAX pstcMftAdcmp == NULL

7.21 (MFS_HL) Multi Function Serial Interface High Level

Type Definition	stc_mfsn_t
Configuration Type	stc_mfs_hl_uart_config_t, stc_mfs_hl_csio_config_t, stc_mfs_hl_lin_config_t, stc_mfs_hl_i2c_config_t
Address Operator	MFS*n*

`Mfs_Hl_Uart_Init()` initializes one of the MFS block instances to UART with a parameter `pstcConfig`. The type of the parmeter is `stc_mfs_hl_uart_config_t`. `Mfs_Hl_Uart_DeInit()` resets all of UART registers.

`Mfs_Hl_Csio_Init()` initializes one of the MFS block instances to CSIO with parameter `pstcConfig`. The type of the parmeter is `stc_mfs_hl_csio_config_t`.

This API sets timer mode or SPI mode using chip select (CS) with a parameter `pstcConfig-`

>pstcMfsSpiConfig. The type of the parameter is stc_mfs_hl_spi_config_t.
Mfs_Hl_Csio_DeInit() is used to reset all CSIO registers.

Mfs_Hl_Csio_Synchronous() transfers and receives data simultaneously. This API is only used by blocking. The interrupt is not used for this function.

Mfs_Hl_Lin_Init() initializes one of the MFS block instances to LIN with the LIN configuration. The type of the structure of the LIN configuration is stc_mfs_hl_lin_config_t.

Mfs_Hl_Lin_DeInit() resets all LIN registers.

Mfs_Hl_Lin_SetBreak() sets the LIN in master mode.

Mfs_Hl_Lin_SetNewBaudDivisor() adjests the baud rate dividor (not the rate itself!) after measurement with an ICU in LIN Slave mode.

Note that the LIN functionality only works properly when the MFS is connected to a LIN transceiver so that the SOT line can be read by SIN.

Mfs_Hl_Lin_DisableRxInterrupt() disables the Rx interrupt, if a LIN frame was completely read and a new frame beginning with the LIN break is awaited to avoid unnecessary reception of a '0'-Byte with a framing error.

Mfs_Hl_Lin_TransferRxBuffer() transfers reception data from the internal ring buffer to a user buffer. This API can be used for LIN Master and Slave mode.

Mfs_Hl_I2c_Init() initializes one of the MFS block instances to I^2C with parameter is pstcConfig. The type of the parameter stc_mfs_hl_i2c_config_t.

Mfs_Hl_I2c_DeInit() resets all I^2C registers.

Mfs_Hl_I2c_Write() transmits data. Mfs_Hl_I2c_Read() receives data. These APIs can use synchronously (blocking-call) or asynchronously(non-blocking-call). If the user uses I2C synchronously, Mfs_Hl_I2c_Write()::bBlocking or/and Mfs_Hl_I2c_Read()::bBlocking should be set to TRUE. If user uses I2C asynchronously, Mfs_Hl_I2c_Write()::bBlocking or/and Mfs_Hl_I2c_Read()::bBlocking should be set to FALSE.

Mfs_Hl_I2c_WaitTxComplete() and Mfs_Hl_I2c_WaitRxComplete() are used to check the completion of a transmission or receiption.

Mfs_Hl_Read() and Mfs_Hl_Write() can't be use for I^2C.

If Mfs_Hl_I2c_Write() and/or Mfs_Hl_I2c_Read() is/are called by non-blocking functions, Care must be taken. In this case, Mfs_Hl_I2c_WaitTxComplete() must be called periodically because the status does not change to standby when a stop condition isn't detected.

For UART, CSIO or LIN, Mfs_Hl_Read() and Mfs_Hl_Write() can use for communication. See the description of these functions for detail.

7.21.1 The usable mode in the Multi Function Serial Interface

Each Multi Function Serial Interface block instance can configured for one of the following modes.

- CSIO
- I2C
- LIN
- UART

CSIO

Role	Mode	Serial Timer(*1)	Chip Select	Example Program	
master	normal	not-use	-	X	using / non-using interrupt
		use	-	X	using / non-using interrupt
	SPI	not-use	-(*2)	X	using / non-using interrupt
			by peripheral(*3)	X	using / non-using interrupt
		use	-(*2)		-
			by peripheral(*3)		-
slave	normal	-	-	X	only using interrupt
	SPI	-	-	X	only using interrupt
			peripheral	X	only using interrupt

(*1) The serial timer is used for transmission in master mode.

(*2) In the example program, the chip select is controled by GPIO.

(*3) The chip select is only for instance #6.

I^2C

Role	Mode	Example Program	
master	normal	X	using blocking process / non-blocikin process
	Fast mode-plus		—
slave	normal	X	using blocking process / non-blocikin process
	Fast mode-plus		—

LIN

Role	Example Program	
master	X	one sample includes master and slave
slave	X	one sample includes master and slave

UART

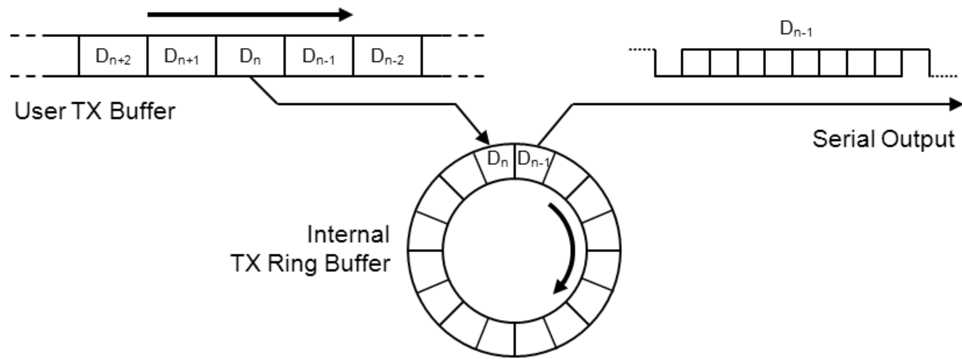
Mode	Example Program	
normal	X	using / non-using interrupt
multi-processor		—

7.21.2 Ring Buffer Principle

7.21.2.1 Transmission Ring Buffer Principle

The following illustration shows, how the transmission ring buffer works.

Figure 7-1. Transmission Ring Buffer Principle

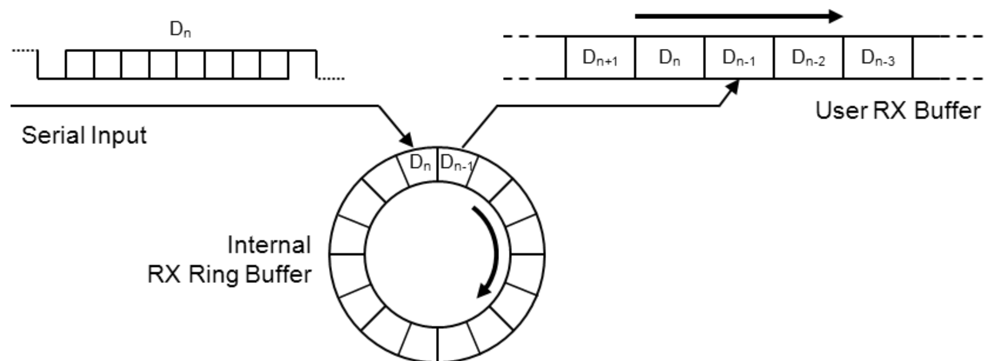


`Mfs_Hl_Write()` transmis data from a user transmission buffer to the internal ring buffer. Note that the hardware transmission FIFO between the internal ring buffer and the serial output is not drawn here. This internal ring buffer is for UART, CSIO or LIN.

7.21.2.2 Reception Ring Buffer Principle

The following illustration shows how the reception ring buffer works.

Figure 7-2. Reception Ring Buffer Principle



Reception data is stored in the internal ring buffer. From this ring buffer the data is transferred to a user buffer. Note that the hardware reception FIFO is not drawn here. This ring buffer is used in the interrupt handler for a reception of data. This is for UART, CSIO or LIN.

7.21.3 MFS_HL Configuration Structure

There are four structures of configuration. They are for UART, CSIO, LIN and I^2C. And CSIO has two more structures of configuration.

7.21.3.1 UART Configuration Structure

The argument of `Mfs_Hl_Uart_Init()` is a pointer to a stracture of the UART configuration. The type of the structure is `stc_mfs_hl_uart_config_t`. The members of `stc_mfs_hl_uart_config_t are:`

Type	Field	Possible Values	Description
uint32_t	u32DataRate	-	Baud rate (bps)
boolean_t	bBitDirection	FALSE TRUE	LSB first MSB first
boolean_t	bSignalSystem	FALSE TRUE	NRZ Inverted NRZ
boolean_t	bHwFlow	FALSE TRUE	Hardware Flow is not used Hardware Flow is used
uint8_t*	pu8TxBuf	-	Pointer to tranasmit FIFO buffer
uint8_t*	pu8RxBuf	-	Pointer to receive FIFO buffer
uint16_t	u16TxBufSize	-	Size of tramsmit FIFO buffer
uint16_t	u16RxBufSize	-	Size of receive FIFO buffer
uint16_t	u16RxCbBufFillLvl	-	Unread counts of data buffer to call RX Callback function
uint8_t	u8UartMode	MfsUartNormal MfsUartMulti	Normal mode Multi-Processor Mode
uint8_t	u8Parity	MfsParityNone MfsParityEven MfsParityOdd	No parity bit is used Even parity bit is used Odd parity bit is used
uint8_t	u8StopBit	MfsOneStopBit MfsTwoStopBits MfsThreeStopBits MfsFourStopBits	1 Stop Bit 2 Stop Bits 3 Stop Bits 4 Stop Bits
uint8_t	u8CharLength	MfsFiveBits MfsSixBits MfsSevenBits MfsEightBits MfsNineBits	5 Bits character length 6 Bits character length 7 Bits character length 8 Bits character length 9 Bits character length
uint8_t	u8FifoUsage	MfsHlUseNoFifo MfsHlUseFifo	Don't use MFS FIFO function Use MFS FIFO function
mfs_hl_rx_cb…_func_ptr_t	pfnRxCb	-	Callback function, if RX buffer is filled more than u16RxCbBufFillLvl
mfs_hl_tx_cb…_func_ptr_t	pfnTxCb	-	Callback function, if TX Buffer is empty

7.21.3.2 CSIO Configuration Structure

The argument of `Mfs_Hl_Csio_Init ()` is a pointer to a structure of the CSIO. The type of the structure is `stc_mfs_hl_csio_config_t`. The members of `stc_mfs_hl_csio_config_t` are:

Type	Field	Possible Values	Description
uint32_t	u32DataRate	-	Baud rate (bps)
boolean_t	bBitDirection	FALSE TRUE	LSB first MSB first
boolean_t	bSignalSystem	FALSE TRUE	SCK Mark Level High SCK Mark Level Low
stc_mfs_hl_spi_config_t*	pstcMfsSpiCsConfig	-	A pointer to a structure of the SPI configuration
stc_mfs_hl_timer_config_t*	pstcMfsTimerConfig	-	A pointer to a structure of the serial timer configuration
uint8_t*	pu8TxBuf	-	A pointer to tranasmit FIFO buffer
uint8_t*	pu8RxBuf	-	A pointer to receive FIFO buffer
uint16_t	u16TxBufSize	-	Size of tramsmit FIFO buffer
uint16_t	u16RxBufSize	-	Size of receive FIFO buffer
uint16_t	u16RxCbBufFillLvl	-	Unread counts of data buffer to call RX Callback function
uint8_t	u8CsioMode	MfsCsioMaster MfsCsioSlave	Master mode Slave mode
uint8_t	u8CsioActMode	MfsCsioAct… NormalMode MfsCsioAct… SpiMode	Normal mode SPI mode
uint8_t	u8SyncWaitTime	MfsSyncWaitZero MfsSyncWaitOne MfsSyncWaitTwo MfsSyncWaitThree	0 wait time insertion 1 wait time insertion 2 wait time insertion 3 wait time insertion
uint8_t	u8CharLength	MfsFiveBits MfsSixBits MfsSevenBits MfsEightBits MfsNineBits MfsTenBits MfsElevenBits MfsTwelveBits MfsThirteenBits MfsFourteenBits MfsFifteenBits MfsSixteenBits MfsTwentyBits MfsTwentyFourBits MfsThirtyTwoBits	5 Bits character length 6 Bits character length 7 Bits character length 8 Bits character length 9 Bits character length 10 Bits character length 11 Bits character length 12 Bits character length 13 Bits character length 14 Bits character length 15 Bits character length 16 Bits character length 20 Bits character length 24 Bits character length 32 Bits character length
uint8_t	u8FifoUsage	MfsHlUseNoFifo MfsHlUseFifo	Don't use MFS FIFO function Use MFS FIFO function
mfs_hl_rx_cb…_f	pfnRxCb	-	Callback function, if RX buffer is filled more

Type	Field	Possible Values	Description
unc_ptr_t			than u16RxCbBufFillLvl
mfs_hl_tx_cb…_func_ptr_t	pfnTxCb	-	Callback function, if TX Buffer is empty

7.21.3.3 I²C Configuration Structure

The argument of Mfs_Hl_I2c_Init () is a pointer to a structure of I²C configuration. The type of the structure is stc_mfs_hl_I2c_config_t. The members of stc_mfs_hl_I2c_config_t are:

Type	Field	Possible Values	Description
uint32_t	u32DataRate	-	Baud rate (bps)
uint8_t	u8I2cMode	MfsI2cMaster MfsI2cSlave	Master mode Slave mode
uint8_t	u8SlvAddr	0x00 - 0x7F	Slave address This is effective when Slave mode is used (u8I2cMode is set to MfsI2cSlave)
uint8_t	u8FastMode	MfsI2cDisable…FastModePlus MfsI2cEnable…FastModePlus	Standard-mode Fast-mode Plus
uint8_t	u8FifoUsage	MfsHlUseNoFifo MfsHlUseFifo	Don't use MFS FIFO function Use MFS FIFO function
mfs_hl_rx_cb…_func_ptr_t	pfnRxCb	-	Callback function, if RX buffer is completed
mfs_hl_tx_cb…_func_ptr_t	pfnTxCb	-	Callback function, if TX is completed
mfs_hl_i2c_slv…_cb_func_ptr_t	pfnI2cSlvStCb	-	Callback function, if slave address is detected This is used for Slave mode (u8I2cMode is set to MfsI2cSlave)

7.21.3.4 LIN Configuration Structure

The argument of `Mfs_Hl_Lin_Init ()` is a pointer to a structure of the LIN configuration. The type of the structure is `stc_mfs_hl_lin_config_t`. The members of `stc_mfs_hl_lin_config_t` are:

Type	Field	Possible Values	Description
uint32_t	u32DataRate	-	Baud rate (bps)
boolean_t	bExtWakeUp	FALSE TRUE	Disable external wake-up Enable external wake-up
boolean_t	bLinBreakIrqEnable	FALSE TRUE	Disable LIN break RX interrupt Enable LIN break RX interrupt
uint8_t*	pu8TxBuf	-	Pointer to tranasmit FIFO buffer
uint8_t*	pu8RxBuf	-	Pointer to receive FIFO buffer
uint16_t	u16TxBufSize	-	Size of tramsmit FIFO buffer
uint16_t	u16RxBufSize	-	Size of receive FIFO buffer
uint8_t	u8LinMode	MfsLinMaster MfsLinSlave	Master mode Slave mode
uint8_t	u8StopBits	MfsLinOneStopBit MfsLinTwoStopBits MfsLinThreeStopBits MfsLinFourStopBits	1 Stop Bit 2 Stop Bits 3 Stop Bits 4 Stop Bits
uint8_t	u8BreakLength	MfsLinBreakLength13 MfsLinBreakLength14 MfsLinBreakLength15 MfsLinBreakLength16	Lin Break Length 13 Bit Times 14 Bit Times 15 Bit Times 16 Bit Times
uint8_t	u8DelimiterLength	MfsLinDelimiterLength1 MfsLinDelimiterLength2 MfsLinDelimiterLength3 MfsLinDelimiterLength4	Lin Break Delimiter Length 1 Bit Time 2 Bit Times 3 Bit Times 4 Bit Times
uint8_t	u8FifoUsage	MfsHlUseNoFifo MfsHlUseFifo	Don't use MFS FIFO function Use MFS FIFO function
mfs_hl_rx_... cb_func_ptr_t	pfnRxCb	-	Callback function, if RX buffer is filled more than 1 byte
mfs_hl_tx_… cb_func_ptr_t	pfnTxCb	-	Callback function, if TX Buffer is empty
mfs_hl_lin_... brk_func_ptr_t	pfnLinBrkCb	-	Callback function, if LIN break was detected

7.21.3.5 SPI Serial Chip Select Configuration Structure

When CSIO is configured as SPI, `pstcMfsSpiConfig` should be set. The type of `pstcMfsSpiConfig` is `stc_mfs_hl_spi_config_t`. The members of `stc_mfs_hl_spi_config_t` are:

Type	Field	Possible Values	Description
boolean_t	bCsLevel	-	Baud rate (bps)
uint16_t	u16CsDeSelect	0x0000 - 0xFFFF	The minimum period from the time when the Serial Chip Select pin becomes inactive to the time when it becomes active again
uint8_t	u8CsSetDelay	0x00 - 0xFF	The period from the time when the Serial Chip Select pin becomes active to the time when the Serial Clock is output
uint8_t	u8CsHoldDelay	0x00 - 0xFF	The periol from the time when the Serial Clock output is finished to the time when the Serial Chip Select pin becomes inactive
uint8_t	u8CsDivision	See below	Serial Chip Select Timing Operation Clock Division

Serial Chip Select Timing Operation Clock Division Definisions

Serial Chip Select Timing Operation Clock Division Definisions

Definition	Description
MFS_SCRCR_CDIV_NONE	No division
MFS_SCRCR_CDIV_2	Divided by 2
MFS_SCRCR_CDIV_4	Divided by 4
MFS_SCRCR_CDIV_8	Divided by 8
MFS_SCRCR_CDIV_16	Divided by 16
MFS_SCRCR_CDIV_32	Divided by 32
MFS_SCRCR_CDIV_64	Divided by 64

7.21.3.6 Serial Timer Configuration Structure

When CSIO is configured as normal master mode with serial timer, `pstcMfsTimerConfig should be set.` The type of `pstcMfsTimerConfig` is `stc_mfs_hl_timer_config_t`. The members of `stc_mfs_hl_timer_config_t` are:

Type	Field	Possible Values	Description
boolean_t	bTimerSyncEnable	FALSE TRUE	Disable synchronous transfer Enable synchronous transfer
uint16_t	u16SerialTimer	0x0000 - 0xFFFF	Serial timer period value If this is not zero serial timer is activate
uint8_t	u8TimerDivision	See belowWc_Init ()	Timer Operation Clock Division
uint8_t	u8TxByte	0x00 - 0xFF	Transfer counts If this sets 0x00, transfer counts is eight. After starting the transmission synchronizing with the timer, the data of count specified this is transferred. This is effective when `bTimerSyncEnable` is set to `TRUE`.

(*) If `pstcMfsSpiCsConfig` in the type `stc_mfs_hl_csio_config_t` sets not `NULL`, this structure (`pstcMfsTimerConfig` in the type `stc_mfs_hl_timer_config`) is not effective.

Timer Operation Clock Division Definitions

Definition	Description
`MFS_SCRCR_TDIV_NONE`	No division
`MFS_SCRCR_TDIV_2`	Divided by 2
`MFS_SCRCR_TDIV_4`	Divided by 4
`MFS_SCRCR_TDIV_8`	Divided by 8
`MFS_SCRCR_TDIV_16`	Divided by 16
`MFS_SCRCR_TDIV_32`	Divided by 32
`MFS_SCRCR_TDIV_64`	Divided by 64
`MFS_SCRCR_TDIV_128`	Divided by 128
`MFS_SCRCR_TDIV_256`	Divided by 256

7.21.4 API Reference

7.21.4.1 Mfs_Hl_Uart_Init()

Configures the MFS block insetance for UART.

Prototype	
`en_result_t Mfs_Hl_Uart_Init( volatile stc_mfsn_t* pstcUart, stc_mfs_hl_uart_config_t* pstcConfig)`	
Parameter Name	**Description**
[in] `pstcUart`	A pointer to the MFS block instance
[in] `pstcConfig`	A pointer to a structure of the UART configuration
Return Values	**Description**
`Ok`	Initialization ended with no error

`ErrorInvalidParameter`	`pstcUart == NULL`
	`pstcConfig == NULL`
	`pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)
	The Parameter is out of range

7.21.4.2 *Mfs_Hl_Uart_DeInit()*

Deinitializes the MFS block insetance, which is configured for UART.

Prototype	
`en_result_t Mfs_Hl_Uart_DeInit(volatile stc_mfsn_t* pstcUart)`	
Parameter Name	**Description**
[in] `pstcUart`	A pointer to the MFS block instance
Return Values	**Description**
`Ok`	Deinitialization ended with no error
`ErrorInvalidParameter`	`pstcUart == NULL` `pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)

7.21.4.3 *Mfs_Hl_Csio_Init()*

Configures the MFS block insetance for CSIO.

Prototype	
`en_result_t Mfs_Hl_Csio_Init( volatile stc_mfsn_t* pstcCsio, stc_mfs_hl_csio_config_t* pstcConfig)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS block instance
[in] `pstcConfig`	A pointer to a structure of the CSIO configuration
Return Values	**Description**
`Ok`	Initialization ended with no error

`ErrorInvalidParameter`	`pstcCsio == NULL`
	`pstcConfig == NULL`
	`pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)
	The parameter is out of range

7.21.4.4 *Mfs_Hl_Csio_DeInit()*

Deinitializes the MFS block insetance, which is configured for CSIO.

Prototype	
`en_result_t Mfs_Hl_Csio_DeInit(volatile stc_mfsn_t* pstcCsio)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS block instance
Return Values	**Description**
`Ok`	Deinitialization ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)

7.21.4.5 Mfs_Hl_Csio_SynchronousTrans()

This function puts and retreives the same amount of data in synchronous mode. This function puts the data from the `pu8TxData` parameter into the TX ring buffer. Simulteneousely it retrieaves the data from the RX ring buffer to the `pu8RxData` parameter.

This function operates in blocking mode. This function waits until the amount of data defined by `u16TransferSize` is put / retreived. The TX / RX callback functions are not called.

`Mfs_HI_Csio_SynchronouseTrans()` is a blocking function. no interrupt is used and no FIFO is used.

Notes:

- `Mfs_Hl_Write()` and `Mfs_Hl_Read()` provides synchronous (non-blocking) TX / RX operations for MFS CSIO master and slave modes. Note that these functions do not support full-duplex operation.
- This function can be used only if the character length was set to less or equal to eight bits.
- `pu8TxData` or `pu8RxData` can be set to NULL, in this case, the other should be set to non-NULL.
- If `pu8TxData` is set to `NULL`, this function sends dummy data.
- If `pu8RxData` is set to `NULL`, this function throws the reception data.

It has the following format:

Prototype

```
en_result_t Mfs_Hl_Csio_SynchronousTrans(volatile stc_mfsn_t*    pstcCsio,
                                         const uint8_t*          pu8TxData,
                                         uint8_t*                pu8RxData,
                                         uint16_t                u16TransferSize,
                                         boolean_t               bCsHolding)
```

Parameter Name	Description
[in] `pstcCsio`	A pointer to the MFS block instance
[in] `pu8TxData`	A pointer to the data to put (can be set `NULL`)
[in,out] `pu8RxData`	A pointer to the data to retreive (can be set `NULL`)
[in] `u16TransferSize`	Data count
[in] `bCsHolding`	Hold chip select

Return Values	Description
`Ok`	Transfer ended with no error

`ErrorInvalidParameter`	`pstcCsio == NULL`
	`pu8TxData == NULL` or `pu8RxData == NULL`
	`u16TransferSize == 0`
	`pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)

7.21.4.6 *Mfs_Hl_I2c_Init()*

Configures the MFS block insetance for I2C.

Prototype	
`en_result_t Mfs_Hl_I2c_Init( volatile stc_mfsn_t* pstcI2c, stc_mfs_hl_i2c_config_t* pstcConfig)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS block instance
[in] `pstcConfig`	A pointer to a structure of the I^{2}C configuration
Return Values	**Description**
`Ok`	Initialization ended with no error

`ErrorInvalidParameter`	`pstcI2c == NULL`
	`pstcConfig == NULL`
	`pstcMfsHlInternData == NULL` (invalid or the MFS block is disabled)
	The parameter is out of range

7.21.4.7 *Mfs_Hl_I2c_DeInit()*

Deinitializes the MFS block insetance, which is configured as I2C.

Prototype	
`en_result_t Mfs_Hl_I2c_DeInit(volatile stc_mfsn_t* pstcI2c)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS block instance
Return Values	**Description**
`Ok`	Deinitialization ended with no error
`ErrorInvalidParameter`	`pstcI2c == NULL` `pstcMfsHlInternData == NULL` (invalid or the MFS block is disabled)

7.21.4.8 *Mfs_Hl_I2c_Write()*

Puts data from the parameter `pu8data` into the MFS block FIFO.
When `bBlocking` is set to `FALSE`, this function returns immediately.
When `bBlocking` is set to `TRUE`, this function waits until all of the data is put into the FIFO.
If I2C is configured to have no FIFO, The single data is put into the FIFO.
Note: Don't access to the data area to put until this function returns.

Prototype	
`en_result_t Mfs_Hl_I2c_Write( volatile stc_mfsn_t* pstcI2c, uint8_t u8SlaveAddr, uint8_t* pu8Data, uint16_t* pu16WriteCnt, boolean_t bBlocking)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS block instance
[in] `u8SlaveAddr`	A slave address in the master mode
[in] `pu8Data`	A pointer to the data area
[in,out] `pu16WriteCnt`	A pointrer to the data count And a pointrer to the actual transfered data count
[in] `bBlocking`	Whether blocking or non-blocking
Return Values	**Description**
`Ok`	Putting data ended with no error

`ErrorInvalidParameter`	`pstcI2c == NULL`
	`pu8Data == NULL`
	`pu16WriteCnt == NULL`
	`pstcMfsHlInternData == NULL` (invalid or the MFS block is disabled)

`ErrorOperationInProgress`	Putting data is still ongoing
`ErrorTimeout`	I2C communication time out occures

7.21.4.9 *Mfs_Hl_I2c_Read()*

Retrieves data from the FIFO.
If `bBlocking` is set to `FALSE` (non-blocking), this function returns immediately after the data is retrieved to the parameter `pu8Data`.
Note: Don't access to the data area to retrieve until this function returns.

Prototype	
`en_result_t Mfs_Hl_I2c_Read(volatile stc_mfsn_t* pstcI2c, uint8_t u8SlaveAddr, uint8_t* pu8Data, uint16_t* pu16ReadCnt, boolean_t bBlocking)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS block instance
[in] `u8SlaveAddr`	A slave address in the master mode
[in,out] `pu8Data`	A pointer to the data area
[in,out] `pu16ReadCnt`	A pointrer to the data count (at least 1) And a pointrer to the actual transfered data count
[in] `bBlocking`	Whether blocking or non-blocking
Return Values	**Description**
`Ok`	Retrieving data ends without erorr

`ErrorInvalidParameter`	`pstcI2c == NULL`
	`pu8Data == NULL`
	`pu16ReadCnt == NULL`
	`pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)

`ErrorOperationInProgress`	Retrieving data is still ongoing
`ErrorTimeout`	I2C communication time out occures

7.21.4.10 Mfs_Hl_I2c_WaitTxComplete()

When this function is called, a internal up-counter increase and the up-counter is cleard to zero when `Mfs_Hl_I2c_Write()` is called.
This function is useful when the user calls `Mfs_Hl_I2c_Write()` in non-blocking mode. The user checks if `Mfs_Hl_I2c_Write()` surely puts the data into the FIFO with this function.

Prototype	
`en_result_t Mfs_Hl_I2c_WaitTxComplete( volatile stc_mfsn_t* pstcI2c, uint32_t u32MaxCnt)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS block instance
[in] `u32MaxCnt`	Maximum period
Return Values	**Description**
`Ok`	Putting the data was completed within the maximum period
`ErrorInvalidParameter`	`pstcI2c == NULL` `pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)
`ErrorInvalidMode`	Not in non-blocking mode
`ErrorOperationInProgress`	Putting data is ongoing
`ErrorTimeout`	Putting data was not completed within the maximum period

7.21.4.11 Mfs_Hl_I2c_WaitRxComplete()

When this function is called, a internal up-counter increase and the up-counter is cleard to zero when `Mfs_Hl_I2c_Read()` is called.
This function is useful when the user calls `Mfs_Hl_I2c_Read()` in non-blocking mode. The user checks if `Mfs_Hl_I2c_Read()` surely retrieves the data from the FIFO with this function.

Prototype	
`en_result_t Mfs_Hl_I2c_WaitRxComplete(volatile stc_mfsn_t* pstcI2c, uint32_t u32MaxCnt)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS block instance
[in] `u32MaxCnt`	Maximum period
Return Values	**Description**
`Ok`	Receiving data was completed within the maximum period
`ErrorInvalidParameter`	`pstcI2c == NULL` `pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)
`ErrorInvalidMode`	Not in non-blocking mode
`ErrorOperationInProgress`	Retrieving data is ongoing
`ErrorTimeout`	Retrieving data was not completed within the maximum period

7.21.4.12 Mfs_Hl_Lin_Init()

Configures the MFS block instance for LIN.

Prototype	
`en_result_t Mfs_Hl_Lin_Init( volatile stc_mfsn_t* pstcLin, stc_mfs_hl_lin_config_t* pstcConfig)`	
Parameter Name	**Description**
[in] `pstcLin`	A pointer to the MFS block instance
[in] `pstcConfig`	A pointer to a structure of the LIN configuration
Return Values	**Description**
`Ok`	Initialization ended with no error

`ErrorInvalidParameter`	`pstcLin == NULL`
	`pstcConfig == NULL`
	`pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)
	the `pstcConfig` parameter is out of range

7.21.4.13 Mfs_Hl_Lin_DeInit()

Deinitializes the MFS block insetance, which is configured for LIN.

Prototype	
`en_result_t Mfs_Hl_Lin_DeInit(volatile stc_mfsn_t* pstcLin)`	
Parameter Name	**Description**
[in] `pstcLin`	A pointer to the MFS block instance
Return Values	**Description**
`Ok`	Deinitialization ended with no error
`ErrorInvalidParameter`	`pstcLin == NULL` `pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)

7.21.4.14 Mfs_Hl_Lin_SetBreak()

Sets the LIN break and the break delimiter length.
The break delimiter length is set by the previous initialization.
LIN must be configured for master mode.

Prototype	
`en_result_t Mfs_Hl_Lin_SetBreak(volatile stc_mfsn_t* pstcLin)`	
Parameter Name	**Description**
[in] `pstcLin`	A pointer to the MFS block instance
Return Values	**Description**
`Ok`	LIN break is (being) generated
`ErrorInvalidParameter`	`pstcLin == NULL` `pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)
`ErrorInvalidMode`	LIN is not in the master mode
`ErrorOperationInProgress`	LIN is not ready to generate LIN break

7.21.4.15 Mfs_Hl_Lin_SetNewBaudDivisor()

Sets a new (calculated) baud divisor, if the LIN is in the slave mode.

Notes:This function should be called:

- Between the end of complete frame and the next LIN break.
- Between the second ICU interrupt within the LIN Synch Field and the next start bit in the LIN Header.

Prototype	
`en_result_t Mfs_Hl_Lin_SetNewBaudDivisor(volatile stc_mfsn_t* pstcLin,` `uint16_t u16BaudDivisor)`	
Parameter Name	**Description**
[in] `pstcLin`	A pointer to the MFS block instance
[in] `u16BaudDivisor`	New (calculated) baud divisor
Return Values	**Description**
`Ok`	Seting a new (calculated) baud divisor ended with no error
`ErrorInvalidParameter`	`pstcLin == NULL` `pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)
`ErrorInvalidMode`	LIN is not in the slave mode

7.21.4.16 Mfs_Hl_Lin_TransferRxBuffer()

Retrieves data from the FIFO.

Prototype	
`en_result_t Mfs_Hl_Lin_TransferRxBuffer(volatile stc_mfsn_t* pstcLin, uint8_t* pu8Data uint16_t u16ReadCount)`	
Parameter Name	**Description**
[in] `pstcLin`	A pointer to the MFS block instance
[in,out] `pu8Data`	A pointer to the data area
[in] `u16ReadCount`	A data count
Return Values	**Description**
`Ok`	Retrieving data ended with no error
`ErrorInvalidParameter`	`pstcLin == NULL` `pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)
`ErrorInvalidMode`	LIN is in slave mode

7.21.4.17 Mfs_Hl_Lin_DisableRxInterrupt()

Disables receive interrupts.

Prototype	
`en_result_t Mfs_Hl_Lin_SetBreak(volatile stc_mfsn_t* pstcLin)`	
Parameter Name	**Description**
[in] `pstcLin`	A pointer to MFS block instance
Return Values	**Description**
`Ok`	Receive interrupts are disabled with no error
`ErrorInvalidParameter`	`pstcLin == NULL` `pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)

7.21.4.18 Mfs_Hl_Write()

Puts the data to MFS synchronously or asynchronously. This function can be used for UART or CSIO or LIN.

The data provided by `pu8Data` is put to the internal TX buffer and the transmission (via TX interrupt) starts, if the previous transmission is not ongoing.

Depending on the parameter `bBlocking`, this function returns differently.

For an asynchronous (non-blocking) call (`bBlocking = FALSE`), the free momory space in the internal buffer must be sufficient to take all of the data (`pu8Data`) of the length of `u16WriteCnt`, otherwise this function will return `ErrorBufferFull`.

After all of the data is put to the internal buffer, this function will return immediately. The transmission may be pending when the function returns.

For a synchronous (blocking) call (`bBlocking = TRUE`), this function will wait until all of the data is transferred to the MFS hardware FIFO. The transmission may be pending when the function returns. If the referenced MFS does not have a FIFO, a single data is put.

If the callback function is not set, this function is called only in blocking mode.

Prototype

```
en_result_t Mfs_Hl_I2c_Write(    volatile stc_mfsn_t*   pstcMfs,
                                         uint8_t*       pu8Data,
                                         uint16_t*      pu16WriteCnt,
                                         boolean_t      bBlocking,
                                         boolean_t      bCsHolding)
```

Parameter Name	Description
[in] `pstcMfs`	A pointer to the MFS block instance
[in] `pu8Data`	A pointer to the data area
[in] `pu16WriteCnt`	A pointer to the data count (at least 1)
[in] `bBlocking`	Whether blocking or non-blocking
[in] `bCsHolding`	Hold chip select This parameter is used in CSIO or SPI master mode and the chip select is used.

Return Values	Description
`Ok`	Putting data ended with no error

`ErrorInvalidParameter`	`pstcMfs == NULL`
	`pu8Data == NULL`
	`pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)

`ErrorOperationInProgress`	Putting data is still ongoing
`ErrorBufferFull`	Free memory space in the TX buffer is not sufficient (only when `bBlocking` is set to `FALSE`)

7.21.4.19 *Mfs_Hl_Read()*

Retrieves the data from MFS synchronously or asynchronously. This function can be used for UART or CSIO or LIN.

The data is retieved from the internal RX buffer to the parameter `pu8Data`. The parameter `pu16DataCnt` is reseive data size. Depending on the parameter `bBlocking`, this function returns differently.

For an asynchronous (non-blocking) call (`bBlocking = FALSE`), this function will return immediately after all of the data (in SW ring buffer and HW FIFO) is retrieved to the parameter `pu8Data` or the retrieved data count reached to the parameter `u16ReadCnt`. The parameter p`u16DataCnt` gives the count of characters that were retrieved.

If the referenced MFS does not have a FIFO, a single data is retrieved.

For a synchronous (blocking) call (`bBlocking == TRUE`), this function will return after the count of retrieved data reached to the parameter `pu16DataCnt`.

Prototype	

`en_result_t Mfs_Hl_Read( volatile stc_mfsn_t* pstcMfs,`
`uint8_t* pu8Data,`
`uint16_t* pu16ReadCnt,`
`uint16_t u16ReadCnt`
`boolean_t bBlocking)`	

Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS block instance
[in,out] `pu8Data`	A pointer to the data area
[in,out] `pu16ReadCnt`	A pointrer to the data count (at least 1) And a pointrer to the actual retrieved data count
[in] `u16ReadCnt`	Maximum receive data count (ensure `u16DataCnt is sufficient`)
[in] bBlocking	Whether blocking or non-blocking
Return Values	**Description**
`Ok`	Retrieving data ends with no erorr

`ErrorInvalidParameter`	`pstcMfs == NULL`
	`pu8Data == NULL`
	`pu16ReadCnt == NULL`
	`pstcMfsHlInternData == NULL` (invalid or disabled MFS unit)

`ErrorOperationInProgress`	Retrieving data is still ongoing

7.21.4.20 Callback Functions

Receive Buffer Filled Callback Function

`Mfs_Hl_Uart_Init()` or `Mfs_Hl_Csio_Init()` or `Mfs_Hl_I2c_Init()` or `Mfs_Hl_Lin_Init()` registers a callback function which is called when the receive data count matches the given data count. For UART or CSIO, the receive data count is given by `Mfs_Hl_Uart_Init()` or the parameter of `Mfs_Hl_Csio_Init()`, `pstcConfig->u16RxCbBufFillLvl`. For I^{2}C, it is given by the parameter of `Mfs_Hl_I2c_Read()`, `pu16ReadCnt`. For LIN, it is always 1.

Prototype	
`void (*mfs_hl_rx_cb_func_ptr_t)(uint16_t)`	
Parameter Name	**Description**
[in] `uint16_t`	Un-read count which filled in the receive buffer

Send Completed Callback Function

A callback function is registered by `Mfs_Hl_Uart_Init()` or `Mfs_Hl_Csio_Init()` or `Mfs_Hl_I2c_Init()` or `Mfs_Hl_Lin_Init()`. The callback function is called when the given data count by `Mfs_Hl_Write()` (for UART or CSIO or LIN) or `Mfs_Hl_I2c_Write()` (for I^{2}C) is sent from the data buffer to the data send register.

Prototype	
`void (*mfs_hl_tx_cb_func_ptr_t)(uint16_t)`	
Parameter Name	**Description**
[in] `uint16_t`	Send data count from the data buffer

Starting I2C Slave Callback Function

`Mfs_Hl_I2c_Init()` registers a callback function which is called when the valid slave address was detected from the I^{2}C master. If the callback function is called, application which registered this callback function should return the first data to send to the master. The return data can be set up arbitrarily. (status, data bytes, etc.)

Prototype	
`uint8_t (*mfs_hl_i2c_slv_cb_func_prt_t)(uint8_t)`	
Parameter Name	**Description**
[in] `uint8_t`	Request from master `MfsI2cRead`: Read request (Slave transfers data to the master) `MfsI2cWrite`: Write request (Slave receives data from the master)
Return Values	Description
`uint8_t`	The first data to send to the master.

LIN Break Detected Callback Function

A callback function is registered by `Mfs_Hl_Lin_Init()`. The callback function is called when the LIN break field was detected from the remote.

Prototype
`void (*mfs_hl_lin_brk_func_ptr_t)(void)`

7.21.5 Example Software

The example software is in *\example\mfs\high_level\<module block>*.

Folder	Summary
\example\mfs\high_level\CSIO	CSIO samples folder
`mfs_csio_normal_master_unuse_int`	CSIO normal master mode without interrupt
`mfs_csio_normal_master_use_int`	CSIO normal master mode with interrupt
`mfs_csio_normal_master_unuse_int_with_tmr`	CSIO normal master mode without interrupt and with serial timer
`mfs_csio_normal_master_use_int_with_tmr`	CSIO normal master mode with interrupt and serial timer
`mfs_csio_normal_slave_use_int`	CSIO normal slave mode with interrupt
`mfs_csio_spi_master_unuse_int`	CSIO SPI master mode without interrupt and with chip select (CS) control by GPIO
`mfs_csio_spi_master_use_int`	CSIO SPI master mode with interrupt and CS control by GPIO
`mfs_csio_spi_master_unuse_int_with_cs`	CSIO SPI master mode without interrupt and with CS control by peripheral function
`mfs_csio_spi_master_use_int_with_cs`	CSIO SPI master mode with interrupt and CS control by peripheral function
`mfs_csio_spi_slave_use_int`	CSIO SPI slave mode with interrupt and without CS control
`mfs_csio_spi_slave_use_int_with_cs`	CSIO SPI slave mode with interrupt and CS control by peripheral function
\example\mfs\high_level\I2C	I^2C samples folder
`i2c_master_blocking`	I^2C master mode by blocking process
`i2c_master_non_blocking`	I^2C master mode by non-blocking process
`i2c_slave_blocking`	I^2C slave mode by blocking process
`i2c_slave_non_blocking`	I^2C slave mode by non-blocking process
\example\mfs\high_level\LIN	LIN sample folder
`mfs_lin`	LIN master and slave mode
\example\mfs\high_level\UART	UART samples folder
`mfs_uart_unuse_int`	UART without interrupt
`mfs_uart_use_int`	UART with interrupt

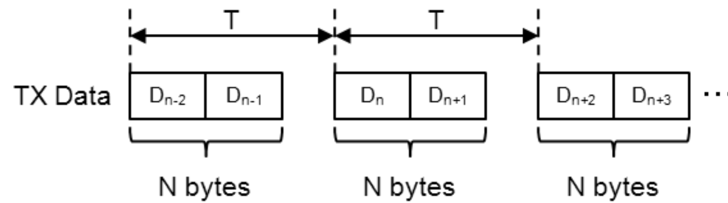
7.21.5.1 CSIO

CSIO example software is broken into three operations.

- Normal mode
- Normal mode with timer mode
- SPI mode

In timer mode, data is transferred as follows synchronizing with the serial timer.

Figure 7-3. Operation of CSIO Timer Mode



If an application runs in timer mode, `pstcMfsTimerConfig` of `stc_mfs_hl_csio_config_t` should be specified, and TRUE should be set to `bTimerSyncEnable` of `stc_mfs_hl_csio_config_t`.

T is determined by `u16SerialTimer` and `u8TimerDivision` in `stc_mfs_hl_csio_config_t`, and N is determined by `u8TxByte` in `stc_mfs_hl_csio_config_t`.

CSIO Normal Master Mode without Interrupt

This example software excerpt shows an usage of the CSIO driver library for a normal master without using interrupt.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_CSIO_TX_BUFFSIZE              (64)
#define SAMPLE_CSIO_RX_BUFFSIZE              (64)
#define SAMPLE_CSIO_RX_BUFF_FILL_LVL          (1)
...
static uint8_t au8CsioTxBuf[SAMPLE_CSIO_TX_BUFFSIZE];
static uint8_t au8CsioRxBuf[SAMPLE_CSIO_RX_BUFFSIZE];

static const stc_mfs_hl_csio_config_t stcMfsHlCsioCfg = {
  2000000,                        // Baud rate
  FALSE,                          // LSB first
  TRUE,                           // SCK Mark Level Low
  NULL,                           // SPI configuration (un-use)
  NULL,                           // Serial timer configuration (un-use)
  au8CsioTxBuf,                   // Tramsmit FIFO buffer
  au8CsioRxBuf,                   // Receive FIFO buffer
  SAMPLE_CSIO_TX_BUFFSIZE,        // Size of tramsmit FIFO buffer
  SAMPLE_CSIO_RX_BUFFSIZE,        // Size of receive FIFO buffer
  SAMPLE_CSIO_RX_BUFF_FILL_LVL,   // Unread counts of reception buffer
                                  //  to call RX Callback
  MfsCsioMaster,                  // Master mode
  MfsCsioActNormalMode,           // Normal mode
  MfsSyncWaitZero,                // Non wait time insersion
  MfsEightBits,                   // 8 data bits
  MfsHlUseNoFifo,                 // MFS FIFO is not used
  NULL,                           // Callback for RX isn't used (unuse nterrupt)
  NULL                            // Callback for TX isn't used (unuse interrupt)
};
...
function
{
  ...
  uint8_t             u8RxData;
  uint8_t             u8TxData;

  // Set CSIO Ch6_0 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR5 = FM4_GPIO->PFR5 | 0x00E0;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x00150000;

  // Initialize the MFS ch.6 as CSIO
  if (Ok != Mfs_Hl_Csio_Init(&MFS6, (stc_mfs_hl_csio_config_t *)&stcMfsHlCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // some code here ...

  while(1)
  {
    // Write and read data synchronously (blocking)
    if (Ok == Mfs_Hl_Csio_SynchronousTrans(&MFS6, &u8TxData, &u8RxData, 1, FALSE))
    {
      // some code here ...
    }
  }
}
```

CSIO Normal Master Mode with Interrupt

This example software excerpt shows an usage of the CSIO driver library for a normal master mode with using interrupt.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_CSIO_TX_BUFFSIZE      (64)
#define SAMPLE_CSIO_RX_BUFFSIZE      (64)
#define SAMPLE_CSIO_RX_BUFF_FILL_LVL          (1)
...
static void SampleMfsRxCallback(uint16_t u16RxBufFill);
static void SampleMfsTxCallback(uint16_t u16TxCnt);
...
static uint8_t au8CsioTxBuf[SAMPLE_CSIO_TX_BUFFSIZE];
static uint8_t au8CsioRxBuf[SAMPLE_CSIO_RX_BUFFSIZE];
static volatile uint16_t u16RxBufFillCnt;

static const stc_mfs_hl_csio_config_t stcMfsHlCsioCfg = {
  2000000,                   // Baud rate
  FALSE,                     // LSB first
  TRUE,                      // SCK Mark Level Low
  NULL,                      // SPI configuration (un-use)
  NULL,                      // Serial timer configuration (un-use)
  au8CsioTxBuf,              // Tramsmit FIFO buffer
  au8CsioRxBuf,              // Receive FIFO buffer
  SAMPLE_CSIO_TX_BUFFSIZE,   // Size of tramsmit FIFO buffer
  SAMPLE_CSIO_RX_BUFFSIZE,   // Size of receive FIFO buffer
  SAMPLE_CSIO_RX_BUFF_FILL_LVL,// Unread counts of reception buffer
                             //  to call RX Callback
  MfsCsioMaster,             // Master mode
  MfsCsioActNormalMode,      // Normal mode
  MfsSyncWaitZero,           // Non wait time insersion
  MfsEightBits,              // 8 data bits
  MfsHlUseNoFifo,            // MFS FIFO is not used
  SampleMfsRxCallback,       // Callback for RX is used (use interrupt)
  SampleMfsTxCallback        // Callback for TX is used (use interrupt)
};
...
```

```
static void SampleMfsRxCallback(uint16_t u16RxBufFill)
{
  // Update the filled count in RX buffer
  u16RxBufFillCnt = u16RxBufFill;
}

static void SampleMfsTxCallback(uint16_t u16TxCnt)
{
  // There is no process that should be executed
}

static en_result_t SampleMfsCsioReadWrite(uint8_t*          pu8TxBuf,
                                        uint16_t  u16WriteCnt,
                                        uint8_t*  pu8RxBuf,
                                        uint16_t* pu16ReadCnt
                                       )
{
  uint8_t           au8CsioRxDummyBuf[SAMPLE_CSIO_RX_BUFFSIZE];
  en_result_t       enResult;
  uint16_tu16ReadCnt;

  // If Rx buffer specified NULL ...
  if (NULL == pu8RxBuf)
  {
    // Use internal buffer for dummy reading
    pu8RxBuf = au8CsioRxDummyBuf;
  }
  // Write transmit data (non-blocking)
  enResult = Mfs_Hl_Write(&MFS6, pu8TxBuf, u16WriteCnt, FALSE, FALSE);
  if ((Ok == enResult) && (0 != u16WriteCnt))
  {
    // Wait to receive transmitted bytes.
    while (u16WriteCnt > u16RxBufFillCnt);
    do
    {
     // Read received data
     enResult=Mfs_Hl_Read(&MFS6,pu8RxBuf,&u16ReadCnt,u16RxBufFillCnt,FALSE);
    // If TX operation is active, RX is tried.
    } while (ErrorOperationInProgress == enResult);
    if (Ok == enResult)
    {
      if (NULL != pu16ReadCnt)
      {
        // Set the received counts
        *pu16ReadCnt = u16ReadCnt;
      }
      // Update fill count of received buffer
      __disable_irq();
      u16RxBufFillCnt -= u16ReadCnt;
      __enable_irq();
    }
  }

  return (enResult);
}
```

```
function
{
  ...
  uint16_t         u16ReadCnt;
  uint8_t          u8RxData;
  uint8_t          u8TxData;

  // Set CSIO Ch6_0 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR5 = FM4_GPIO->PFR5 | 0x00E0;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x00150000;

  // Clear the filled count of reception buffer
  u16RxBufFillCnt = 0;

  // Initialize the MFS ch.6 as CSIO
  if (Ok != Mfs_Hl_Csio_Init(&MFS6, (stc_mfs_hl_csio_config_t
*)&stcMfsHlCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // some code here ...

  while(1)
  {
    // Write and read data synchronously (blocking)
    if (Ok == SampleMfsCsioReadWrite(&u8TxData, 1 &u8RxData,
&u16ReadCnt))
    {
      // some code here ...
    }
  }
}
```

CSIO Normal Master Mode without Interrupt and with Serial Timer

This example software excerpt shows an usage of the CSIO driver library for a normal master without using interrupt and with using serial timer.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_CSIO_TX_BUFFSIZE    (64)
#define SAMPLE_CSIO_RX_BUFFSIZE    (64)
#define SAMPLE_CSIO_RX_BUFF_FILL_LVL          (1)
...
static uint8_t au8CsioTxBuf[SAMPLE_CSIO_TX_BUFFSIZE];
static uint8_t au8CsioRxBuf[SAMPLE_CSIO_RX_BUFFSIZE];

static const stc_mfs_hl_timer_config_t stcMfsHlTimerCfg = {
  TRUE,                // Enable synchronous transfer
  62500,               // Serial timer value
  MFS_SACSR_TDIV_256,  // Serial timer divider: Bus clk/256
  2                    // Transfer length : Interval is inserted per 2bytes
};

static const stc_mfs_hl_csio_config_t stcMfsHlCsioCfg = {
  100000,              // Baud rate
  FALSE,               // LSB first
  TRUE,                // SCK Mark Level Low
  NULL,                // SPI configuration (un-use)
  (stc_mfs_hl_timer_config_t *)&stcMfsHlTimerCfg,
                       // Serial timer configuration (use timer)
  au8CsioTxBuf,        // Tramsmit FIFO buffer
  au8CsioRxBuf,        // Receive FIFO buffer
  SAMPLE_CSIO_TX_BUFFSIZE         // Size of tramsmit FIFO buffer
  SAMPLE_CSIO_RX_BUFFSIZE         // Size of receive FIFO buffer
  SAMPLE_CSIO_RX_BUFF_FILL_LVL,// Unread counts of reception buffer
                       //  to call RX Callback
  MfsCsioMaster,       // Master mode
  MfsCsioActNormalMode,           // Normal mode
  MfsSyncWaitThree,    // Three bits wait time insersion
  MfsEightBits,        // 8 data bits
  MfsHlUseFifo,        // MFS FIFO is used
  NULL,                // Callback for RX isn't used (unuse interrupt)
  NULL                 // Callback for TX isn't used (unuse interrupt)
};
...
function
{
  uint8_t  au8RxBuf[4];
  uint8_t  au8TxData[4] = {0x00, 0x01, 0x02, 0x03};
  ...

  // Set CSIO Ch6_0 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR5 = FM4_GPIO->PFR5 | 0x00E0;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x00150000;

  // Initialize the MFS ch.6 as CSIO
  if (Ok != Mfs_Hl_Csio_Init(&MFS6, (stc_mfs_hl_csio_config_t *)&stcMfsHlCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // some code here ...

  while(1)
  {
    // Write and read data synchronously (blocking)
    if (Ok == Mfs_Hl_Csio_SynchronousTrans(&MFS6, au8TxData, au8RxBuf, 4, FALSE))
    {
      // some code here ...
    }
  }
}
```

CSIO Normal Master Mode with Interrupt and Serial Timer

This example software excerpt shows an usage of the CSIO driver library for a normal master with using interrupt and serial timer.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_CSIO_TX_BUFFSIZE        (64)
#define SAMPLE_CSIO_RX_BUFFSIZE        (64)
#define SAMPLE_CSIO_RX_BUFF_FILL_LVL   (1)
...
static void SampleMfsRxCallback(uint16_t u16RxBufFill);
static void SampleMfsTxCallback(uint16_t u16TxCnt);
...
static uint8_t au8CsioTxBuf[SAMPLE_CSIO_TX_BUFFSIZE];
static uint8_t au8CsioRxBuf[SAMPLE_CSIO_RX_BUFFSIZE];

static const stc_mfs_hl_csio_config_t stcMfsHlCsioCfg = {
  100000,                            // Baud rate
  FALSE,                             // LSB first
  TRUE,                              // SCK Mark Level Low
  NULL,                              // SPI configuration (un-use)
  (stc_mfs_hl_timer_config_t *)&stcMfsHlTimerCfg,
                                     // Serial timer configuration (use timer)
  au8CsioTxBuf,                      // Tramsmit FIFO buffer
  au8CsioRxBuf,                      // Receive FIFO buffer
  SAMPLE_CSIO_TX_BUFFSIZE,           // Size of tramsmit FIFO buffer
  SAMPLE_CSIO_RX_BUFFSIZE,           // Size of receive FIFO buffer
  SAMPLE_CSIO_RX_BUFF_FILL_LVL,      // Unread counts of reception buffer
                                     //  to call RX Callback
  MfsCsioMaster,                     // Master mode
  MfsCsioActNormalMode,              // Normal mode
  MfsSyncWaitThree,                  // Three bits wait time insersion
  MfsEightBits,                      // 8 data bits
  MfsHlUseFifo,                      // MFS FIFO is used
  SampleMfsRxCallback,               // Callback for RX is used (use interrupt)
  SampleMfsTxCallback                // Callback for TX is used (use interrupt)
};
...
...
function
{
   uint8_t  au8RxBuf[4];
   uint8_t  au8TxData[4] = {0x00, 0x01, 0x02, 0x03};
   uint16_t u16RxCnt;
   ...

   // Set CSIO Ch6_0 Port (SIN, SOT, SCK)
   FM4_GPIO->PFR5 = FM4_GPIO->PFR5 | 0x00E0;
   FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x00150000;

   // Initialize the MFS ch.6 as CSIO
   if (Ok != Mfs_Hl_Csio_Init(&MFS6, (stc_mfs_hl_csio_config_t *)&stcMfsHlCsioCfg))
   {
     // some code here ...
     while(1);
   }

   // some code here ...

   while(1)
   {
     // Write and read data synchronously (blocking)
     if (Ok == SampleMfsCsioReadWrite(au8TxData, 4, au8RxBuf, &u16RxCnt))
     {
       // some code here ...
     }
   }
}
```

CSIO Normal Slave Mode with Interrupt

This example software excerpt shows an usage of the CSIO driver library For a normal slave with using interrupt.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_CSIO_TX_BUFFSIZE         (64)
#define SAMPLE_CSIO_RX_BUFFSIZE         (64)
#define SAMPLE_CSIO_RX_BUFF_FILL_LVL    (1)
...
static void SampleMfsRxCallback(uint16_t u16RxBufFill);
static void SampleMfsTxCallback(uint16_t u16TxCnt);
...
static uint8_t au8CsioTxBuf[SAMPLE_CSIO_TX_BUFFSIZE];
static uint8_t au8CsioRxBuf[SAMPLE_CSIO_RX_BUFFSIZE];
static volatile uint16_t u16RxBufFillCnt;

static const stc_mfs_hl_csio_config_t stcMfsHlCsioCfg = {
  2000000,                             // Baud rate
  FALSE,                               // LSB first
  TRUE,                                // SCK Mark Level Low
  NULL,                                // SPI configuration (un-use)
  NULL,                                // Serial timer configuration (un-use)
  au8CsioTxBuf,                        // Tramsmit FIFO buffer
  au8CsioRxBuf,                        // Receive FIFO buffer
  SAMPLE_CSIO_TX_BUFFSIZE,             // Size of tramsmit FIFO buffer
  SAMPLE_CSIO_RX_BUFFSIZE,             // Size of receive FIFO buffer
  SAMPLE_CSIO_RX_BUFF_FILL_LVL,        // Unread counts of reception buffer
                                       //  to call RX Callback
  MfsCsioSlave,                        // Slave mode
  MfsCsioActNormalMode,                // Normal mode
  MfsSyncWaitZero,                     // Non wait time insersion
  MfsEightBits,                        // 8 data bits
  MfsHlUseNoFifo,                      // MFS FIFO is not used
  SampleMfsRxCallback,                 // Callback for RX is used (use interrupt)
  SampleMfsTxCallback                  // Callback for TX is used (use interrupt)
};
...
SampleMfsRxCallback(),SampleMfsTxCallback() and SampleMfsCsioReadWrite() are same as () here.
...
function
{
  uint16_t  u16ReadCnt;
  uint8_t               u8RxData;
  uint8_t               u8TxData;

  // Set CSIO Ch6_0 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR5 = FM4_GPIO->PFR5 | 0x00E0;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x00150000;

  // Clear the filled count of reception buffer
  u16RxBufFillCnt = 0;

  // Initialize the MFS ch.6 as CSIO
  if (Ok != Mfs_Hl_Csio_Init(&MFS6, (stc_mfs_hl_csio_config_t *)&stcMfsHlCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // some code here ...

  while(1)
  {
    // Write and read data synchronously (blocking)
    if (Ok == SampleMfsCsioReadWrite(&u8TxData, 1 &u8RxData, &u16ReadCnt))
    {
      // some code here ...
    }
  }
}
```

```
function
{
  uint16_t        u16ReadCnt;
  uint8_t         u8RxData;
  uint8_t         u8TxData;

  // Set CSIO Ch6_0 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR5 = FM4_GPIO->PFR5 | 0x00E0;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x00150000;

  // Clear the filled count of reception buffer
  u16RxBufFillCnt = 0;

  // Initialize the MFS ch.6 as CSIO
  if (Ok != Mfs_Hl_Csio_Init(&MFS6, (stc_mfs_hl_csio_config_t
*)&stcMfsHlCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // some code here ...

  while(1)
  {
    // Write and read data synchronously (blocking)
    if (Ok == SampleMfsCsioReadWrite(&u8TxData, 1 &u8RxData,
&u16ReadCnt))
    {
      // some code here ...
    }
  }
```

CSIO SPI Master Mode without Interrupt and with CS Control

This software example excerpt shows an usage of the CSIO driver library for a SPI master without using interrupt and with using CS control.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_SPI_TX_BUFFSIZE           (64)
#define SAMPLE_SPI_RX_BUFFSIZE           (64)
#define SAMPLE_SPI_RX_BUFF_FILL_LVL      (1)
...
static uint8_t au8CsioTxBuf[SAMPLE_SPI_TX_BUFFSIZE];
static uint8_t au8CsioRxBuf[SAMPLE_SPI_RX_BUFFSIZE];

static const stc_mfs_hl_csio_config_t stcMfsHlCsioCfg = {
  2000000,                      // Baud rate
  FALSE,                        // LSB first
  TRUE,                         // SCK Mark Level Low
  NULL,                         // SPI configuration (un-use)
  NULL,                         // Serial timer configuration (un-use)
  au8CsioTxBuf,                 // Tramsmit FIFO buffer
  au8CsioRxBuf,                 // Receive FIFO buffer
  SAMPLE_SPI_TX_BUFFSIZE,       // Size of tramsmit FIFO buffer
  SAMPLE_SPI_RX_BUFFSIZE,       // Size of receive FIFO buffer
  SAMPLE_SPI_RX_BUFF_FILL_LVL,// Unread counts of reception buffer
                                //  to call RX Callback
  MfsCsioMaster,                // Master mode
  MfsCsioActSpiMode,            // SPI mode
  MfsSyncWaitZero,              // Non wait time insersion
  MfsEightBits,                 // 8 data bits
  MfsHlUseNoFifo,               // MFS FIFO is not used
  NULL,                         // Callback for RX isn't used (unuse interrupt)
  NULL                          // Callback for TX isn't used (unuse interrupt)
};
...
static void SampleMfsSpiWait(volatile uint32_t u32Wait)
{
  // Wait specified count
  while (0 != (u32Wait--));
}

static void SampleMfsSpiEnableCs(void)
{
  // Enable CS
  FM4_GPIO->PDOR0_f.P0E = TRUE;
  // Insert wait
  SampleMfsSpiWait(40);
}

static void SampleMfsSpiDisableCs(void)
{
  // Insert wait
  SampleMfsSpiWait(40);
  // Disable CS
  FM4_GPIO->PDOR0_f.P0E = FALSE;
}
```

```
function
{
  en_result_t     enResult;
  uint8_t         u8RxData;
  uint8_t         u8TxData;

  // Set CSIO Ch6_1 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR0    = FM4_GPIO->PFR0 | 0x3800;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x002A0000;

  // Initialize the MFS ch.6 as CSIO(SPI master mode)
  if (Ok != Mfs_Hl_Csio_Init(&MFS6, (stc_mfs_hl_csio_config_t
*)&stcMfsHlCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // Chip select port is output
  FM4_GPIO->DDR0_f.P0E  = TRUE;
  FM4_GPIO->PDOR0_f.P0E = FALSE;

  // some code here ...

  while(1)
  {
    // Enable CS
    SampleMfsSpiEnableCs();
    // Write and read data synchronously (blocking)
    enResult =
Mfs_Hl_Csio_SynchronousTrans(&MFS6,&u8TxData,&u8RxData,1,FALSE);
    // Disable CS
    SampleMfsSpiDisableCs();
    if (Ok == enResult)
    {
      // some code here ...
    }
  }
}
```

CSIO SPI Master Mode with Interrupt and CS Control

This example software excerpt shows an usage of the CSIO driver library for a SPI master with using interrupt and CS control.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_SPI_TX_BUFFSIZE       (64)
#define SAMPLE_SPI_RX_BUFFSIZE       (64)
#define SAMPLE_SPI_RX_BUFF_FILL_LVL  (1)
...
static void SampleMfsRxCallback(uint16_t u16RxBufFill);
static void SampleMfsTxCallback(uint16_t u16TxCnt);
...
static uint8_t au8CsioTxBuf[SAMPLE_SPI_TX_BUFFSIZE];
static uint8_t au8CsioRxBuf[SAMPLE_SPI_RX_BUFFSIZE];
static volatile uint16_t u16RxBufFillCnt;

Configuration Structure of SPI Serial Chip Selectis same as0here

static const stc_mfs_hl_csio_config_t stcMfsHlCsioCfg = {
  2000000,                   // Baud rate
  FALSE,                     // LSB first
  TRUE,                      // SCK Mark Level Low
  NULL,                      // SPI configuration (un-use)
  NULL,                      // Serial timer configuration (un-use)
  au8CsioTxBuf,              // Tramsmit FIFO buffer
  au8CsioRxBuf,              // Receive FIFO buffer
  SAMPLE_SPI_TX_BUFFSIZE,    // Size of tramsmit FIFO buffer
  SAMPLE_SPI_RX_BUFFSIZE,    // Size of receive FIFO buffer
  SAMPLE_SPI_RX_BUFF_FILL_LVL,// Unread counts of reception buffer
                             //  to call RX Callback
  MfsCsioMaster,             // Master mode
  MfsCsioActSpiMode,         // SPI mode
  MfsSyncWaitZero,           // Non wait time insersion
  MfsEightBits,              // 8 data bits
  MfsHlUseNoFifo,            // MFS FIFO is not used
  SampleMfsRxCallback,       // Callback for RX is used (use interrupt)
  SampleMfsTxCallback        // Callback for TX is used (use interrupt)
};
...
SampleMfsRxCallback() and SampleMfsTxCallback() are same as here.
```

```
static en_result_t SampleMfsSpiReadWrite(uint8_t*      pu8TxBuf,
                                         uint16_t      u16WriteCnt,
                                         uint8_t*      pu8RxBuf,
                                         uint16_t*     pu16ReadCnt
                                         )
{
  uint8_t         au8CsioRxDummyBuf[SAMPLE_SPI_RX_BUFFSIZE];
  en_result_t     enResult;
  uint16_t        u16ReadCnt;

  // If Rx buffer specified NULL ...
  if (NULL == pu8RxBuf)
  {
    // Use internal buffer for dummy reading
    pu8RxBuf = au8CsioRxDummyBuf;
  }
  // Enable CS
  SampleMfsSpiEnableCs();
  // Write transmit data (blocking)
  enResult = Mfs_Hl_Write(&MFS6, pu8TxBuf, u16WriteCnt, TRUE, FALSE);
  // Disable CS
  SampleMfsSpiDisableCs();
  if ((Ok == enResult) && (0 != u16WriteCnt))
  {
    // Wait to receive transmitted bytes.
    while (u16WriteCnt > u16RxBufFillCnt);
    do
    {
     // Read received data

enResult=Mfs_Hl_Read(&MFS6,pu8RxBuf,&u16ReadCnt,u16RxBufFillCnt,FALSE);
    // If TX operation is active, RX is tried.
    } while (ErrorOperationInProgress == enResult);
    if (Ok == enResult)
    {
      if (NULL != pu16ReadCnt)
      {
        // Set the received counts
        *pu16ReadCnt = u16ReadCnt;
      }
      // Update fill count of received buffer
      __disable_irq();
      u16RxBufFillCnt -= u16ReadCnt;
      __enable_irq();
    }
  }

  return (enResult);
}
```

```
function
{
   uint16_t        u16ReadCnt;
  uint8_t          u8RxData;
  uint8_t          u8TxData;

  // Set CSIO Ch6_1 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR0    = FM4_GPIO->PFR0 | 0x3800;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x002A0000;

  // Clear the filled count of reception buffer
  u16RxBufFillCnt = 0;

  // Initialize the MFS ch.6 as CSIO(SPI master mode)
  if (Ok != Mfs_Hl_Csio_Init(&MFS6, (stc_mfs_hl_csio_config_t
*)&stcMfsHlCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // Chip select port is output
  FM4_GPIO->DDR0_f.P0E  = TRUE;
  FM4_GPIO->PDOR0_f.P0E = FALSE;

  // some code here ...

  while(1)
  {
    // Write and read data synchronously (blocking)
    if (Ok == SampleMfsSpiReadWrite(&u8TxData, 1, &u8RxData,
&u16ReadCnt))
    {
      // some code here ...
    }
  }
}
```

CSIO SPI Master Mode without Interrupt and with CS Control

This example software excerpt shows an usage of the CSIO dirver library for a SPI master without using interrupt and with using CS control.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_SPI_TX_BUFFSIZE       (64)
#define SAMPLE_SPI_RX_BUFFSIZE       (64)
#define SAMPLE_SPI_RX_BUFF_FILL_LVL  (1)
...
static uint8_t au8CsioTxBuf[SAMPLE_SPI_TX_BUFFSIZE];
static uint8_t au8CsioRxBuf[SAMPLE_SPI_RX_BUFFSIZE];

static const stc_mfs_hl_spi_config_t  stcMfsHlSpiCfg = {
  FALSE,          // Chip select active level
                  // (This isn't effective for master)
  20,             // Chip de-select bit
  0xFF,           // Chip select setup delay
  0xFF,           // Chip select hold delay
  MFS_SCRCR_CDIV_64// Setting for Chip select timing divider
};

static const stc_mfs_hl_csio_config_t stcMfsHlCsioCfg = {
  2000000,        // Baud rate
  FALSE,          // LSB first
  TRUE,           // SCK Mark Level Low
  (stc_mfs_hl_spi_config_t *)&stcMfsHlSpiCfg,// SPI configuration (use)
  NULL,           // Serial timer configuration (use timer)
  au8CsioTxBuf,   // Tramsmit FIFO buffer
  au8CsioRxBuf,   // Receive FIFO buffer
  SAMPLE_SPI_TX_BUFFSIZE,  // Size of tramsmit FIFO buffer
  SAMPLE_SPI_RX_BUFFSIZE,  // Size of receive FIFO buffer
  SAMPLE_SPI_RX_BUFF_FILL_LVL,// Unread counts of reception buffer
                  //  to call RX Callback
  MfsCsioMaster,  // Master mode
  MfsCsioActSpiMode,// SPI mode
  MfsSyncWaitZero,// Non wait time insersion
  MfsEightBits,   // 8 data bits
  MfsHlUseNoFifo, // MFS FIFO is not used
  NULL,           // Callback for RX isn't used (unuse interrupt)
  NULL            // Callback for TX isn't used (unuse interrupt)
};
...
...
```

```
function
{
  uint8_t         u8RxData;
  uint8_t         u8TxData;
  ...

  // Set CSIO Ch6_1 Port (SIN, SOT, SCK, SCS)
  FM4_GPIO->PFR0   = FM4_GPIO->PFR0 | 0x7800;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x002A0000;
  FM4_GPIO->EPFR16 = FM4_GPIO->EPFR16 | 0x00000002;

  // Initialize the MFS ch.6 as CSIO(SPI master mode, use CS function)
  if (Ok != Mfs_Hl_Csio_Init(&MFS6, (stc_mfs_hl_csio_config_t
*)&stcMfsHlCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // some code here ...

  while(1)
  {
    // Write and read data synchronously (blocking)
    if (Ok ==Mfs_Hl_Csio_SynchronousTrans(&MFS6, &u8TxData, &u8RxData,
1, FALSE))
    {
      // some code here ...
    }
  }
}
```

CSIO SPI Master Mode with Interrupt and CS Control

This sample software excerpt shows an usage of the CSIO SPI for a master with using interrupt and CS control.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_SPI_TX_BUFFSIZE       (64)
#define SAMPLE_SPI_RX_BUFFSIZE       (64)
#define SAMPLE_SPI_RX_BUFF_FILL_LVL  (1)
...
static void SampleMfsRxCallback(uint16_t u16RxBufFill);
static void SampleMfsTxCallback(uint16_t u16TxCnt);
...
static uint8_t au8CsioTxBuf[SAMPLE_SPI_TX_BUFFSIZE];
static uint8_t au8CsioRxBuf[SAMPLE_SPI_RX_BUFFSIZE];
static volatile uint16_t u16RxBufFillCnt;

Configuration Structure of SPI Serial Chip Select is same as here

static const stc_mfs_hl_csio_config_t stcMfsHlCsioCfg = {
  2000000,                 // Baud rate
  FALSE,                   // LSB first
  TRUE,                    // SCK Mark Level Low
  (stc_mfs_hl_spi_config_t *)&stcMfsHlSpiCfg,// SPI configuration (use)
  NULL,                    // Serial timer configuration (use timer)
  au8CsioTxBuf,            // Tramsmit FIFO buffer
  au8CsioRxBuf,            // Receive FIFO buffer
  SAMPLE_SPI_TX_BUFFSIZE,  // Size of tramsmit FIFO buffer
  SAMPLE_SPI_RX_BUFFSIZE,  // Size of receive FIFO buffer
  SAMPLE_SPI_RX_BUFF_FILL_LVL,// Unread counts of reception buffer
                           // to call RX Callback
  MfsCsioMaster,           // Master mode
  MfsCsioActSpiMode,       // SPI mode
  MfsSyncWaitZero,         // Non wait time insersion
  MfsEightBits,            // 8 data bits
  MfsHlUseNoFifo,          // MFS FIFO is not used
  SampleMfsRxCallback,     // Callback for RX isn't used (unuse
                              interrupt)
  SampleMfsTxCallback      // Callback for TX isn't used (unuse
                              interrupt)
};
...
SampleMfsRxCallback() and SampleMfsTxCallback() are same as here.
...
```

```
static en_result_t SampleMfsSpiReadWrite(uint8_t*     pu8TxBuf,
                                          uint16_t      u16WriteCnt,
                                          uint8_t*      pu8RxBuf,
                                          uint16_t*     pu16ReadCnt
                                          )
{
  uint8_t         au8CsioRxDummyBuf[SAMPLE_SPI_RX_BUFFSIZE];
  en_result_t     enResult;
  uint16_t        u16ReadCnt;

  // If Rx buffer specified NULL ...
  if (NULL == pu8RxBuf)
  {
    // Use internal buffer for dummy reading
    pu8RxBuf = au8CsioRxDummyBuf;
  }
  // Write transmit data (blocking)
  enResult = Mfs_Hl_Write(&MFS6, pu8TxBuf, u16WriteCnt, TRUE,
FALSE);
  if ((Ok == enResult) && (0 != u16WriteCnt))
  {
    // Wait to receive transmitted bytes.
    while (u16WriteCnt > u16RxBufFillCnt);
    do
    {
     // Read received data

enResult=Mfs_Hl_Read(&MFS6,pu8RxBuf,&u16ReadCnt,u16RxBufFillCnt,FALS
E);
    // If TX operation is active, RX is tried.
    } while (ErrorOperationInProgress == enResult);
    if (Ok == enResult)
    {
      if (NULL != pu16ReadCnt)
      {
        // Set the received counts
        *pu16ReadCnt = u16ReadCnt;
      }
      // Update fill count of received buffer
      __disable_irq();
      u16RxBufFillCnt -= u16ReadCnt;
      __enable_irq();
    }
  }

  return (enResult);
}
```

```
function
{
  ...
  uint16_t        u16ReadCnt;
  uint8_t         u8RxData;
  uint8_t         u8TxData;

  // Set CSIO Ch6_1 Port (SIN, SOT, SCK, SCS)
  FM4_GPIO->PFR0   = FM4_GPIO->PFR0 | 0x7800;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x002A0000;
  FM4_GPIO->EPFR16 = FM4_GPIO->EPFR16 | 0x00000002;

  // Clear the filled count of reception buffer
  u16RxBufFillCnt = 0;

  // Initialize the MFS ch.6 as CSIO(SPI master mode, use CS function)
  if (Ok != Mfs_Hl_Csio_Init(&MFS6, (stc_mfs_hl_csio_config_t
*)&stcMfsHlCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // some code here ...

  while(1)
  {
    // Write and read data synchronously (blocking)
    if (Ok == SampleMfsSpiReadWrite(&u8TxData, 1, &u8RxData,
&u16ReadCnt))
    {
      // some code here ...
    }
  }
}
```

CSIO SPI Slave Mode with Interrupt and without CS Control

This example software excerpt shows an usage of the CSIO driver library for a SPI slave with using interrupt and without using CS control.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_SPI_TX_BUFFSIZE       (64)
#define SAMPLE_SPI_RX_BUFFSIZE       (64)
#define SAMPLE_SPI_RX_BUFF_FILL_LVL  (1)
...
static void SampleMfsRxCallback(uint16_t u16RxBufFill);
static void SampleMfsTxCallback(uint16_t u16TxCnt);
...
static uint8_t au8CsioTxBuf[SAMPLE_SPI_TX_BUFFSIZE];
static uint8_t au8CsioRxBuf[SAMPLE_SPI_RX_BUFFSIZE];
static volatile uint16_t u16RxBufFillCnt;

static const stc_mfs_hl_csio_config_t stcMfsHlCsioCfg = {
  2000000,                   // Baud rate
  FALSE,                     // LSB first
  TRUE,                      // SCK Mark Level Low
  NULL,                      // SPI configuration (un-use)
  NULL,                      // Serial timer configuration (un-use)
  au8CsioTxBuf,              // Tramsmit FIFO buffer
  au8CsioRxBuf,              // Receive FIFO buffer
  SAMPLE_SPI_TX_BUFFSIZE,    // Size of tramsmit FIFO buffer
  SAMPLE_SPI_RX_BUFFSIZE,    // Size of receive FIFO buffer
  SAMPLE_SPI_RX_BUFF_FILL_LVL,// Unread counts of reception buffer
                             //  to call RX Callback
  MfsCsioSlave,              // Slave mode
  MfsCsioActSpiMode,         // SPI mode
  MfsSyncWaitZero,           // Non wait time insersion
  MfsEightBits,              // 8 data bits
  MfsHlUseNoFifo,            // MFS FIFO is not used
  SampleMfsRxCallback,       // Callback for RX is used (use interrupt)
  SampleMfsTxCallback        // Callback for TX is used (use interrupt)
};
...
SampleMfsRxCallback()and SampleMfsTxCallback()are same as here. And
SampleMfsSpiReadWrite() is same as 12.5.1.9 here.
...
```

```
function
{
  uint8_t          au8RxData[64];
  uint16_t         u16ReadCnt;
  uint8_t          u8TxData;

  // Set CSIO Ch6_1 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR0   = FM4_GPIO->PFR0 | 0x3800;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x002A0000;

  // Clear the filled count of reception buffer
  u16RxBufFillCnt = 0;

  // Initialize the MFS ch.6 as CSIO(SPI slave mode)
  if (Ok != Mfs_Hl_Csio_Init(&MFS6, (stc_mfs_hl_csio_config_t
*)&stcMfsHlCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // some code here ...

  while(1)
  {
    // Write and read data synchronously (blocking)
    if (Ok == SampleMfsSpiReadWrite(&u8TxData, 1, au8RxData,
&u16ReadCnt))
    {
      // some code here ...
    }
  }
}
```

CSIO SPI Slave Mode with Interrupt and CS Control

This example software excerpt shows an usage of the CSIO driver library for a SPI slave with using interrupt and CS control.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_SPI_TX_BUFFSIZE           (64)
#define SAMPLE_SPI_RX_BUFFSIZE           (64)
#define SAMPLE_SPI_RX_BUFF_FILL_LVL      (1)
...
static void SampleMfsRxCallback(uint16_t u16RxBufFill);
static void SampleMfsTxCallback(uint16_t u16TxCnt);
...
static uint8_t au8CsioTxBuf[SAMPLE_SPI_TX_BUFFSIZE];
static uint8_t au8CsioRxBuf[SAMPLE_SPI_RX_BUFFSIZE];
static volatile uint16_t u16RxBufFillCnt;

static const stc_mfs_hl_spi_config_t  stcMfsHlSpiCfg = {
  TRUE,              // Chip select active level : High
  20,                // Chip de-select bit
  0xFF,              // Chip select setup delay
  0xFF,              // Chip select hold delay
  MFS_SCRCR_CDIV_64 // Setting for Chip select timing divider
};

static const stc_mfs_hl_csio_config_t stcMfsHlCsioCfg = {
  2000000,                    // Baud rate
  FALSE,                      // LSB first
  TRUE,                       // SCK Mark Level Low
  (stc_mfs_hl_spi_config_t *)&stcMfsHlSpiCfg,// SPI configuration (use)
  NULL,                       // Serial timer configuration (use timer)
  au8CsioTxBuf,               // Tramsmit FIFO buffer
  au8CsioRxBuf,               // Receive FIFO buffer
  SAMPLE_SPI_TX_BUFFSIZE,     // Size of tramsmit FIFO buffer
  SAMPLE_SPI_RX_BUFFSIZE,     // Size of receive FIFO buffer
  SAMPLE_SPI_RX_BUFF_FILL_LVL,// Unread counts of reception buffer
                              //  to call RX Callback
  MfsCsioSlave,               // Slave mode
  MfsCsioActSpiMode,          // SPI mode
  MfsSyncWaitZero,            // Non wait time insersion
  MfsEightBits,               // 8 data bits
  MfsHlUseNoFifo,             // MFS FIFO is not used
  SampleMfsRxCallback,        // Callback for RX is used (use interrupt)
  SampleMfsTxCallback         // Callback for TX is used (use interrupt)
};
...
SampleMfsRxCallback()and SampleMfsTxCallback()are same as here. And
SampleMfsSpiReadWrite() is same as 12.5.1.9 here.
...
```

```
function
{
  uint8_t          au8RxData[64];
  uint16_t         u16ReadCnt;
  uint8_t          u8TxData;

  // Set CSIO Ch6_1 Port (SIN, SOT, SCK, SCS)
  FM4_GPIO->PFR0    = FM4_GPIO->PFR0 | 0x7800;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x002A0000;
  FM4_GPIO->EPFR16 = FM4_GPIO->EPFR16 | 0x00000002;

  // Clear the filled count of reception buffer
  u16RxBufFillCnt = 0;

  // Initialize MFS ch.6 as CSIO(SPI slave mode, use CS function)
  if (Ok != Mfs_Hl_Csio_Init(&MFS6, (stc_mfs_hl_csio_config_t
*)&stcMfsHlCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // some code here ...

  while(1)
  {
    // Write and read data synchronously (blocking)
    if (Ok == SampleMfsSpiReadWrite(&u8TxData, 1, au8RxData,
&u16ReadCnt))
    {
      // some code here ...
    }
  }
}
```

7.21.5.2 I^2C

This example software shows an usage of the I2C driver library with non-blocking process.

Figure 7-4. Sequence Of Data Writing To A Slave (Ex. Non-Blocking)

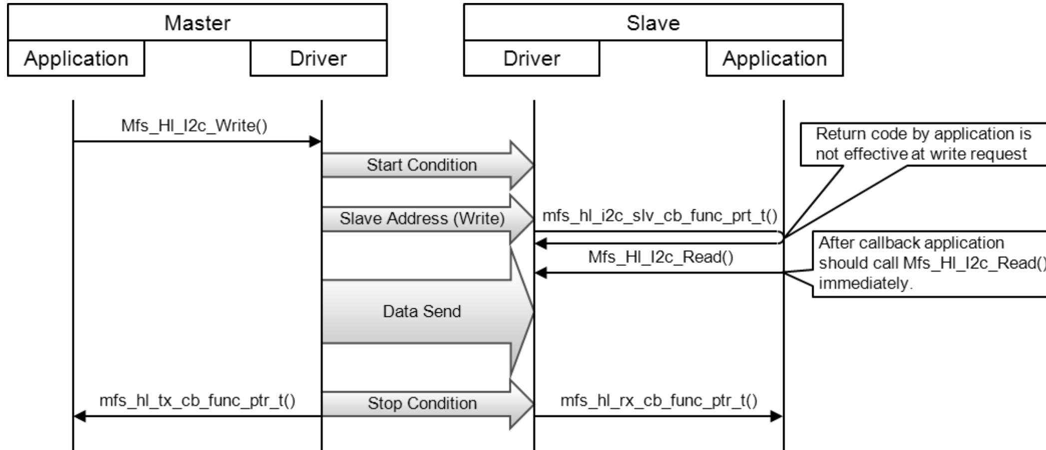
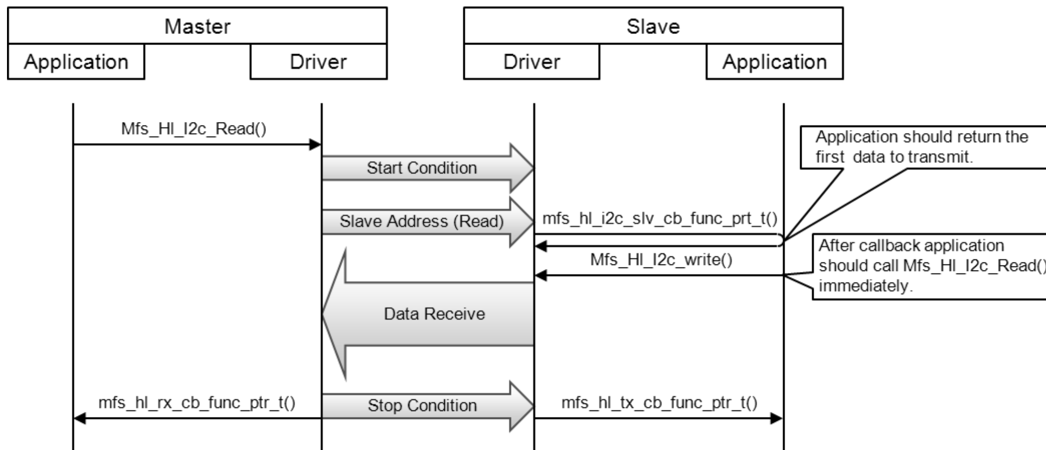


Figure 7-5. Sequence Of Data Reading From A Slave (Ex. Non-Blocking)



I²C Master Mode with Blocking Process

This example software excerpt shows an usage of the I²C driver librry for a master with blocking process.

```
#include "mfs/mfs_hl.h"

...
static const stc_mfs_hl_i2c_config_t stcMfsHlI2cCfg= {
  100000,           // Baud rate
  MfsI2cMaster,     // Master mode
  0,                // Slave address (this is not effective on master)
  MfsI2cDisableFastModePlus, // Disable Fast mode-plus
  MfsHlUseNoFifo, // MFS FIFO is not used
  NULL,             // Callback for RX isn't used
  NULL,             // Callback for TX isn't used
  NULL              // Callback when slave address was detected
                    // from master isn't used (this is for slave)
};

static uint8_t au8TxData[4] = {0x00, 0x01, 0x02, 0x03};
static uint8_t au8RxData[5];
...
```

```
function
{
  uint16_t u16TxRxCnt;
  ...

  // Set I2C Ch2_1 Port (SOT, SCK)
  FM4_GPIO->PFR2    = FM4_GPIO->PFR2 | 0x0060;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00280000;

  // Initialize MFS ch.2 as I2C Master
  if (Ok != Mfs_Hl_I2c_Init(&MFS2, (stc_mfs_hl_i2c_config_t
*)&stcMfsHlI2cCfg))
  {
    // some code here ...
    while(1);
  }

  // some code here ...

  while (1)
  {
    // Send data to slave
    u16TxRxCnt = 4;
    if (Ok == Mfs_Hl_I2c_Write(&MFS2, 0x3E, au8TxData, &u16TxRxCnt,
TRUE))
    {
      // some code here ...

      // Receive data from slave
      u16TxRxCnt = 5;
      if (Ok == Mfs_Hl_I2c_Read(&MFS2, 0x3E, au8RxData, &u16TxRxCnt,
TRUE))
      {
        // some code here ...
      }
      else
      {
        // some code here ...
      }
    }
    else
    {
      // some code here ...
    }
    // some code here ...
  }
}
```

I²C Master Mode with Non-Blocking Process

This example software excerpt shows an usage of the I²C driver library for a master by non-blocking process.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_I2C_STATUS_STBY        (0)
#define SAMPLE_I2C_STATUS_TX          (1)
#define SAMPLE_I2C_STATUS_RX_RQ       (2)
#define SAMPLE_I2C_STATUS_RX          (3)
...
static void SampleMfsI2cRxComplate(uint16_t u16ReceivedCnt);
static void SampleMfsI2cTxComplate(uint16_t u16TxCnt);
...
static const stc_mfs_hl_i2c_config_t stcMfsHlI2cCfg= {
  100000,          // Baud rate
  MfsI2cMaster,    // Master mode
  0,               // Slave address (this is not effective on master)
  MfsI2cDisableFastModePlus// Disable Fast mode-plus
  MfsHlUseNoFifo, // MFS FIFO is not used
  SampleMfsI2cRxComplate,// Callback for RX is used
  SampleMfsI2cTxComplate,// Callback for TX is used
  NULL             // Callback when slave address was detected
                   // from master isn't used (this is for slave)
};

static uint8_t    au8TxData[4] = {0x00, 0x01, 0x02, 0x03};
static uint8_t    au8RxData[5];
static uint16_t   u16TxRxCnt;
static uint8_t    u8I2cState;
...

static void SampleMfsI2cRxComplate(uint16_t u16ReceivedCnt)
{
  // Update the received count in RX buffer
  u16TxRxCnt = u16ReceivedCnt;
}

static void SampleMfsI2cTxComplate(uint16_t u16TxCnt)
{
  // Update the transfered count
  u16TxRxCnt = u16TxCnt;
}
```

```
function
{
  en_result_t enResult;
  ...

  // Set I2C Ch2_1 Port (SOT, SCK)
  FM4_GPIO->PFR2    = FM4_GPIO->PFR2 | 0x0060;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00280000;

  // Clear TX/RX count
  u16TxRxCnt = 0;

  // Initialize MFS ch.2 as I2C Master
  if (Ok != Mfs_Hl_I2c_Init(&MFS2, (stc_mfs_hl_i2c_config_t
*)&stcMfsHlI2cCfg))
  {
    // some code here ...
    while(1);
  }

  // some code here ...

  // Initialize state
  u8I2cState = SAMPLE_I2C_STATUS_STBY;

  while (1)
  {
    switch (u8I2cState)
    {
    case SAMPLE_I2C_STATUS_STBY:
      // some code here ...
      // Send data to slave
      u16TxRxCnt = 4;
      enResult = Mfs_Hl_I2c_Write(&MFS2, 0x3E, au8TxData, &u16TxRxCnt,
FALSE);
      if (Ok == enResult)
      {
        u8I2cState = SAMPLE_I2C_STATUS_TX;
      }
      else
      {
        u8I2cState = SAMPLE_I2C_STATUS_RX_RQ;
        // some code here ...
      }
      break;
    ...
```

```
    ...
     case SAMPLE_I2C_STATUS_TX:
       // Check to complete TX (This is for fail safe)
       enResult = Mfs_Hl_I2c_WaitTxComplete(&MFS2, 10000000);
       if (Ok == enResult)
       {
         // some code here ...
         u8I2cState = SAMPLE_I2C_STATUS_RX_RQ;
       }
       else
       {
         if (ErrorOperationInProgress != enResult)
         {
           u8I2cState = SAMPLE_I2C_STATUS_RX_RQ;
           // some code here ...
         }
       }
       break;
     case SAMPLE_I2C_STATUS_RX_RQ:
       // some code here ...
       // Receive data from slave
       u16TxRxCnt = 5;
       enResult = Mfs_Hl_I2c_Read(&MFS2, 0x3E, au8RxData, &u16TxRxCnt, FALSE);
       if (Ok == enResult)
       {
         u8I2cState = SAMPLE_I2C_STATUS_RX;
       }
       else
       {
         u8I2cState = SAMPLE_I2C_STATUS_STBY;
         // some code here ...
       }
       break;
     case SAMPLE_I2C_STATUS_RX:
       // Check to complete RX (This is for fail safe)
       enResult = Mfs_Hl_I2c_WaitRxComplete(&MFS2, 10000000);
       if (Ok == enResult)
       {
         u8I2cState = SAMPLE_I2C_STATUS_STBY;
         // some code here ...
       }
       else
       {
         if (ErrorOperationInProgress != enResult)
         {
           u8I2cState = SAMPLE_I2C_STATUS_STBY;
           // some code here ...
         }
       }
       break;
     default:
       u8I2cState = SAMPLE_I2C_STATUS_STBY;
       break;
     }
   }
 }
```

I²C Slave Mode with Blocking Process

This example software excerpt shows an usage of the I²C driver library for a slave mode by blocking process.

```
#include "mfs/mfs_hl.h"

...
static uint8_t SampleMfsI2cSlvStCb(uint8_t u8Status);
...
static const stc_mfs_hl_i2c_config_t stcMfsHlI2cCfg= {
  100000,                    // Baud rate (this is not effective on slave)
  MfsI2cSlave,               // Slave mode
  0x3E,                      // Slave address
  MfsI2cDisableFastModePlus, // Disable Fast mode-plus
  MfsHlUseNoFifo,            // MFS FIFO is not used
  NULL,                      // Callback for RX isn't used
  NULL,                      // Callback for TX isn't used
  SampleMfsI2cSlvStCb        // Callback when slave address was detected
                             // from master is used
};

static uint8_t      au8TxData[4] = {0x00, 0x01, 0x02, 0x03};
static uint8_t      au8RxData[5];
static uint16_t     u8I2cStatus;
...
static uint8_t SampleMfsI2cSlvStCb(uint8_t u8Status)
{
  // Initialize return code (for 1st transmit data when request is TX)
  uint8_t u8Data = 0u;
  // Memorize status of the request from master
  u8I2cStatus = u8Status;
  // TX
  if (MfsI2cWrite == u8Status)
  {
    // Set any value to return code ...
    u8Data = ...
  }

  return (u8Data);
}

static uint8_t SampleMfsGetI2cSlvStatus(void)
{
  uint8_t u8Status;

  __disable_irq();
  // Set status of the request from master to the return code
  u8Status = u8I2cStatus;
  // Clear status
  u8I2cStatus = 0xee;
  __enable_irq();

  return (u8Status);
}
```

```
function
{
  uint16_t u16TxRxCnt;

  // Set I2C Ch2_1 Port (SOT, SCK)
  FM4_GPIO->PFR2    = FM4_GPIO->PFR2 | 0x0060;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00280000;

  // Initialize status of the request from master
  u8I2cStatus = 0xee;
  ...
  // Initialize MFS ch.2 as I2C Slave
  if (Ok != Mfs_Hl_I2c_Init(&MFS2, (stc_mfs_hl_i2c_config_t *)&stcMfsHlI2cCfg))
  {
    // some code here ...
    while(1);
  }

  while (1)
  {
    // Get I2C status (check the request from master)
    switch (SampleMfsGetI2cSlvStatus())
    {
      // Write (Request to read from master)
      case MfsI2cWrite:
        // Send data to master
        u16TxRxCnt = 4;
        if (Ok == Mfs_Hl_I2c_Write(&MFS2, 0, au8TxData, &u16TxRxCnt, TRUE))
        {
          // some code here ...
        }
        else
        {
          // some code here ...
        }
        break;
      // Read (Request to write from master)
      case MfsI2cRead:
        // Receive data from master
        u16TxRxCnt = 4;
        if (Ok == Mfs_Hl_I2c_Read(&MFS2, 0, au8RxData, &u16TxRxCnt, TRUE))
        {
          // some code here ...
        }
        else
        {
          // some code here ...
        }
        break;
      default:
        break;
    }
  }
}
```

I²C Slave Mode with Non-Blocking Process

This example software excerpt shows an usage of the I²C driver library for a slave by non-blocking process.

```
#include "mfs/mfs_hl.h"

...
static void SampleMfsI2cRxComplate(uint16_t u16ReceivedCnt);
static void SampleMfsI2cTxComplate(uint16_t u16TxCnt);
static uint8_t SampleMfsI2cSlvStCb(uint8_t u8Status);
...
static const stc_mfs_hl_i2c_config_t stcMfsHlI2cCfg= {
  100000,                      // Baud rate (this is not effective on slave)
  MfsI2cSlave,                 // Slave mode
  0x3E,                        // Slave address
  MfsI2cDisableFastModePlus,   // Disable Fast mode-plus
  MfsHlUseNoFifo,              // MFS FIFO is not used
  SampleMfsI2cRxComplate,      // Callback for RX is used
  SampleMfsI2cTxComplate,      // Callback for TX is used
  SampleMfsI2cSlvStCb          // Callback when slave address was detected
                               // from master is used
};

static uint8_t      au8TxData[4] = {0x00, 0x01, 0x02, 0x03};
static uint8_t      au8RxData[5];
static uint16_t     u16TxRxCnt;
static uint8_t      u8I2cStatus;
static uint8_t      u8TxRxStatus;
...
SampleMfsI2cRxComplate() and SampleMfsI2cTxComplate() are same as here.

SampleMfsI2cSlvStCb() and SampleMfsGetI2cSlvStatus() are same as here.
```

```
function
{
  en_result_t enResult;

  // Set I2C Ch2_1 Port (SOT, SCK)
  FM4_GPIO->PFR2   = FM4_GPIO->PFR2 | 0x0060;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00280000;
  // Initialize status of the request from master
  u8I2cStatus = 0xee;
  // Initialize state
  u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_STANDBY;
  ...
  // Initialize MFS ch.2 as I2C Slave
  if (Ok != Mfs_Hl_I2c_Init(&MFS2, (stc_mfs_hl_i2c_config_t *)&stcMfsHlI2cCfg))
  {
    // some code here ...
    while(1);
  }
  while (1)
  {
    // State is standby
    if (SAMPLE_MFS_I2C_STATUS_STANDBY == u8TxRxStatus)
    {
      // Get I2C status (check the request from master)
      switch (SampleMfsGetI2cSlvStatus())
      {
        // Write (Request to read from master)
        case MfsI2cWrite:
          // Send data to master
          u16TxRxCnt = 4;
          enResult = Mfs_Hl_I2c_Write(&MFS2, 0, au8TxData, &u16TxRxCnt, FALSE);
          if (Ok == enResult)
          {
            // Change state to transmitting
            u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_TRANS;
          }
          // some code here ...
          break;
        // Read (Request to write from master)
        case MfsI2cRead:
          // Receive data from master
          u16TxRxCnt = 4;
          enResult = Mfs_Hl_I2c_Read(&MFS2, 0, au8RxData, &u16TxRxCnt, FALSE);
          if (Ok == enResult)
          {
            // Change state to receiving
            u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_RCV;
          }
          // some code here ...
          break;
        default:
          break;
      }
    }
    ...
```

```
  ...
    // Transmitting or Receiving
    if (SAMPLE_MFS_I2C_STATUS_STANDBY != u8TxRxStatus)
    {
      // Transmitting
      if(SAMPLE_MFS_I2C_STATUS_TRANS == u8TxRxStatus)
      {
        // Check to complete TX (This is for fail safe)
        enResult = Mfs_Hl_I2c_WaitTxComplete(&MFS2, 10000000);
        // Success
        if (Ok == enResult)
        {
          // Change state to standby
          u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_STANDBY;
          // some code here ...
        }
        else
        {
          // Transmission error
          if (ErrorOperationInProgress != enResult)
          {
            // Change state to standby
            u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_STANDBY;
            // some code here ...
          }
        }
      }
      // Receiving
      else
      {
        // Check to complete RX (This is for fail safe)
        enResult = Mfs_Hl_I2c_WaitRxComplete(&MFS2, 10000000);
        // Success
        if (Ok == enResult)
        {
          // Change state to standby
          u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_STANDBY;
          // some code here ...
        }
        else
        {
          // Reception error
          if (ErrorOperationInProgress != enResult)
          {
            // Change state to standby
            u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_STANDBY;
            // some code here ...
          }
        }
      }
    }
  }
}
```

7.21.5.3 LIN

This example software excerpt shows an usage of the LIN driver library for a master and a slave with using interrupt.

```
#include "mfs/mfs_hl.h"

...
#define MFS_LIN_TX_BUFFER_SIZE      (128)
#define MFS_LIN_RX_BUFFER_SIZE      (128)
...
static void SampleMfsLinRxCallback1(uint16_t u16RxBufFill);
static void SampleMfsLinBrkCallback1(void);
static void SampleMfsLinRxCallback2(uint16_t u16RxBufFill);
static void SampleMfsLinBrkCallback2(void);
...
static uint8_t au8TxBuffer1[MFS_LIN_TX_BUFFER_SIZE];
static uint8_t au8RxBuffer1[MFS_LIN_RX_BUFFER_SIZE];
static uint8_t au8TxBuffer2[MFS_LIN_TX_BUFFER_SIZE];
static uint8_t au8RxBuffer2[MFS_LIN_RX_BUFFER_SIZE];
...
static volatile uint16_t   u16RxBufFillCnt1;
static volatile uint16_t   u16RxBufFillCnt2;
static volatile uint8_t             u8Dummy;
static volatile uint8_t             u8LinBrk2;
...
```

```
// Configuration for master
static const stc_mfs_hl_lin_config_t stcMfsHlLinCfg1 = {
    19200,                  // Baud rate
    FALSE,                  // Disable external wake-up
    FALSE,                  // Disable LIN break RX interrupt
    au8TxBuffer1,           // Tramsmit FIFO buffer
    au8RxBuffer1,           // Receive FIFO buffer
    MFS_LIN_TX_BUFFER_SIZE, // Size of tramsmit FIFO buffer
    MFS_LIN_RX_BUFFER_SIZE, // Size of receive FIFO buffer
    MfsLinMaster,           // Master mode
    MfsOneStopBit,          // 1 stop bit
    MfsLinBreakLength16,    // Lin Break Length: 16 bit times
    MfsLinDelimiterLength1, // Lin Break Delimiter Length: 1 bit times
    MfsHlUseNoFifo,         // MFS FIFO is not used
    SampleMfsLinRxCallback1,// Callback for RX is used
    NULL,                   // Callback for TX isn't used
    SampleMfsLinBrkCallback1// Callback for LIN break detection is used
};

// Configuration for slave
static const stc_mfs_hl_lin_config_t stcMfsHlLinCfg2 = {
    19200,                  // Baud rate
    FALSE,                  // Disable external wake-up
    TRUE,                   // Enable LIN break RX interrupt
    au8TxBuffer2,           // Tramsmit FIFO buffer
    au8RxBuffer2,           // Receive FIFO buffer
    MFS_LIN_TX_BUFFER_SIZE, // Size of tramsmit FIFO buffer
    MFS_LIN_RX_BUFFER_SIZE, // Size of receive FIFO buffer
    MfsLinSalve,            // Slave mode
    MfsOneStopBit,          // 1 stop bit
    MfsLinBreakLength16,    // Lin Break Length: 16 bit times
    MfsLinDelimiterLength1, // Lin Break Delimiter Length: 1 bit times
    MfsHlUseNoFifo,         // MFS FIFO is not used
    SampleMfsLinRxCallback2,// Callback for RX is used
    NULL,                   // Callback for TX isn't used
    SampleMfsLinBrkCallback2// Callback for LIN break detection is used
};
```

```
// Only for example! Do not use this in your application!

static void SampleMfsLinRxCallback1(uint16_t u16RxBufFill)
{
  // Memorize filled count of reception buffer
  u16RxBufFillCnt1 = u16RxBufFill;
}

static void SampleMfsLinBrkCallback1(void)
{
  // some code here ...
}

static void SampleMfsLinRxCallback2(uint16_t u16RxBufFill)
{
  // Memorize filled count of reception buffer
  u16RxBufFillCnt2 = u16RxBufFill;
}

static void SampleMfsLinBrkCallback2(void)
{
  // LIN break detected
  u8LinBrk2 = TRUE;
}

static void SampleMfsLinWait(volatile uint32_t u32WaitCount)
{
  while(u32WaitCount--);
}
```

```
static void SampleMfsLin(void)
{
  uint16_t u16ReadCount1;
  uint16_t u16ReadCount2;

  // Temporarily disable Slave RX interrupt. Will be reenabled by Mfs_Read()
  Mfs_Hl_Lin_DisableRxInterrupt(&MFS6);
  u8LinBrk2 = FALSE;
  // LIN Master Task
  au8WriteBuffer1[0] = 0x55;      // Synch Field
  au8WriteBuffer1[1] = 'M';       // Header (no LIN meaning, just test byte)
  au8WriteBuffer1[2] = 'S';       // Data (no LIN meaning, just test byte)
  au8WriteBuffer1[3] = 'T';       // Checksum (no LIN meaning, just test byte)
  u16RxBufFillCnt2 = 0;
  // First Set LIN Break
  Mfs_Hl_Lin_SetBreak(&MFS0);
  while (FALSE == u8LinBrk2);     // wait for break received by slave
  // Prepare Read LIN Slave
  Mfs_Hl_Read(&MFS6, au8ReadBuffer2, &u16ReadCount2, 4, FALSE);
  // Write rest of LIN Frame
  Mfs_Hl_Write(&MFS0, au8WriteBuffer1, 4, FALSE, FALSE);
  // Wait for all data read in MFS6 (slave)
  while (u16RxBufFillCnt2 < 4);
  // Transfer LIN slave data from internal buffer to au8ReadBuffer2
  Mfs_Hl_Read(&MFS6, au8ReadBuffer2, &u16ReadCount2, u16RxBufFillCnt2, FALSE);

  // Wait time for next frame (usually done by scheduler timer)
  SampleMfsLinWait(20000);
  // Temporarily disable Slave RX interrupt. Will be reenabled by Mfs_Read()
  Mfs_Hl_Lin_DisableRxInterrupt(&MFS6);
  u8LinBrk2 = FALSE;
  // LIN Slave Task
  au8WriteBuffer1[0] = 0x55;      // Synch Field
  au8WriteBuffer1[1] = 'S';       // Header (no LIN meaning, just test byte)
  au8WriteBuffer2[0] = 'L';       // Data (no LIN meaning, just test byte)
  au8WriteBuffer2[1] = 'V';       // Checksum (no LIN meaning, just test byte)
  u16RxBufFillCnt2 = 0;
  // First Set LIN Break
  Mfs_Hl_Lin_SetBreak(&MFS0);
  while (FALSE == u8LinBrk2);     // wait for break received by slave
  // Prepare Read LIN Slave
  Mfs_Hl_Read(&MFS6, au8ReadBuffer2, &u16ReadCount2, 4, FALSE);
  // Write synch field and header of LIN Frame
  Mfs_Hl_Write(&MFS0, au8WriteBuffer1, 2, FALSE, FALSE);
  // Wait for all data read in MFS6 (slave)
  while (u16RxBufFillCnt2 < 2);
  // Transfer LIN slave data from internal buffer to au8ReadBuffer2
  Mfs_Hl_Read(&MFS6, au8ReadBuffer2, &u16ReadCount2, u16RxBufFillCnt2, FALSE);
  u16RxBufFillCnt1 = 0;
  // Write rest of LIN Frame
  Mfs_Hl_Write(&MFS6, au8WriteBuffer2, 2, FALSE, FALSE);
  // Wait for all data read back in MFS0 (master)
  while (u16RxBufFillCnt1 < 2);
  // Transfer LIN master data from internal buffer to au8ReadBuffer1
  Mfs_Hl_Read(&MFS0,au8ReadBuffer1 + 2,&u16ReadCount1,u16RxBufFillCnt1,FALSE);
}
```

```
function
{
  en_result_t enResult;

  // Disable Analog input (P21:SIN0_0/AN17, P22:SOT0_0/AN16)
  FM4_GPIO->ADE = 0;

  // Set LIN Ch0_0 Port (SIN, SOT)
  FM4_GPIO->PFR2 = FM4_GPIO->PFR2 | 0x0006;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00000040;
  // Set LIN Ch6_0 Port (SIN, SOT)
  FM4_GPIO->PFR5 = FM4_GPIO->PFR5 | 0x0060;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x00050000;

  // some code here ...

  // Initialize the MFS ch.0 as LIN master
  enResult = Mfs_Hl_Lin_Init(&MFS0, (stc_mfs_hl_lin_config_t
*)&stcMfsHlLinCfg1);
  if (Ok == enResult)
  {
    // Initialize the MFS ch.6 as LIN slave
    enResult = Mfs_Hl_Lin_Init(&MFS6, (stc_mfs_hl_lin_config_t
*)&stcMfsHlLinCfg2);
    if (Ok != enResult)
    {
      // some code here ...
    }
  }

  // Initialization is faled ...
  if (Ok != enResult)
  {
    // some code here ...
    while(1);
  }

  // If initilazation is successful, LIN master and slave activation
sample is performed.
  SampleMfsLin();

  while (1);
}
```

7.21.5.4 UART

UART example software only offers nomal mode usage.

UART with interrupt

This example software excerpt shows an usage of the UART driver library with using interrupt.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_UART_TX_BUFFSIZE      (128)
#define SAMPLE_UART_RX_BUFFSIZE      (256)
#define SAMPLE_UART_RX_BUFF_FILL_LVL          (1)
...
static uint8_t au8UartTxBuf[SAMPLE_UART_TX_BUFFSIZE];
static uint8_t au8UartRxBuf[SAMPLE_UART_RX_BUFFSIZE];

static const stc_mfs_hl_uart_config_t stcMfsHlUartCfg = {
  115200,          // Baud rate
  FALSE,           // LSB first
  FALSE,           // NRZ
  FALSE,           // Not use Hardware Flow
  au8UartTxBuf,    // Tramsmit FIFO buffer
  au8UartRxBuf,    // Receive FIFO buffer
  SAMPLE_UART_TX_BUFFSIZE, // Size of tramsmit FIFO buffer
  SAMPLE_UART_RX_BUFFSIZE, // Size of receive FIFO buffer
  SAMPLE_UART_RX_BUFF_FILL_LVL,// Unread counts of reception buffer
                   //  to call RX Callback
  MfsUartNormal,   // Normal mode
  MfsParityNone,   // Non parity
  MfsOneStopBit,   // 1 stop bit
  MfsEightBits,    // 8 data bits
  MfsHlUseNoFifo,  // MFS FIFO is not used
  NULL,            // Callback for RX isn't used (unuse interrupt)
  NULL             // Callback for TX isn't used (unuse interrupt)
};
...
```

```
function
{
  uint8_t au8ReadBuf[SAMPLE_UART_RX_BUFFSIZE];
  uint16_t        u16ReadCnt;

  // Disable Analog input (P21:SIN0_0/AN17, P22:SOT0_0/AN16)
  FM4_GPIO->ADE = 0;

  // Set UART Ch0_0 Port (SIN, SOT)
  FM4_GPIO->PFR2 = FM4_GPIO->PFR2 | 0x0006;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00000040;

  // Initialize the MFS ch.0 as UART
  if (Ok != Mfs_Hl_Uart_Init(&MFS0, (stc_mfs_hl_uart_config_t
*)&stcMfsHlUartCfg))
  {
    // some code here ...
    while(1);
  }

  // some code here ...

  while(1)
  {
    // Receive data from UART asynchrnously (Non-blocking)
    Mfs_Hl_Read(&MFS0, au8ReadBuf, &u16ReadCnt,
                SAMPLE_UART_RX_BUFFSIZE, FALSE);
    // If data is received from UART,
    if (0 < u16ReadCnt)
    {
      // Send received data to UART (Echo)
      Mfs_Hl_Write(&MFS0, au8ReadBuf, u16ReadCnt, FALSE, FALSE);
    }
  }
}
```

UART without interrupt

This example software excerpt shows an usage of the UART driver library without using interrupt.

```
#include "mfs/mfs_hl.h"

#define SAMPLE_UART_TX_BUFFSIZE      (128)
#define SAMPLE_UART_RX_BUFFSIZE      (256)
#define SAMPLE_UART_RX_BUFF_FILL_LVL          (1)
...
static void SampleMfsRxCallback(uint16_t u16RxBufFill);
static void SampleMfsTxCallback(uint16_t u16TxCnt);
...
static uint8_t             au8UartTxBuf[SAMPLE_UART_TX_BUFFSIZE];
static uint8_t             au8UartRxBuf[SAMPLE_UART_RX_BUFFSIZE];
static volatile uint16_t   u16RxBufFillCnt;
static volatile uint8_t    u8UartTxInt;

static const stc_mfs_hl_uart_config_t stcMfsHlUartCfg = {
  115200,                  // Baud rate
  FALSE,                   // LSB first
  FALSE,                   // NRZ
  FALSE,                   // Not use Hardware Flow
  au8UartTxBuf,            // Tramsmit FIFO buffer
  au8UartRxBuf,            // Receive FIFO buffer
  SAMPLE_UART_TX_BUFFSIZE, // Size of tramsmit FIFO buffer
  SAMPLE_UART_RX_BUFFSIZE, // Size of receive FIFO buffer
  SAMPLE_UART_RX_BUFF_FILL_LVL,// Unread counts of reception buffer
                           //  to call RX Callback
  MfsUartNormal,           // Normal mode
  MfsParityNone,           // Non parity
  MfsOneStopBit,           // 1 stop bit
  MfsEightBits,            // 8 data bits
  MfsHlUseNoFifo,          // MFS FIFO is not used
  SampleMfsRxCallback,     // Callback for RX is used (use interrupt)
  SampleMfsTxCallback      // Callback for TX is used (use interrupt)
};
...
```

```
static void SampleMfsRxCallback(uint16_t u16RxBufFill)
{
  // Update the filled count in RX buffer
  u16RxBufFillCnt = u16RxBufFill;
}

static void SampleMfsTxCallback(uint16_t u16TxCnt)
{
  // Set the TX completed flag
  u8UartTxInt = TRUE;
}

static en_result_t SampleMfsUartRead(uint8_t*               pu8RxBuf,
                                   uint16_t*      pu16ReadCnt
                                  )
{
  en_result_t enResult;

  // Read received data (non-blocking)
  enResult = Mfs_Hl_Read(&MFS0, pu8RxBuf, pu16ReadCnt, u16RxBufFillCnt, FALSE);
  if (0 != *pu16ReadCnt)
  {
    // Update fill count of received buffer
    __disable_irq();
    u16RxBufFillCnt -= *pu16ReadCnt;
    __enable_irq();
  }

  return (enResult);
}

static en_result_t SampleMfsUartWrite(uint8_t* pu8TxBuf,
                                      uint16_t u16WriteCnt
                                     )
{
  en_result_t enResult;

  // Write transmit data (non-blocking)
  enResult = Mfs_Hl_Write(&MFS0, pu8TxBuf, u16WriteCnt, FALSE, FALSE);
  if ((Ok == enResult) && (0 != u16WriteCnt))
  {
      // Wait to complete transmission
      while (FALSE == u8UartTxInt);
      // Clear the TX completed flag
      u8UartTxInt = FALSE;
  }

  return (enResult);
}
```

```
function
{
  uint8_t  au8ReadBuf[SAMPLE_UART_RX_BUFFSIZE];
  uint16_t u16ReadCnt;

  // Disable Analog input (P21:SIN0_0/AN17, P22:SOT0_0/AN16)
  FM4_GPIO->ADE = 0;

  // Set UART Ch0_0 Port (SIN, SOT)
  FM4_GPIO->PFR2 = FM4_GPIO->PFR2 | 0x0006;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00000040;

  // Initialize the filled count of RX buffer
  u16RxBufFillCnt = 0;
  // Initialize the TX completed flag
  u8UartTxInt = FALSE;

  // Initialize the MFS ch.0 as UART
  if (Ok != Mfs_Hl_Uart_Init(&MFS0, (stc_mfs_hl_uart_config_t
*)&stcMfsHlUartCfg))
  {
    // some code here ...
    while(1);
  }

  // some code here ...

  while(1)
  {
    // Receive data from UART asynchrnously (Non-blocking)
    SampleMfsUartRead(au8ReadBuf, &u16ReadCnt);;
    // If data is received from UART,
    if (0 < u16ReadCnt)
    {
      // Send received data to UART (Echo)
      SampleMfsUartWrite(au8ReadBuf, u16ReadCnt);
    }
  }
}
```

7.22 (MFS) Multi Function Serial Interface

Type Definition	`stc_mfsn_t`
Configuration Type	`stc_mfs_uart_config_t`, `stc_mfs_csio_config_t`, `stc_mfs_lin_config_t`, `stc_mfs_i2c_config_t`
Address Operator	MFS*n*

The MFS driver library APIs are register access unlike `MFS_HL`.

`Mfs_Uart_Init()` initializes an MFS instance to the UART with `pstcConfig` in `stc_mfs_uart_config_t`.

`Mfs_Uart_DeInit()` deinitilizes all of the MFS UART registers.

`Mfs_Csio_Init()` initializes an MFS instance to the CSIO with `pstcConfig` in stc_mfs_csio_config_t.

`Mfs_Csio_DeInit()` deinitializes all of the MFS CSIO registers.

`Mfs_Lin_Init()` initializes an MFS instance to the LIN with given LIN configuration (`stc_mfs_lin_config_t`).

`Mfs_Lin_DeInit()` deinitializes all of the MFS LIN registers.

`Mfs_I2c_Init()` initializes an MFS instance to the I^2C with `pstcConfig` in `stc_mfs_i2c_config_t`.

`Mfs_I2c_DeInit()` deinitializes all of MFS I^2C registers.

`Mfs_SetRxIntCallBack()`, `Mfs_SetTxIntCallBack()` and `Mfs_SetStsIntCallBack()` register callback function to be called when the interrupt is generated.

`Mfs_SetUpperLayerHandle()` registers a pointer to the internal data for upper layer software.

The other functions are used for reading or setting registers, etc.

7.22.1 MFS Configuration Structure

The MFS driver library uses four structures of configuration for the UART, CSIO, I^2C and LIN. And this library also uses structures of configuration for the CSIO serial chip select and FIFO.

7.22.1.1 UART Configuration Structure

The MFS UART driver library uses `stc_mfs_uart_config_t`:

Type	Field	Possible Values	Description
uint32_t	u32DataRate	-	Baud rate (bps)
uint8_t	u8UartMode	MfsUartNormal MfsUartMulti	Normal mode Multi-Processor Mode
uint8_t	u8Parity	MfsParityNone MfsParityEven MfsParityOdd	No parity bit is used Even parity bit is used Odd parity bit is used
uint8_t	u8StopBit	MfsOneStopBit MfsTwoStopBits MfsThreeStopBits MfsFourStopBits	1 Stop Bit 2 Stop Bits 3 Stop Bits 4 Stop Bits
uint8_t	u8CharLength	MfsFiveBits MfsSixBits MfsSevenBits MfsEightBits MfsNineBits	5 Bits character length 6 Bits character length 7 Bits character length 8 Bits character length 9 Bits character length
boolean_t	bBitDirection	FALSE TRUE	LSB first MSB first
boolean_t	bSignalSystem	FALSE TRUE	NRZ Inverted NRZ
boolean_t	bHwFlow	FALSE TRUE	Hardware Flow is not used Hardware Flow is used

7.22.1.2 CSIO Configuration Structure

The MFS CSIO driver library uses `stc_mfs_csio_config_t`:

Type	Field	Possible Values	Description
uint32_t	u32DataRate	-	Baud rate (bps)
uint8_t	u8CsioMode	MfsCsioMaster MfsCsioSlave	Master mode Slave mode
uint8_t	u8CsioActMode	MfsCsioActNormalMode MfsCsioActSpiMode	Normal mode SPI mode
uint8_t	u8SyncWaitTime	MfsSyncWaitZero MfsSyncWaitOne MfsSyncWaitTwo MfsSyncWaitThree	0 wait time insertion 1 wait time insertion 2 wait time insertion 3 wait time insertion
uint8_t	u8CharLength	MfsFiveBits MfsSixBits MfsSevenBits MfsEightBits MfsNineBits MfsTenBits MfsElevenBits MfsTwelveBits MfsThirteenBits MfsFourteenBits MfsFifteenBits MfsSixteenBits MfsTwentyBits MfsTwentyFourBits MfsThirtyTwoBits	5 Bits character length 6 Bits character length 7 Bits character length 8 Bits character length 9 Bits character length 10 Bits character length 11 Bits character length 12 Bits character length 13 Bits character length 14 Bits character length 15 Bits character length 16 Bits character length 20 Bits character length 24 Bits character length 32 Bits character length
boolean_t	bBitDirection	FALSE TRUE	LSB first MSB first
boolean_t	bSignalSystem	FALSE TRUE	SCK Mark Level High SCK Mark Level Low

7.22.1.3 I2C Configuration Structure

The MFS I2C driver library uses `stc_mfs_i2c_config_t`:

Type	Field	Possible Values	Description
uint32_t	u32DataRate	-	Baud rate (bps)
uint8_t	u8I2cMode	MfsI2cMaster MfsI2cSlave	Master mode Slave mode
uint8_t	u8NoizeFilter	MfsI2cNoizeFilterLess40M MfsI2cNoizeFilterLess60M MfsI2cNoizeFilterLess80M MfsI2cNoizeFilterLess100M MfsI2cNoizeFilterLess120M MfsI2cNoizeFilterLess140M MfsI2cNoizeFilterLess160M MfsI2cNoizeFilterLess180M MfsI2cNoizeFilterLess200M MfsI2cNoizeFilterLess220M MfsI2cNoizeFilterLess240M MfsI2cNoizeFilterLess260M MfsI2cNoizeFilterLess280M MfsI2cNoizeFilterLess300M MfsI2cNoizeFilterLess320M MfsI2cNoizeFilterLess340M MfsI2cNoizeFilterLess360M MfsI2cNoizeFilterLess380M MfsI2cNoizeFilterLess400M	8MHz <= APB1 bus clock < 40MHz 40MHz <= APB1 bus clock < 60MHz 60MHz <= APB1 bus clock < 80MHz 80MHz <= APB1 bus clock < 100MHz 100MHz <= APB1 bus clock < 120MHz 120MHz <= APB1 bus clock < 140MHz 140MHz <= APB1 bus clock < 160MHz 160MHz <= APB1 bus clock < 180MHz 180MHz <= APB1 bus clock < 200MHz 200MHz <= APB1 bus clock < 220MHz 220MHz <= APB1 bus clock < 240MHz 240MHz <= APB1 bus clock < 260MHz 260MHz <= APB1 bus clock < 280MHz 280MHz <= APB1 bus clock < 300MHz 300MHz <= APB1 bus clock < 320MHz 320MHz <= APB1 bus clock < 340MHz 340MHz <= APB1 bus clock < 360MHz 360MHz <= APB1 bus clock < 380MHz 380MHz <= APB1 bus clock < 400MHz
uint8_t	u8SlvAddr	0x00 - 0x7F	Slave address
uint8_t	u8FastMode	MfsI2cDisableFastModePlus MfsI2cEnableFastModePlus	Standard-mode Fast-mode Plus

7.22.1.4 LIN Configuration Structure

The MFS LIN driver library uses `stc_mfs_lin_config_t`:

Type	Field	Possible Values	Description
uint32_t	u32DataRate	-	Baud rate (bps)
boolean_t	bExtWakeUp	FALSE TRUE	Disable external wake-up Enable external wake-up
boolean_t	bLinBreakIrqEnable	FALSE TRUE	Disable LIN break RX interrupt Enable LIN break RX interrupt
uint8_t	u8LinMode	MfsLinMaster MfsLinSlave	Master mode Slave mode
uint8_t	u8StopBits	MfsLinOneStopBit MfsLinTwoStopBits MfsLinThreeStopBits MfsLinFourStopBits	1 Stop Bit 2 Stop Bits 3 Stop Bits 4 Stop Bits
uint8_t	u8BreakLength	MfsLinBreakLength13 MfsLinBreakLength14 MfsLinBreakLength15 MfsLinBreakLength16	Lin Break Length 13 Bit Times Lin Break Length 14 Bit Times Lin Break Length 15 Bit Times Lin Break Length 16 Bit Times
uint8_t	u8DelimiterLength	MfsLinDelimiterLength1 MfsLinDelimiterLength2 MfsLinDelimiterLength3 MfsLinDelimiterLength4	Lin Break Delimiter Length 1 Bit Time Lin Break Delimiter Length 2 Bit Times Lin Break Delimiter Length 3 Bit Times Lin Break Delimiter Length 4 Bit Times

7.22.1.5 CSIO Serial Chip Select Timing Configuration Structure

The MFS CSIO driver library uses `stc_mfs_csio_cs_timing_t` for chip select setup.
This structure is used when the CSIO is used as a SPI mode:

Type	Field	Possible Values	Description
uint8_t	u8CsSetupDelay	-	Timing for chip select setup delay
uint8_t	u8CsHoldDelay	-	Timing for chip select hold delay
uint16_t	u16CsDeSelect	-	Minimum time from inactivation until activation of chip select

7.22.1.6 MFS FIFO Configuration Structure

The MFS driver library uses `stc_mfs_fifo_config_t` for FIFO setup:

Type	Field	Possible Values	Description
uint8_t	u8FifoSel	MfsTxFifo1RxFifo2 MfsTxFifo2RxFifo1	TX FIFO:FIFO1, RX FIFO:FIFO2 TX FIFO:FIFO2, RX FIFO:FIFO1
uint8_t	u8ByteCount1	*	Transfer data count for FIFO1
uint8_t	u8ByteCount2	*	Transfer data count for FIFO2

7.22.2 API Reference

7.22.2.1 *Mfs_Uart_Init()*

Initializes the MFS block to UART.

Prototype	
`en_result_t Mfs_Uart_Init(volatile stc_mfsn_t* pstcUart, const stc_mfs_uart_config_t* pstcConfig)`	
Parameter Name	**Description**
[in] `pstcUart`	A pointer to the MFS instance
[in] `pstcConfig`	A pointer to a structure of the UART configuration
Return Values	**Description**
`Ok`	Initialization ended with no error

`ErrorInvalidParameter`	`pstcUart == NULL`
	`pstcConfig == NULL`
	`pstcMfsInternData == NULL` (invalid or disabled MFS unit)
	Parameter out of range

7.22.2.2 *Mfs_Uart_DeInit()*

Deinitializes the UART block.

Prototype	
`en_result_t Mfs_Uart_DeInit(volatile stc_mfsn_t* pstcUart)`	
Parameter Name	**Description**
[in] `pstcUart`	A Pointer to the MFS instance
Return Values	**Description**
`Ok`	Deinitialization ended with no error
`ErrorInvalidParameter`	`pstcUart == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.3 *Mfs_Uart_SetBaudRate()*

Sets the baud rate of the UART block.

Prototype	
`en_result_t Mfs_Uart_SetBaudRate(volatile stc_mfsn_t* pstcUart, uint32_t u32BaudRate)`	
Parameter Name	**Description**
[in] `pstcUart`	A pointer to the MFS instance
[in] `u32BaudRate`	Baud rate (bps)
Return Values	**Description**
`Ok`	Setting baud rate ended with no error
`ErrorInvalidParameter`	`pstcUart == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)
`ErrorUninitialized`	Baud rate was not set properly

7.22.2.4 *Mfs_Csio_Init()*

Initializes the MFS block to CSIO. The CSIO block can be initialized to one of the following modes.

CSIO normal master mode

CSIO normal master mode using serial timer

CSIO normal slave mode

CSIO SPI master mode without chip select

CSIO SPI master mode with chip select (Chip select is available only channel 6,7)

CSIO SPI slave mode without chip select

CSIO SPI slave mode with chip select (Chip select is available only channel 6,7)

Prototype	
`en_result_t Mfs_Csio_Init(volatile stc_mfsn_t* pstcCsio, const stc_mfs_csio_config_t* pstcConfig)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `pstcConfig`	A pointer to a structure of the CSIO configuration
Return Values	**Description**
`Ok`	Initialization ended with no error

`ErrorInvalidParameter`	`pstcCsio == NULL`
	`pstcConfig == NULL`
	`pstcMfsInternData == NULL` (invalid or disabled MFS unit)
	Parameter out of range

7.22.2.5 *Mfs_Csio_DeInit()*

Deinitializes the CSIO block.

Prototype	
`en_result_t Mfs_Csio_DeInit(volatile stc_mfsn_t* pstcCsio)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
Return Values	**Description**
`Ok`	Deinitialization ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.6 *Mfs_Csio_SetSckOutEnable()*

Enables or disables serial clock output signal of the CSIO block.

Prototype	
`en_result_t Mfs_Csio_SetSckOutEnable(volatile stc_mfsn_t* pstcCsio, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable serial clock output `FALSE`: Disable serial clock output
Return Values	**Description**
`Ok`	Enabling/Disabling serial clock output ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.7 *Mfs_Csio_ReadData32()*

Reads data from Receve Data Register (`RDR`) and returns.
Note: This function accesses `RDR` by 32-bits. If `RDR` is accessed by less than or equal to 16-bits, `Mfs_ReadData()` is used.

Prototype	
`uint32_t Mfs_Csio_ReadData32(volatile stc_mfsn_t* pstcCsio)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
Return Values	**Description**
`uint32_t`	The value of RDR (32bit) If `pstcCsio` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.8 Mfs_Csio_WriteData32()

Sets data to Transmit Data Register (`TDR`).
Note: This function accesses `TDR` by 32-bits. If `TDR` is accessed by less than or equal to 16bits, `Mfs_WriteData()` is used.

Prototype	
`en_result_t Mfs_Csio_WriteData32( volatile stc_mfsn_t* pstcCsio, const uint32_t u32Data)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `u32Data`	Data to set to TDR
Return Values	**Description**
`Ok`	Setting data to TDR ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.9 Mfs_Csio_SetChipSelectErrEnable()

Enables or disables chip select error.

Prototype	
en_result_t Mfs_Csio_SetChipSelectErrEnable(volatile stc_mfsn_t* pstcCsio, boolean_t bEnable)	
Parameter Name	**Description**
[in] pstcCsio	A pointer to MFS instance
[in] bEnable	TRUE: Enable chip select error FALSE: Disable chip select error
Return Values	**Description**
Ok	Enabling/disabling generation of chip select error ended with no error
ErrorInvalidParameter	pstcCsio == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.10 Mfs_Csio_SetChipSelectErrIntEnable()

Enables or disables interrupt by chip select error.

Prototype	
`en_result_t Mfs_Csio_SetChipSelectErrIntEnable( volatile stc_mfsn_t* pstcCsio, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable interrupt by chip select error `FALSE`: Disable interrupt by chip select error
Return Values	**Description**
`Ok`	Enabling/Disabling interrupt by chip select error ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.11 Mfs_Csio_GetStatus()

Gets serial status of the CSIO block.

Prototype	
`uint16_t Mfs_Csio_GetStatus(volatile stc_mfsn_t* pstcCsio, uint16_t u16StatusMask)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `u16StatsuMask`	Mask data to status
Return Values	**Description**
`uint16_t`	Masked SACSR value If `pstcCsio` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.12 Mfs_Csio_ClrChipSelectErr()

Clears chip select error status of the CSIO block.

Prototype	
`en_result_t Mfs_Csio_ClrChipSelectErr(volatile stc_mfsn_t* pstcCsio)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
Return Values	**Description**
`Ok`	Clearing chip select error ends with not error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.13 Mfs_Csio_ClrTimerIntReq()

Clears timer interrupt request (`TINT` in `SACSR`) of the CSIO block.

Prototype	
`en_result_t Mfs_Csio_ClrTimerIntReq(volatile stc_mfsn_t* pstcCsio)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
Return Values	**Description**
`Ok`	Clearing timer interrupt source ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.14 Mfs_Csio_SetSerialTimerIntEnable()

Enables or disables serial timer interrupt of the CSIO block.

Prototype	

`en_result_t Mfs_Csio_SetSerialTimerIntEnable(`
`volatile stc_mfsn_t* pstcCsio,`
`boolean_t bEnable)`	

Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable serial timer interrupt `FALSE`: Disable serial timer interrupt
Return Values	**Description**
`Ok`	Enabling/Disabling serial timer interrupt ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.15 *Mfs_Csio_SetSyncTransEnable()*

Enables or disables synchronous transfer with serial timer of the CSIO block.

Prototype	
`en_result_t Mfs_Csio_SetSyncTransEnable(volatile stc_mfsn_t* pstcCsio, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable synchronous transfer with serial timer `FALSE`: Disable synchronous transfer with serial timer
Return Values	**Description**
`Ok`	Enabling/Disabling synchronous transfer with serial timer ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.16 *Mfs_Csio_SetTimerPrescale()*

Sets serial timer prescale (`TDIV` in `SACSR`) of the CSIO block

Prototype	
`en_result_t Mfs_Csio_SetTimerPrescale(volatile stc_mfsn_t* pstcCsio, uint8_t u8Prescale)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `u8Prescale`	Prescale value
Return Values	**Description**
`Ok`	Setting serial timer prescale ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.17 Mfs_Csio_SetSerialTimerEnable()

Enables or disables serial timer of the CSIO block.

Prototype	
`en_result_t Mfs_Csio_SetSerialTimerEnable(volatile stc_mfsn_t* pstcCsio, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable serial timer `FALSE`: Disable serial timer
Return Values	**Description**
`Ok`	Enabling/Disabling serial timer ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.18 Mfs_Csio_GetSerialTimer()

Gets a value of serial timer of the CSIO block.

Prototype	
`uint16_t Mfs_Csio_GetSerialTimer(volatile stc_mfsn_t* pstcCsio)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
Return Values	**Description**
`uint16_t`	The value of STMR If `pstcCsio` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.19 Mfs_Csio_SetCmpVal4SerialTimer()

Sets a compare value of serial timer of the CSIO block.

Prototype	
`en_result_t Mfs_Csio_SetCmpVal4SerialTimer( volatile stc_mfsn_t* pstcCsio, uint16_t u16CompareValue)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `u16CompareValue`	Compare value of serial timer
Return Values	**Description**
`Ok`	Setting compare value ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.20 *Mfs_Csio_SetCsHold()*

Sets serial chip select to non-hold or hold for the CSIO block (SPI).

Prototype	
`en_result_t Mfs_Csio_SetCsHold( volatile stc_mfsn_t* pstcCsio, boolean_t bHold)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `bHold`	`FALSE`: Non-hold the active status of the serial chip select `TRUE`: Hold the active status of the serial chip select
Return Values	**Description**
`Ok`	Setting serial chip select to non-hold or hold ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.21 *Mfs_Csio_SetCsTimingPrescale()*

Sets prescale for serial chip select timing (`CDIV` in `SCSCR`) of the CSIO block (SPI).

Prototype	
`en_result_t Mfs_Csio_SetCsTimingPrescale(volatile stc_mfsn_t* pstcCsio, uint8_t u8Prescale)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `u8Prescale`	Prescale value for the serial chip select timing
Return Values	**Description**
`Ok`	Setting prescale value ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.22 *Mfs_Csio_SetCsInActiveLevel()*

Sets a signal level of the CSIO block (SPI) when serial chip select is in-active.

Prototype	
`en_result_t Mfs_Csio_SetCsInActiveLevel(volatile stc_mfsn_t* pstcCsio, boolean_t bLevel)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `bLevel`	`FALSE`: In-active level is low `TRUE`: In-active level is high
Return Values	**Description**
`Ok`	Setting in-active level ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.23 *Mfs_Csio_SetChipSelectEnable()*

Enables or disables a serial chip select of the CSIO block (SPI).

Prototype	
`en_result_t Mfs_Csio_SetChipSelectEnable(volatile stc_mfsn_t* pstcCsio, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `bEnable`	`FALSE`: Disable serial chip select `TRUE`: Enable serial chip select
Return Values	**Description**
`Ok`	Enabling/Disabling serial chip select ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.24 Mfs_Csio_SetChipSelectOutEnable()

Enables or disables a serial chip select output signal of the CSIO block (SPI).

Prototype	
`en_result_t Mfs_Csio_SetChipSelectOutEnable(volatile stc_mfsn_t* pstcCsio, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `bEnable`	`FALSE`: Disable serial chip select output `TRUE`: Enable serial chip select output
Return Values	**Description**
`Ok`	Enabling/Disabling serial chip select output ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.25 Mfs_Csio_SetCsTimingConfig()

Sets a configuration of timings for the serial chip select of the CSIO block (SPI). This function sets `SCSTR10` and `SCSTR32`.

Prototype	
`en_result_t Mfs_Csio_SetCsTimingConfig(volatile stc_mfsn_t* pstcCsio, stc_mfs_csio_cs_timing_t* pstcCsTimingCfg)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `pstcCsTimingCfg`	A pointer to a configuration of serial chip select timings
Return Values	**Description**
`Ok`	Setting a configuration of serial chip select timings ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.26 Mfs_Csio_SetTxLength()

Sets a transfer length (`TBYTE0`) of the CSIO block. This function works when the serial timer or the serial chip select is used.

Prototype	
`en_result_t Mfs_Csio_SetTxLength(volatile stc_mfsn_t* pstcCsio, uint8_t u8TxBytes)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `u8TxBytes`	Transfer length to `TBYTE0`
Return Values	**Description**
`Ok`	Setting transfer length ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.27 Mfs_Csio_SetBaudRate()

Sets a baud rate of the CSIO block.

Prototype	
`en_result_t Mfs_Csio_SetBaudRate(volatile stc_mfsn_t* pstcCsio, uint32_t u32BaudRate)`	
Parameter Name	**Description**
[in] `pstcCsio`	A pointer to the MFS instance
[in] `u32BaudRate`	Baud rate (bps)
Return Values	**Description**
`Ok`	Setting baud rate ended with no error
`ErrorInvalidParameter`	`pstcCsio == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)
`ErrorUninitialized`	Baud rate was not set properly

7.22.2.28 Mfs_I2c_Init()

Initializes the MFS block to I^2C.

Prototype	
`en_result_t Mfs_I2c_Init( volatile stc_mfsn_t* pstcI2c, const stc_mfs_i2c_config_t* pstcConfig)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS instance
[in] `pstcConfig`	A pointer to a structure of the I^2C configuration
Return Values	**Description**
`Ok`	Initialization ended with no error

`ErrorInvalidParameter`	`pstcI2c == NULL`
	`pstcConfig == NULL`
	`pstcMfsInternData == NULL` (invalid or disabled MFS unit)
	Parameter out of range

7.22.2.29 Mfs_I2c_DeInit()

Dinitializes the I^2C block.

Prototype	
`en_result_t Mfs_I2c_DeInit(volatile stc_mfsn_t* pstcI2c)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS instance
Return Values	**Description**
`Ok`	Deinitialization ended with no error
`ErrorInvalidParameter`	`pstcI2c == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.30 Mfs_I2c_SetTxIntEnable()

Enables or disables transfer interrupt of the I²C block.

Prototype	
`en_result_t Mfs_I2c_SetTxIntEnable( volatile stc_mfsn_t* pstcI2c, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable the transmission interrupt `FALSE`: Disable the transmission interrupt
Return Values	**Description**
`Ok`	Enabling/Disabling transmission interrupt ended with no error
`ErrorInvalidParameter`	`pstcI2c == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.31 Mfs_I2c_SetRxIntEnable()

Enables or disables receive interrupt of the I²C block.

Prototype	
`en_result_t Mfs_I2c_SetRxIntEnable(volatile stc_mfsn_t* pstcI2c, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable receive interrupt `FALSE`: Disable receive interrupt
Return Values	**Description**
`Ok`	Enabling/Disabling receive interrupt ended with no error
`ErrorInvalidParameter`	`pstcI2c == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.32 Mfs_I2c_SetAckEnable()

Enables or disables ACK of the I^{2}C block.

Prototype	
`en_result_t Mfs_I2c_SetAckEnable(volatile stc_mfsn_t* pstcI2c, boolean_t bEnable)`	
Parameter Name	**Description**
[in] pstcI2c	A pointer to the MFS instance
[in] bEnable	TRUE: Enable ACK FALSE: Disable ACK (NACK)
Return Values	**Description**
Ok	Enabling/Disabling ACK ended with no error
ErrorInvalidParameter	pstcI2c == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.33 Mfs_I2c_SetWaitSelect()

Sets wait selection of the I^{2}C block.

Prototype	
`en_result_t Mfs_I2c_SetWaitSelect(volatile stc_mfsn_t* pstcI2c, uint8_t u8WaitSelect)`	
Parameter Name	**Description**
[in] pstcCsio	A pointer to the MFS instance
[in] u8WaitSelect	MfsI2cWaitSelAfterAck: Waits (9bits) after acknowledge MfsI2cWaitSelDataTxRx: Waits (8bits) after data send or receive
Return Values	**Description**
Ok	Setting wait selection ended with no error
ErrorInvalidParameter	pstcI2c == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.34 Mfs_I2c_SetCondDetIntEnable()

Enables or disables condition detection interrupt of the I^2C block.

Prototype	
`en_result_t Mfs_I2c_SetCondDetIntEnable (volatile stc_mfsn_t* pstcI2c, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable condition detection interrupt `FALSE`: Disable condition detection interrupt
Return Values	**Description**
`Ok`	Enabling/Disabling condition detection interrupt ended with no error
`ErrorInvalidParameter`	`pstcI2c == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.35 Mfs_I2c_SetIntEnable()

Enables or disables interrupt of the I^2C block.

Prototype	
`en_result_t Mfs_I2c_SetIntEnable( volatile stc_mfsn_t* pstcI2c, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable interrupt `FALSE`: Disable interrupt
Return Values	**Description**
`Ok`	Enabling/Disabling interrupt ended with no error
`ErrorInvalidParameter`	`pstcI2c == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.36 Mfs_I2c_GetBusErrStatus()

Returns bus error indication (BER in IBCR) of the I^{2}C block.

Prototype	
boolean_t Mfs_I2c_GetBusErrStatus(volatile stc_mfsn_t* pstcI2c)	
Parameter Name	**Description**
[in] pstcI2c	A pointer to the MFS instance
Return Values	**Description**
boolean_t	FALSE: No error TRUE: A bus error was detected If pstcI2c is NULL or pstcMfsInternData is NULL (invalid or disabled MFS unit), this function returns FALSE.

7.22.2.37 Mfs_I2c_GetIntStatus()

Returns interrupt indication (INT in IBCR) of the I^{2}C block.

Prototype	
boolean_t Mfs_I2c_GetIntStatus(volatile stc_mfsn_t* pstcI2c)	
Parameter Name	**Description**
[in] pstcI2c	A pointer to the MFS instance
Return Values	**Description**
boolean_t	FALSE: Interrupt is not generated TRUE: Interrupt has been generated If pstcI2c is NULL or pstcMfsInternData is NULL (invalid or disabled MFS unit), this function returns FALSE.

7.22.2.38 Mfs_I2c_ClearIntStatus()

Clears interrupt flag (INT in IBCR) for the I^{2}C block.

Prototype	
en_result_t Mfs_I2c_ClearIntStatus(volatile stc_mfsn_t* pstcI2c)	
Parameter Name	**Description**
[in] pstcI2c	A pointer to the MFS instance
Return Values	**Description**
Ok	Clearing interrupt sources ended with no error
ErrorInvalidParameter	pstcI2c == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.39 Mfs_I2c_SetTransmitEmpty()

Sets transfer empty (clear TSET in SSR) of the I^2C block.

Prototype	
en_result_t Mfs_I2c_SetTransmitEmpty(volatile stc_mfsn_t* pstcI2c)	
Parameter Name	**Description**
[in] pstcI2c	A pointer to the MFS instance
Return Values	**Description**
Ok	Setting transmission empty ended with no error
ErrorInvalidParameter	pstcI2c == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.40 Mfs_I2c_SetDmaModeEnable()

Enables or disables DMA mode of the I^2C block.

Prototype	
en_result_t Mfs_I2c_SetDmaModeEnable(volatile stc_mfsn_t* pstcI2c, boolean_t bEnable)	
Parameter Name	**Description**
[in] pstcI2c	A pointer to the MFS instance
[in] bEnable	TRUE: Enable DMA mode FALSE: Disable DMA mode
Return Values	**Description**
Ok	Enabling/Disabling DMA mode ended with no error
ErrorInvalidParameter	pstcI2c == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.41 Mfs_I2c_SetNoizeFilter()

Sets noise filter value (NFCR) of the I^{2}C block.

Prototype	
en_result_t Mfs_I2c_SetNoizeFilter(volatile stc_mfsn_t* pstcI2c, uint8_t u8NzFilter)	
Parameter Name	**Description**
[in] pstcI2c	A pointer to the MFS instance
[in] u8NzFilter	Noise filter setting
Return Values	**Description**
Ok	Setting noise filter ended with no error
ErrorInvalidParameter	pstcI2c == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit) u8NzFilter is out of range

7.22.2.42 Mfs_I2c_SetSdaOutLevel()

Sets SDA output level after passing noise filter of the I^{2}C block.

Prototype	
en_result_t Mfs_I2c_SetSdaOutLevel(volatile stc_mfsn_t* pstcI2c, boolean_t bLevel)	
Parameter Name	**Description**
[in] pstcI2c	A pointer to the MFS instance
[in] bLevel	FALSE: Set “L” level after passing noise filter for SDA output TRUE: Set “H” level after passing noise filter for SDA output
Return Values	**Description**
Ok	Setting SDA output level after passing noise filter ended with no error
ErrorInvalidParameter	pstcI2c == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.43 Mfs_I2c_SetSclOutLevel()

Sets SCL output level after passing noise filter of the I2C block.

Prototype	
`en_result_t Mfs_I2c_SetSclOutLevel( volatile stc_mfsn_t* pstcI2c, boolean_t bLevel)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS instance
[in] `bLevel`	`FALSE`: Set "L" level after passing noise filter for SCL output `TRUE`: Set "H" level after passing noise filter for SCL output
Return Values	**Description**
`Ok`	Setting SCL output level after passing noise filter ended with no error
`ErrorInvalidParameter`	`pstcI2c == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.44 Mfs_I2c_SetSerlalOutEnable()

Enables or disables output signal of the I2C block.

Prototype	
`en_result_t Mfs_I2c_SetSerlalOutEnable(volatile stc_mfsn_t* pstcI2c, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable the serial output `FALSE`: Disable the serial output
Return Values	**Description**
`Ok`	Enabling/Disabling serial output ended with no error
`ErrorInvalidParameter`	`pstcI2c == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.45 Mfs_I2c_SetBusErrorControlEnable()

Enables or disables the I^2C block operation after bus error.

Prototype	
`en_result_t Mfs_I2c_SetBusErrorControlEnable( volatile stc_mfsn_t* pstcI2c, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable I^2C to continue after bus error `FALSE`: Disable I^2C to continue after bus error
Return Values	**Description**
`Ok`	Enabling/Disabling the I^2C to continue its opration after the bus error ended with no error
`ErrorInvalidParameter`	`pstcI2c == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.46 Mfs_I2c_SetI2cEnable()

Enables or disables I^2C bus and sets a slave address mask value.

Prototype	
`en_result_t Mfs_I2c_SetI2cEnable(volatile stc_mfsn_t* pstcI2c, boolean_t bEnable, uint8_t u8AddrMask)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable I^2C interface `FALSE`: Disable I^2C interface
[in] `u8AddrMask`	Slave address mask value
Return Values	**Description**
`Ok`	Enabling/Disabling the I^2C interface and seting slave address mask ended with no error
`ErrorInvalidParameter`	`pstcI2c == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.47 Mfs_I2c_SetSlvAddrEnable()

Enables or disables slave address for the I2C block.

Prototype	
`en_result_t Mfs_I2c_SetSlvAddrEnable(volatile stc_mfsn_t* pstcI2c, boolean_t bEnable, uint8_t u8SlvAdr)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable slave address `FALSE`: Disable slave address
[in] `u8SlvAdr`	Slave address
Return Values	**Description**
Ok	Enabling/Disabling slave address ended with no error
ErrorInvalidParameter	`pstcI2c == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.48 Mfs_I2c_SetBaudRate()

Sets a baud rate for the I2C block.

Prototype	
`en_result_t Mfs_I2c_SetBaudRate(volatile stc_mfsn_t* pstcI2c, uint32_t u32BaudRate)`	
Parameter Name	**Description**
[in] `pstcI2c`	A pointer to the MFS instance
[in] `u32BaudRate`	Baud rate (bps)
Return Values	**Description**
`Ok`	Setting baud rate ended with no error
`ErrorInvalidParameter`	`pstcI2c == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)
`ErrorUninitialized`	Baud rate was not set properly

7.22.2.49 Mfs_Lin_Init()

Initializes the MFS block to LIN.

Prototype	
`en_result_t Mfs_Lin_Init(volatile stc_mfsn_t* pstcLin,` `const stc_mfs_lin_config_t* pstcConfig)`	
Parameter Name	**Description**
[in] `pstcLin`	A pointer to the MFS instance
[in] `pstcConfig`	A pointer to structure of the LIN configuration
Return Values	**Description**
`Ok`	Initialization ended with no error

`ErrorInvalidParameter`	`pstcLin == NULL`
	`pstcConfig == NULL`
	`pstcMfsInternData == NULL` (invalid or disabled MFS unit)
	Parameter out of range

7.22.2.50 Mfs_Lin_DeInit()

Deinitializes the LIN block.

Prototype	
`en_result_t Mfs_Lin_DeInit(volatile stc_mfsn_t* pstcLin)`	
Parameter Name	**Description**
[in] `pstcLin`	A pointer to the MFS instance
Return Values	**Description**
`Ok`	Dinitialization ended with no error
`ErrorInvalidParameter`	`pstcLin == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.51 Mfs_Lin_SetBreak()

Sets the LIN break field (`LBR` in `SCR`).

Prototype	
`en_result_t Mfs_Lin_SetBreak(volatile stc_mfsn_t* pstcLin)`	
Parameter Name	**Description**
[in] `pstcLin`	A pointer to the MFS instance
Return Values	**Description**
`Ok`	Setting the LIN break field ended with no error
`ErrorInvalidParameter`	`pstcLin == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.52 Mfs_Lin_ClearBreakDetFlag

Clears the LIN break detection flag (LBD in SSR).

Prototype	
`en_result_t Mfs_Lin_ClearBreakDetFlag(volatile stc_mfsn_t* pstcLin)`	
Parameter Name	**Description**
[in] pstcLin	A pointer to the MFS instance
Return Values	**Description**
Ok	Setting the LIN break field ended with no error
ErrorInvalidParameter	pstcLin == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.53 Mfs_Lin_SetBreakDetIntEnable()

Enables or disables the LIN break field detection interrupt.

Prototype	
`en_result_t Mfs_Lin_SetBreakDetIntEnable(volatile stc_mfsn_t* pstcLin, boolean_t bEnable)`	
Parameter Name	**Description**
[in] pstcLin	A pointer to the MFS instance
[in] bEnable	TRUE: Enable the LIN break field detection interrupt FALSE: Disable the LIN break field detection interrupt
Return Values	**Description**
Ok	Enabling/Disabling the LIN break field detection interrupt ended with no error
ErrorInvalidParameter	pstcLin == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.54 Mfs_Lin_SetBreakConfig()

Sets the LIN break filed length and break delimiter length (`LBL` and `DEL` in ESCR).

Prototype	
`en_result_t Mfs_Lin_SetBreakConfig( volatile stc_mfsn_t* pstcLin, uint8_t u8BreakLen uint8_t u8DelimiterLen)`	
Parameter Name	**Description**
[in] `pstcLin`	A pointer to the MFS instance
[in] `u8BreakLen`	Length of the LIN break field length
[in] `u8DelimiterLen`	Length of the LIN break delimiter
Return Values	**Description**
`Ok`	Setting the LIN break field length and break delimiter length ended with no error
`ErrorInvalidParameter`	`pstcLin == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit) Parameter out of range

7.22.2.55 Mfs_Lin_SetBaudRate()

Sets a baud rate of the LIN block.

Prototype	
`en_result_t Mfs_Lin_SetBaudRate(volatile stc_mfsn_t* pstcLin, uint32_t u32BaudRate)`	
Parameter Name	**Description**
[in] `pstcLin`	A pointer to the MFS instance
[in] `u32BaudRate`	Baud rate (bps)
Return Values	**Description**
`Ok`	Setting the baud rate ended with no error
`ErrorInvalidParameter`	`pstcLin == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)
`ErrorUninitialized`	The baud rate was not set properly

7.22.2.56 Mfs_SetSerialOutputEnable()

Enables or disables serial output signal of the MFS block (only for UART/CSIO/LIN).

Prototype	
en_result_t Mfs_SetSerialOutputEnable(volatile stc_mfsn_t* pstcMfs, boolean_t bEnable)	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
[in] bEnable	TRUE: Enable serial output FALSE: Disable serial output
Return Values	**Description**
Ok	Enabling/Disabling serial output ended with no error
ErrorInvalidParameter	pstcMfs == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.57 Mfs_SetWakeUpControlEnable()

Enables or disables wake up function of the MFS block (only for UART/LIN).

Prototype	
en_result_t Mfs_SetWakeUpControlEnable(volatile stc_mfsn_t* pstcMfs, boolean_t bEnable)	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
[in] bEnable	TRUE: Enable wake up function FALSE: Disable wake up function
Return Values	**Description**
Ok	Enabling/Disabling wake up function ended with no error
ErrorInvalidParameter	pstcMfs == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.58 Mfs_SoftwareReset()

Generates software reset of the MFS block (only for UART/CSIO/LIN). This function initializes internal state of the MFS block.

Prototype	
`en_result_t Mfs_SoftwareReset(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
Return Values	**Description**
`Ok`	Generating software reset ended with no error
`ErrorInvalidParameter`	pstcMfs == NULL pstcMfsInternData == NULL `(invalid or disabled MFS unit)`

7.22.2.59 Mfs_SetRxIntEnable()

Enables or disables receive interrupt of the MFS block (only for UART/CSIO/LIN).

Prototype	
`en_result_t Mfs_SetRxIntEnable(volatile stc_mfsn_t* pstcMfs,` `boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable receive interrupt `FALSE`: Disable receive interrupt
Return Values	**Description**
`Ok`	Enabling/Disabling receive interrupt ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.60 Mfs_SetTxIntEnable()

Enables or disables send interrupt of the MFS block (only for UART/CSIO/LIN).

Prototype	
en_result_t Mfs_SetTxIntEnable(volatile stc_mfsn_t* pstcMfs, boolean_t bEnable)	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
[in] bEnable	TRUE: Enable send interrupt FALSE: Disable send interrupt
Return Values	**Description**
Ok	Enabling/Disabling send interrupt ended with no error
ErrorInvalidParameter	pstcMfs == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.61 Mfs_SetTxBusIdleIntEnable()

Enables or disables TX bus idle interrupt of the MFS block (only for UART/CSIO/LIN).

Prototype	
en_result_t Mfs_SetTxIntEnable(volatile stc_mfsn_t* pstcMfs, boolean_t bEnable)	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
[in] bEnable	TRUE: Enable TX bus idle interrupt FALSE: Disable TX bus idle interrupt
Return Values	**Description**
Ok	Enabling/Disabling TX bus idle interrupt ended with no error
ErrorInvalidParameter	pstcMfs == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.62 Mfs_SetTxFifoIntEnable()

Enables or disables TX FIFO interrupt of the MFS block (only for UART/CSIO/LIN).

Prototype	
`en_rsult_t Mfs_SetTxFifoIntEnable(volatile stc_mfsn_t* pstcMfs, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable TX FIFO interrupt `FALSE`: Disable TX FIFO interrupt
Return Values	**Description**
`Ok`	Enabling/Disabling TX FIFO interrupt ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.63 Mfs_SetRxEnable()

Enables or disables receive of the MFS block (only for UART/CSIO/LIN).

Prototype	
`en_result_t Mfs_SetRxEnable(volatile stc_mfsn_t* pstcMfs, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `bEnable`	`TRUE`: Enable receive `FALSE`: Disable receive
Return Values	**Description**
`Ok`	Enabling/Disabling receive ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.64 Mfs_SetTxEnable()

Enables or disables transfer of the MFS block.

Prototype	
`en_result_t Mfs_SetTxEnable(volatile stc_mfsn_t* pstcMfs, boolean_t bEnable)`	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
[in] bEnable	TRUE: Enable send transfer FALSE: Disable send transfer
Return Values	**Description**
Ok	Enabling/Disabling send transfer ended with no error
ErrorInvalidParameter	pstcMfs == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.65 Mfs_GetStatus()

Gets a value of Serial Status Regsiter.

Prototype	
`uint8_t Mfs_GetStatus(volatile stc_mfsn_t* pstcMfs, uint8_t u8StatusMask)`	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
[in] u8StatusMask	Mask value
Return Values	**Description**
uint8_t	Masked SSR value If pstcMfs is NULL or pstcMfsInternData is NULL (invalid or disabled MFS unit), this function returns zero.

7.22.2.66 Mfs_ErrorClear()

Clears error flags of the MFS block. This function sets REC in SSR.

Prototype	
`en_result_t Mfs_ErrorClear(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
Return Values	**Description**
Ok	Clearing error sources ended with no error
ErrorInvalidParameter	pstcMfs == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.67 *Mfs_ReadData()*

Retreive data from RX data register (RDR) of the MFS block.

Note: This function accesses to `RDR` by 16-bits size. If `RDR` is accessed by upper 16-bits (only for CSIO), `Mfs_Csio_ReadData32()` should be used instead.

Prototype	
`uint16_t Mfs_ ReadData(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
Return Values	**Description**
`uint16_t`	The value of RDR (16bit) If `pstcMfs` is NULL or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.68 *Mfs_WriteData()*

Puts data to TX data register (`TDR`) of the MFS block.

Note: This function accesses to `TDR` by 16-bits size. If `TDR` is accessed by upper 16-bits (only for CSIO), `Mfs_Csio_WriteData32()` should be used instead.

Prototype	
`en_result_t Mfs_WriteData(volatile stc_mfsn_t* pstcMfs,` `const uint16_t u16Data)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `u16Data`	Data to TDR
Return Values	**Description**
`Ok`	Putting data to TDR ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.69 Mfs_ConfigFifo()

Configures FIFO of the MFS block.

Prototype	
en_result_t Mfs_ConfigFifo (volatile stc_mfsn_t* pstcMfs, stc_mfs_fifo_config_t* pstcFifoConfig)	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
[in] pstcFifoConfig	A pointer to a structure of MFS FIFO
Return Values	**Description**
Ok	Configuring ended with no error
ErrorInvalidParameter	pstcMfs == NULL pstcFifoConfig = NULL pstcMfsInternData == NULL (invalid or disabled MFS unit) Parameter was out of range

7.22.2.70 Mfs_GetTxFifoReqStatus()

Gets TX FIFO request status (FDRQ in FCR1) of the MFS block.

Prototype	
boolean_t Mfs_ GetTxFifoReqStatus(volatile stc_mfsn_t* pstcMfs)	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
Return Values	**Description**
boolean_t	FALSE: No request was generated to TX FIFO TRUE: Request was generated to TX FIFO If pstcMfs is NULL or pstcMfsInternData is NULL (invalid or disabled MFS unit), this function returns FALSE.

7.22.2.71 Mfs_ClrTxFifoReqStatus()

Clears TX FIFO request status (FDRQ in FCR1) of the MFS block.

Prototype	
en_result_t Mfs_ClrTxFifoReqStatus(volatile stc_mfsn_t* pstcMfs)	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
Return Values	**Description**
Ok	Clearing TX FIFO request source ended with no error
ErrorInvalidParameter	pstcMfs == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.72 Mfs_ResetFifo()

Resets FIFO of the MFS block.

Prototype	
`en_result_t Mfs_ResetFifo(volatile stc_mfsn_t* pstcMfs, uint8_t u8FIfoNumber)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `u8FIfoNumber`	FIFO number `MfsFifo1`: FIFO1 `MfsFifo2`: FIFO2
Return Values	**Description**
`Ok`	Resetting FIFO ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `u8FifoNumber` is not `MfsFifo1` or `MfsFifo2`. `pstcMfsInternData` == `NULL` (invalid or disabled MFS unit)

7.22.2.73 Mfs_SetFifoEnable()

Enables of disables FIFO of the MFS block. This function sets or clears `FE1` or `FE2` in `FCR0`.

Prototype	
`en_result_t Mfs_SetFifoEnable(volatile stc_mfsn_t* pstcMfs, uint8_t u8FIfoNumber, boolean_t bEnable)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `u8FIfoNumber`	FIFO number `MfsFifo1`: FIFO1 `MfsFifo2`: FIFO2
[in] `bEnable`	`TRUE`: Enable FIFO `FALSE`: Disable FIFO
Return Values	**Description**
`Ok`	Enabling/Disabling FIFO ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `u8FifoNumber` is not `MfsFifo1` or `MfsFifo2`. `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.74 Mfs_GetFifoBytes()

Gets data count in FIFO of the MFS block.

Prototype	
`uint16_t Mfs_GetFifoBytes(volatile stc_mfsn_t* pstcMfs, uint8_t u8FIfoNumber)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `u8FIfoNumber`	FIFO number `MfsFifo1`: FIFO1 `MfsFifo2`: FIFO2
Return Values	**Description**
`uint16_t`	Valid data count in the FIFO If `pstcMfs` is `NULL`, `u8FifoNumber` is out of range or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.75 Mfs_GetBusClock()

Returns APB2 bus clock frequency of the MFS block.

Prototype	
`uint32_t Mfs_GetBusClock(void)`	
Return Values	**Description**
`uint32_t`	APB2 bus clock frequency to the MFS block

7.22.2.76 Mfs_GetReloadValue()

Returns a reload value of register `BGR` of the MFS block (only for UART/CSIO/LIN).

Prototype	
`uint32_t Mfs_GetReloadValule(uint32_t u32BaudRate)`	
Parameter Name	**Description**
[in] `u32BaudRate`	Baud rate (bps)
Return Values	**Description**
`uint32_t`	Reload value of BGR

7.22.2.77 Mfs_I2c_GetReloadValue()

Returns a reload value of register BGR of the I²C block.

Prototype	
`uint32_t Mfs_I2c_GetReloadValule(uint32_t u32BaudRate)`	
Parameter Name	**Description**
[in] u32BaudRate	Baud rate (bps)
Return Values	**Description**
uint32_t	Reload value of BGR

7.22.2.78 Mfs_SetSMR()

Sets data to Serial Mode Register (SMR).

Prototype	
`en_result_t Mfs_SetSMR(volatile stc_mfsn_t* pstcMfs, uint8_t u8SMR)`	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
[in] u8SMR	Data to SMR
Return Values	**Description**
Ok	Sutting data to SMR ended with no error
ErrorInvalidParameter	pstcMfs == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.79 Mfs_GetSMR()

Returns a value of Serial Mode Register (SMR).

Prototype	
`uint8_t Mfs_GetSmr(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
Return Values	**Description**
uint8_t	The value of SMR If pstcMfs is NULL or pstcMfsInternData is NULL (invalid or disabled MFS unit), this function returns zero.

7.22.2.80 Mfs_SetSCR()

Sets data to Serial Control Register of the MFS block (only for UART/CSIO/LIN).

Prototype	
`en_result_t Mfs_SetSCR(volatile stc_mfsn_t* pstcMfs, uint8_t u8SCR)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `u8SCR`	Data to SCR
Return Values	**Description**
`Ok`	Sutting data to SCR ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.81 Mfs_GetSCR()

Returns a value of `SCR` of the MFS block (only for UART/CSIO/LIN).

Prototype	
`uint8_t Mfs_GetScr(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
Return Values	**Description**
`uint8_t`	The value of SCR If `pstcMfs` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.82 Mfs_GetSCR()

Returns a value of `SCR` of the MFS block (only for UART/CSIO/LIN).

Prototype	
`uint8_t Mfs_GetScr(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
Return Values	**Description**
`uint8_t`	The value of SCR If `pstcMfs` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.83 Mfs_GetESCR()

Returns a value of `ESCR` of the MFS block (only for UART/CSIO/LIN).

Prototype	
`uint8_t Mfs_GetEscr(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
Return Values	**Description**
`uint8_t`	The value of ESCR If `pstcMfs` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.84 Mfs_SetBGR()

Sets data to Baud rate Generator Register.

Prototype	
`en_result_t Mfs_SetBGR(volatile stc_mfsn_t* pstcMfs,` `uint16_t u16BGR)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `u16BGR`	Data to BGR
Return Values	**Description**
`Ok`	Setting data to BGR ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.85 Mfs_GetBGR()

Returns a value of Baud rate Generator Register.

Prototype	
`uint16_t Mfs_GetBGR(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
Return Values	**Description**
`uint16_t`	The value of BGR If `pstcMfs` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.86 Mfs_SetFCR1()

Sets data to FIFO Control Register 1 (FCR1).

Prototype	
en_result_t Mfs_SetFCR1(volatile stc_mfsn_t* pstcMfs, uint8_t u8FCR1)	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
[in] u8FCR1	Data to FCR1
Return Values	**Description**
Ok	Setting data to FCR1 ended with no error
ErrorInvalidParameter	pstcMfs == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.87 Mfs_GetFCR1()

Returns a value of FIFO Control Register 1 (FCR1).

Prototype	
uint8_t Mfs_GetFCR1(volatile stc_mfsn_t* pstcMfs)	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
Return Values	**Description**
uint16_t	The value of FCR1 If pstcMfs is NULL or pstcMfsInternData is NULL (invalid or disabled MFS unit), this function returns zero.

7.22.2.88 Mfs_SetFCR0()

Sets data to FIFO Control Register 0 (FCR0).

Prototype	
en_result_t Mfs_SetFCR0(volatile stc_mfsn_t* pstcMfs, uint8_t u8FCR0)	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
[in] u8FCR0	Data to FCR0
Return Values	**Description**
Ok	Setting data to FCR0 ended with no error
ErrorInvalidParameter	pstcMfs == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.89 Mfs_GetFCR0()

Returns a value of FIFO Control Register 0 (`FCR0`).

Prototype	
`uint8_t Mfs_GetFCR0(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
Return Values	**Description**
`uint16_t`	The value of FCR0 If `pstcMfs` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.90 Mfs_SetSCSTR10()

Sets data to Serial Chip Select Timing Register 1-0 (`SCSTR10`) of the CSIO block.

Prototype	
`en_result_t Mfs_SetSCSTR10(volatile stc_mfsn_t* pstcMfs,` `uint16_t u16SCSTR10)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `u16SCSTR10`	Data to SCSTR10
Return Values	**Description**
`Ok`	Setting data to SCSTR10 ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.91 Mfs_GetSCSTR10()

Returns a value of Serial Chip Select Timing Register 1-0 (`SCSTR10`) of the CSIO block.

Prototype	
`uint16_t Mfs_GetSCSTR10(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
Return Values	**Description**
`uint16_t`	The value of SCSTR10 If `pstcMfs` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.92 Mfs_SetSCSTR32()

Sets data to Serial Chip Select Timing Register 3-2 (`SCSTR32`) of the CSIO block.

Prototype	
`en_result_t Mfs_SetSCSTR32(volatile stc_mfsn_t* pstcMfs, uint16_t u16SCSTR32)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `u16SCSTR32`	Data to SCSTR32
Return Values	**Description**
`Ok`	Setting data to SCSTR32 ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.93 Mfs_GetSCSTR32()

Returns a value of Serial Chip Select Timing Register 3-2 (`SCSTR32`) of the CSIO block.

Prototype	
`uint16_t Mfs_GetSCSTR32(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
Return Values	**Description**
`uint16_t`	Value of SCSTR32 If `pstcMfs` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.94 Mfs_SetSACSR()

Sets data to Serial Assistant Status Control Register (`SACSR`) of the CSIO block.

Prototype	
`en_result_t Mfs_SetSACSR(volatile stc_mfsn_t* pstcMfs, uint16_t u16SACSR)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `u16SACSR`	Data to SACSR
Return Values	**Description**
`Ok`	Setting data to SACSR ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.95 Mfs_GetSACSR()

Returns a value of Serial Assistant Status Control Register (SACSR) of the CSIO block.

Prototype	
`uint16_t Mfs_GetSCSTR32(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
Return Values	**Description**
uint16_t	The value of SACSR If pstcMfs is NULL or pstcMfsInternData is NULL (invalid or disabled MFS unit), this function returns zero.

7.22.2.96 Mfs_GetSTMCR()

Returns a value of Serial Timer Compare Register (STMCR) of the CSIO block.

Prototype	
`uint16_t Mfs_GetSTMCR(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
Return Values	**Description**
uint16_t	The value of STMCR If pstcMfs is NULL or pstcMfsInternData is NULL (invalid or disabled MFS unit), this function returns zero.

7.22.2.97 Mfs_SetSCSCR()

Sets data to Serial Chip Select Control Register (SCSCR) of the CSIO block.

Prototype	
`en_result_t Mfs_SetSCSCR(volatile stc_mfsn_t* pstcMfs,` `uint16_t u16SCSCR)`	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
[in] u16SCSCR	Data to SCSCR
Return Values	**Description**
Ok	Setting data to SCSCR ended with no error
ErrorInvalidParameter	pstcMfs == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.98 Mfs_GetSCSCR()

Returns a value of Serial Chip Select Control Register (SCSCR) of the CSIO block.

Prototype	
`uint16_t Mfs_GetSCSCR(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
Return Values	**Description**
uint16_t	The value of SCSCR If pstcMfs is NULL or pstcMfsInternData is NULL (invalid or disabled MFS unit), this function returns zero.

7.22.2.99 Mfs_SetTBYTE0()

Sets data to Transfer Byte 0 (TBYTE0) of the CSIO block.

Prototype	
`en_result_t Mfs_SetTBYTE0(volatile stc_mfsn_t* pstcMfs, uint8_t u8TBYTE0)`	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
[in] u8TBYTE0	Data to TBYTE0
Return Values	**Description**
Ok	Setting data to TBYTE0 ended with no error
ErrorInvalidParameter	pstcMfs == NULL pstcMfsInternData == NULL (invalid or disabled MFS unit)

7.22.2.100 Mfs_GetTBYTE0()

Returns a value of Transfer Byte 0 (TBYTE0) of the CSIO block.

Prototype	
`uint8_t Mfs_GetSCSCR(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] pstcMfs	A pointer to the MFS instance
Return Values	**Description**
uint8_t	The value of TBYTE0 If pstcMfs is NULL or pstcMfsInternData is NULL (invalid or disabled MFS unit), this function returns zero.

7.22.2.101 Mfs_SetISBA()

Sets data to 7-bit Slave Address Register (`ISBA`) of the I²C block.

Prototype	
`en_result_t Mfs_SetISBA(volatile stc_mfsn_t* pstcMfs, uint8_t u8ISBA)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `u8ISBA`	Data to ISBA
Return Values	**Description**
`Ok`	Setting data to ISBA ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.102 Mfs_GetISBA()

Returns a value of 7-bit Slave Address Register (`ISBA`) of the I²C block.

Prototype	
`uint8_t Mfs_GetISBA(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
Return Values	**Description**
`uint8_t`	The value of ISBA If `pstcMfs` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.103 Mfs_SetISMK()

Sets data to 7-bit Slave Address Mask Register (`ISMK`) of the I²C block.

Prototype	
`en_result_t Mfs_SetISMK(volatile stc_mfsn_t* pstcMfs, uint8_t u8ISMK)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `u8ISMK`	Data to ISMK
Return Values	**Description**
`Ok`	Setting data to ISMK ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.104 Mfs_GetISMK()

Returns a value 7-bit Slave Address Mask Register (`ISMK`) of the I²C block.

Prototype	
`uint8_t Mfs_GetISMK(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
Return Values	**Description**
`uint8_t`	The value of ISMK If `pstcMfs` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.105 Mfs_GetNFCR()

Returns a value of Noize Filter Control Register (`NFCR`) of the I²C block.

Prototype	
`uint8_t Mfs_GetNFCR (volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
Return Values	**Description**
`uint16_t`	The value of ISMK If `pstcMfs` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.106 Mfs_SetEIBCR()

Sets data to Extended I²C Bus Control Register (`EIBCR`) of the I²C block.

Prototype	
`en_result_t Mfs_SetEIBCR(volatile stc_mfsn_t* pstcMfs, uint8_t u8EIBCR)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `u8EIBCR`	Data to EIBCR
Return Values	**Description**
`Ok`	Setting data to EIBCR ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.107 Mfs_GetEIBCR()

Returns a value of Extended I^{2}C Bus Control Register (`EIBCR`) of the I^{2}C block.

Prototype	
`uint8_t Mfs_SetEIBCR(volatile stc_mfsn_t* pstcMfs)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
Return Values	**Description**
`uint8_t`	The value of EIBCR If `pstcMfs` is `NULL` or `pstcMfsInternData` is `NULL` (invalid or disabled MFS unit), this function returns zero.

7.22.2.108 Mfs_SetTxIntCallBack()

Sets a send callback function for the MFS block. The given function is called when a send interrupt is generated.

Prototype	
`en_result_t Mfs_SetTxIntCallBack(volatile stc_mfsn_t* pstcMfs,` `mfs_tx_cb_func_ptr_t pfnTxCbFunc)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `pfnTxCbFunc`	A pointer to a callback function
Return Values	**Description**
`Ok`	Setting the callback function ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.109 Mfs_SetRxIntCallBack

Sets a receive callback function for the MFS block. The givne function is called when a receive interrupt is generated.

Prototype	
`en_result_t Mfs_SetRxIntCallBack(volatile stc_mfsn_t* pstcMfs,` `mfs_rx_cb_func_ptr_t pfnRxCbFunc)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `pfnRxCbFunc`	A pointer to a callback function
Return Values	**Description**
`Ok`	Setting the receive callback function ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.110 Mfs_SetStsIntCallBack

Sets the status callback function for the MFS block. The given function is called when a status interrupt is generated.

Prototype	
`en_result_t Mfs_SetStsIntCallBack(volatile stc_mfsn_t* pstcMfs, mfs_sts_cb_func_ptr_t pfnStsCbFunc)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `pfnStsCbFunc`	A pointer to a callback function
Return Values	**Description**
`Ok`	Setting the status callback function ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.111 Mfs_SetUpperLayerHandle()

Set a pointer to the handle (ex. intern data) for the upper software layer of the MFS block.

Prototype	
`en_result_t Mfs_SetUpperLayerHandle(volatile stc_mfsn_t* pstcMfs, void* pvHandle)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `pvHandle`	A pointer to a handle
Return Values	**Description**
`Ok`	Setting the handle ended with no error
`ErrorInvalidParameter`	`pstcMfs == NULL` `pstcMfsInternData == NULL` (invalid or disabled MFS unit)

7.22.2.112 MACRO definition to be able to use as same as function

Some register of the MFS block is mapped to the same address shared with another register. The MFS driver library offers APIs which have register name, for the user who wants to access by register name. The macro definitions are followings.

MACRO name	Definition	Usage
`Mfs_SetIBCR`	`Mfs_SetSCR`	I^2C
`Mfs_GetIBCR`	`Mfs_GetSCR`	I^2C
`Mfs_SetIBSR`	`Mfs_SetESCR`	I^2C
`Mfs_GetIBSR`	`Mfs_GetESCR`	I^2C
`Mfs_GetSSR(pstcMfs)`	`Mfs_GetStatus(pstcMfs, 0xFF)`	UART/CSIO/LIN/I^2C
`Mfs_GetRDR`	`Mfs_ReadData`	UART/CSIO/LIN/I^2C
`Mfs_GetTDR`	`Mfs_WriteData`	UART/CSIO/LIN/I^2C
`Mfs_GetFBYTE1(pstcMfs)`	`Mfs_GetFifoBytes(pstcMfs, MfsFifo1)`	UART/CSIO/LIN/I^2C
`Mfs_GetFBYTE2(pstcMfs)`	`Mfs_GetFifoBytes(pstcMfs, MfsFifo2)`	UART/CSIO/LIN/I^2C
`Mfs_GetSTMR`	`Mfs_Csio_GetSerialTimer`	CSIO
`Mfs_SetSTMCR`	`Mfs_Csio_SetCmpVal4SerialTimer`	CSIO
`Mfs_SetNFCR`	`Mfs_I2c_SetNoizeFilter`	I^2C

7.22.2.113 Callback functions

Receive interrupt callback function

The callback function is registered by `Mfs_SetRxIntCallBack()`. The callback function is called in the receive ISR.

Prototype	
`void (*mfs_rx_cb_func_ptr_t)(volatile stc_mfsn_t* pstcMfs, void* pvHandle)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `pvHandle`	A pointer to the intern data

Transmit interrupt callback function

The callback function is registered by `Mfs_SetTxIntCallBack()`. The callback function is called in the transmit ISR.

Prototype	
`void (*mfs_tx_cb_func_ptr_t)(volatile stc_mfsn_t* pstcMfs, void* pvHandle)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `pvHandle`	A pointer to the intern data

Status interrupt callback function

The callback function is registered by `Mfs_SetStsIntCallBack()`. The callback function is called in the status ISR.

Prototype	
`void (*mfs_tx_cb_func_ptr_t)(volatile stc_mfsn_t* pstcMfs, void* pvHandle)`	
Parameter Name	**Description**
[in] `pstcMfs`	A pointer to the MFS instance
[in] `pvHandle`	A pointer to intern data

7.22.3 Example Code

The example software is in *\example\mfs\low_level\<module block>*.

Folder	Summary
\example\mfs\low_level\CSIO	CSIO samples folder
`mfs_csio_normal_master_unuse_int`	CSIO normal master mode without interrupt
`mfs_csio_normal_master_use_int`	CSIO normal master mode with interrupt
`mfs_csio_normal_master_unuse_int_with_tmr`	CSIO normal master mode without interrupt and with serial timer
`mfs_csio_normal_master_use_int_with_tmr`	CSIO normal master mode with interrupt and serial timer
`mfs_csio_normal_slave_use_int`	CSIO normal slave mode with interrupt
`mfs_csio_spi_master_unuse_int`	CSIO SPI master mode without interrupt and with chip select (CS) control by GPIO
`mfs_csio_spi_master_use_int`	CSIO SPI master mode with interrupt and CS control by GPIO
`mfs_csio_spi_master_unuse_int_with_cs`	CSIO SPI master mode without interrupt and with CS control by peripheral function
`mfs_csio_spi_master_use_int_with_cs`	CSIO SPI master mode with interrupt and CS control by peripheral function
`mfs_csio_spi_slave_use_int`	CSIO SPI slave mode with interrupt and without CS control
`mfs_csio_spi_slave_use_int_with_cs`	CSIO SPI slave mode with interrupt and CS control by peripheral function
`\example\mfs\low_level\I2C\`	I²C samples folder
`i2c_master_blocking`	I²C master mode by blocking process
`i2c_master_non_blocking`	I²C master mode by non-blocking process
`i2c_slave_blocking`	I²C slave mode by blocking process
`i2c_slave_non_blocking`	I²C slave mode by non-blocking process
`\example\mfs\low_level\LIN\`	LIN sample folder
`mfs_lin`	LIN master and slave mode
`\example\mfs\low_level\UART\`	UART samples folder
`mfs_uart_unuse_int`	UART without interrupt
`mfs_uart_use_int`	UART with interrupt

These are the same action as samples for MFS_HL.

7.22.3.1 CSIO

CSIO normal master mode without using interrupt

This example software excerpt shows an usage of the CSIO driver library for a normal master without using interrupt.

```
#include "mfs/mfs.h"

...
static const stc_mfs_csio_config_t stcMfsCsioCfg = {
2000000,            // Baud rate
MfsCsioMaster,      // Master mode
MfsCsioActNormalMode, // CSIO normal mode
MfsSyncWaitZero,    // Non wait time insersion
MfsEightBits,       // 8 data bits
FALSE,              // LSB first
TRUE                // SCK Mark level High
};
...
```

```
static en_result_t SampleMfsCsioReadWrite(uint8_t*    pu8TxBuf,
                                          uint8_t*    pu8RxBuf,
                                          uint16_t    u16TransferSize
                                         )
{
  uint16_t        u16DataToSendCounter;
  uint16_t        u16DataReceivedCounter;
  uint16_t        u16Data;
  volatile uint8_t u8Reg;

  // Check for valid pointers
  if (((NULL == pu8TxBuf) && (NULL == pu8RxBuf))
  // Check for 0 < transmission length
  ||  (0 == u16TransferSize)
    )
  {
      return (ErrorInvalidParameter);
  }

  u16DataToSendCounter   = 0;
  u16DataReceivedCounter = 0;

  while (u16DataReceivedCounter != u16TransferSi
  {
    if (u16DataToSendCounter < u16TransferSize
    {
      do
      {
        // If Transmit Data Register (TDR)
        u8Reg = Mfs_GetStatus(&MFS6, MFS_CSIO_SSR_TDRE);
      // Wait for TDR empty
      } while (0 == u8Reg);
      // If pu8TxData is NULL, dummy data is sent.
      u16Data = 0;
      if (NULL != pu8TxBuf)
      {
        u16Data = (uint16_t)pu8TxBuf[u16DataToSendCounter];
      }
      // Write the data to Transmit Data Register
      Mfs_WriteData(&MFS6, u16Data);
      u16DataToSendCounter++;
    }
    ...
```

```
    ...
    // To avoid inter-byte spikes on SOT, send out 2nd byte immediately,
    //   if TDR is empty!
    if ((1 == u16DataToSendCounter) && (1 < u16TransferSize))
    {
      do
      {
        // If Transmit Data Register (TDR) is empty
        u8Reg = Mfs_GetStatus(&MFS6, MFS_CSIO_SSR_TDRE);
      // Wait for TDR empty
      } while (0 == u8Reg);
      // If pu8TxData is NULL, dummy data is sent.
      u16Data = 0;
      if (NULL != pu8TxBuf)
      {
        u16Data = (uint16_t)pu8TxBuf[u16DataToSendCounter];
      }
      // Write the data to Transmit Data Register
      Mfs_WriteData(&MFS6, u16Data);
      u16DataToSendCounter++;
    }
    do
    {
      // Check reception
      u8Reg = Mfs_GetStatus(&MFS6, 0xFF);
    // wait for reception finsihed
    } while (0 == (u8Reg & MFS_CSIO_SSR_RDRF));
    // Check error (OVR)
    if (0u != (u8Reg & MFS_CSIO_SSR_ERR))
    {
      // Clear possible reception error
      Mfs_ErrorClear(&MFS6);
    }
    // If pu8RxData is NULL, dummy data is received.
    u16Data = Mfs_ReadData(&MFS6);
    if (NULL != pu8RxBuf)
    {
      pu8RxBuf[u16DataReceivedCounter] = (uint8_t)u16Data;
    }
    u16DataReceivedCounter++;
  }

  return (Ok);
}
```

```
function
{
  uint8_t          u8RxData;
  uint8_t          u8TxData;
  ...

  // Set CSIO Ch6_0 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR5 = FM4_GPIO->PFR5 | 0x00E0;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x00150000;

  // At first un-initialize CSIO
  (void)Mfs_Csio_DeInit(&MFS6);

  // Initialize MFS as CSIO
  if (Ok != Mfs_Csio_Init(&MFS6, (stc_mfs_csio_config_t
*)&stcMfsCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // Enable serial clock output
  Mfs_Csio_SetSckOutEnable(&MFS6, TRUE);
  // Clear possible reception errors
  Mfs_ErrorClear(&MFS6);
  // Enable TX function
  Mfs_SetTxEnable(&MFS6, TRUE);
  // Enable RX function
  Mfs_SetRxEnable(&MFS6, TRUE);

  // some code here ...

  while(1)
  {
    if (Ok == SampleMfsCsioReadWrite(&u8TxData, &u8RxData, 1))
    {
      // some code here ...
    }
  }
}
```

CSIO normal master mode with using interrupt

This example software excerpt shows an usage of the CSIO driver library for a normal master with using interrupt.

```
#include "mfs/mfs.h"

...
Configuration Structure of CSIO is same as here
...

static void SampleMfsTxIrqHandler(volatile stc_mfsn_t*pstcMfs,
                                  void*                  pvHandle
                                  )
{
  // Put data from Buffer into Transmit Data Register
  Mfs_WriteData(pstcMfs, (uint16_t)pu8CsioTxBuf[u16TxBufOutIndex]);
  // Update tail
  u16TxBufOutIndex++;
  u16TxBufFillCount--;

  // If no more bytes to sent ...
  if (0 == u16TxBufFillCount)
  {
    // Disable transmission interrupt
    Mfs_SetTxIntEnable(pstcMfs, FALSE);
  }
}
```

```
static void SampleMfsRxIrqHandler(volatile stc_mfsn_t*pstcMfs,
                                  void*                 pvHandle
                                 )
{
  uint16_t        u16Data;
  volatile uint8_tu8Ssr;

  // Check Overrun error
  u8Ssr = Mfs_GetStatus(pstcMfs, MFS_CSIO_SSR_ERR);
  if (0 != u8Ssr)
  {
    // Clear possible reception errors
    Mfs_ErrorClear(pstcMfs);
  }

  // Read data from Read Data Register
  u16Data = Mfs_ReadData(pstcMfs);

  // If there is empty space in RX buffer
  if (u16RxBufFillCount < SAMPLE_CSIO_RX_BUFFSIZE)
  {
    // Store read data to RX buffer
    au8CsioRxBuf[u16RxBufInIndex] = (uint8_t)u16Data;
    // Increment in index
    u16RxBufInIndex++;
    if (SAMPLE_CSIO_RX_BUFFSIZE <= u16RxBufInIndex)
    {
      u16RxBufInIndex = 0;
    }
    // Count bytes in RX buffer
    u16RxBufFillCount++;
    u16RxBufFillCnt = u16RxBufFillCount;
  }
}
```

```
static en_result_t SampleMfsCsioRead(uint8_t*pu8RxBuf,
                                   uint16_t*pu16ReadCnt
                                  )
{
  uint16_t u16Idx;
  uint16_t u16Length;
  uint16_t u16BytesToReadLeft;

  // Check for valid pointers
  if ((NULL == pu8RxBuf) || (NULL == pu16ReadCnt)) {
      return (ErrorInvalidParameter);
  }

  // Save Read Count for later use
  u16BytesToReadLeft = *pu16ReadCnt;
  *pu16ReadCnt = 0;    // Preset to default

  u16Idx = 0;

  // Read all available bytes from ring buffer, blocking.
  while (0 < u16BytesToReadLeft) {
    if (0 == u16RxBufFillCount) {
      return (Ok);
    }
    // Disable reception interrupt
    Mfs_SetRxIntEnable(&MFS6, FALSE);

    // Copy data to destinaton buffer and save no. of bytes been read
    // get number of bytes to read
    u16Length = MIN(u16RxBufFillCount, u16BytesToReadLeft);

    // if there are any bytes left to read
    if (0 != u16Length) {
      // read bytes out of RX buffer
      for (u16Idx = *pu16ReadCnt; u16Idx < (u16Length + *pu16ReadCnt);
u16Idx++) {
        pu8RxBuf[u16Idx] = au8CsioRxBuf[u16RxBufOutIndex];
        // Update out index
        u16RxBufOutIndex++;
        if (SAMPLE_CSIO_RX_BUFFSIZE <= u16RxBufOutIndex) {
          u16RxBufOutIndex = 0;
        }
      }
      u16RxBufFillCount -= u16Length; // Update fill counter
    }

    *pu16ReadCnt += u16Length;           // Provide no. of read to the caller
    u16BytesToReadLeft -= u16Length; // Some data processed

    // Enable reception interrupt
    Mfs_SetRxIntEnable(&MFS6, TRUE);
  }

  return (Ok);
}
```

```
static en_result_t SampleMfsCsioWrite(uint8_t*
         pu8TxBuf,
                                    uint16_t  u16WriteCnt
                                   )
{
  // Check for valid pointer
  if (NULL == pu8TxBuf)
  {
    return (ErrorInvalidParameter);
  }

  // Check if nothing to do
  if (0 == u16WriteCnt)
  {
    return (Ok);
  }

  // Disable transmit interrupt
  Mfs_SetTxIntEnable(&MFS6, FALSE);
  // Set tx data area
  pu8CsioTxBuf = pu8TxBuf;
  // Set tx data fill count (to transmit by interrupt)
  u16TxBufFillCount = u16WriteCnt;
  // Initialize output index
  u16TxBufOutIndex = 0;
  // Enable transmit interrupt
  Mfs_SetTxIntEnable(&MFS6, TRUE);
  // Wait until all data has been tranferred to the MFS
  // This is fully INT driven
  while (0 != u16TxBufFillCount);

  return (Ok);
}
```

```
static en_result_t SampleMfsCsioReadWrite(uint8_t*    pu8TxBuf,
                                          uint16_t     u16WriteCnt,
                                          uint8_t*     pu8RxBuf,
                                          uint16_t*    pu16ReadCnt
                                         )
{
  uint8_t          au8CsioRxDummyBuf[SAMPLE_CSIO_RX_BUFFSIZE];
  en_result_t      enResult;
  uint16_t         u16ReadCnt;

  // If Rx buffer specified NULL ...
  if (NULL == pu8RxBuf)
  {
    // Use internal buffer for dummy reading
    pu8RxBuf = au8CsioRxDummyBuf;
  }
  // Write transmit data (blocking)
  enResult = SampleMfsCsioWrite(pu8TxBuf, u16WriteCnt);
  if ((Ok == enResult) && (0 != u16WriteCnt))
  {
    // Wait to receive transmitted bytes.
    while (u16WriteCnt > u16RxBufFillCnt)
    {
      continue;
    }
    u16ReadCnt = u16RxBufFillCnt;
    // Read received data
    enResult = SampleMfsCsioRead(pu8RxBuf, &u16ReadCnt)
    if (Ok == enResult)
    {
      if (NULL != pu16ReadCnt)
      {
        // Set the received counts
        *pu16ReadCnt = u16ReadCnt;
      }
      // Update fill count of received buffer
      __disable_irq();
      u16RxBufFillCnt -= u16ReadCnt;
      __enable_irq();
    }
  }

  return (enResult);
}
```

```
function
{
  uint8_t au8RxData[128];
  uint16_tu16ReadCnt;
  uint8_t u8TxData;
  ...

  // Set CSIO Ch6_0 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR5 = FM4_GPIO->PFR5 | 0x00E0;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x00150000;

  // At first un-initialize CSIO
  (void)Mfs_Csio_DeInit(&MFS6);
  // Initialize MFS as CSIO
  if (Ok != Mfs_Csio_Init(&MFS6, (stc_mfs_csio_config_t *)&stcMfsCsioCfg))
  {
    // some code here ...
    while(1);
  }
  // Register interrupt handler and internal handle
  Mfs_SetTxIntCallBack(&MFS6, SampleMfsTxIrqHandler);
  Mfs_SetRxIntCallBack(&MFS6, SampleMfsRxIrqHandler);

  // Initialize variables for RX
  // (Variables for TX is initialized in SampleMfsCsioWrite())
  u16RxBufInIndex = 0;
  u16RxBufOutIndex = 0;
  u16RxBufFillCount = 0;
  u16RxBufFillCnt = 0;

  // Enable serial clock output
  Mfs_Csio_SetSckOutEnable(&MFS6, TRUE);
  // Clear possible reception errors
  Mfs_ErrorClear(&MFS6);
  // Enable TX function
  Mfs_SetTxEnable(&MFS6, TRUE);
  // Enable RX function
  Mfs_SetRxEnable(&MFS6, TRUE);
  // Enable RX interrupt
  Mfs_SetRxIntEnable(&MFS6, TRUE);
  // Init transmission interrupt
  Mfs_InitTxIrq(&MFS6);
  // Init reception interrupt
  Mfs_InitRxIrq(&MFS6);

  // some code here ...

  while(1)
  {
    if (Ok == SampleMfsCsioReadWrite(&u8TxData, 1, au8RxData, &u16ReadCnt))
    {
      // some code here ...
    }
  }
}
```

CSIO normal master mode without using interrupt and with using serial timer

This example software excerpt shows an usage of the CSIO driver library for a normal master without using interrupt and using serial timer.

```
#include "mfs/mfs.h"

...
Configuration Structure of CSIO is same as here
...

static en_result_t SampleMfsCsioReadWrite(uint8_t*          pu8TxBuf,
                                          uint8_t*  pu8RxBuf,
                                          uint16_t  u16TransferSize)
{
  uint16_tu16DataToSendCounter;
  uint16_tu16DataReceivedCounter;
  uint16_tu16Data;
  volatile uint8_t  u8Reg;

  // Check for valid pointers and check for 0 < transmission length
  if (((NULL == pu8TxBuf) && (NULL == pu8RxBuf)) || (0 == u16TransferSize))
  {
      return (ErrorInvalidParameter);
  }
  // Enable serial timer
  Mfs_SetTxEnable(&MFS6, FALSE);
  Mfs_Csio_SetSerialTimerEnable(&MFS6, TRUE);
  Mfs_SetTxEnable(&MFS6, TRUE);

  u16DataToSendCounter   = 0;
  u16DataReceivedCounter = 0;

  while (u16DataReceivedCounter != u16TransferSize)
  {
    if (u16DataToSendCounter < u16TransferSize)
    {
      do
      {
        // If Transmit Data Register (TDR) is empty
        u8Reg = Mfs_GetStatus(&MFS6, MFS_UART_SSR_TDRE);
      // Wait for TDR empty
      } while (0 == u8Reg);
      // If pu8TxData is NULL, dummy data is sent.
      u16Data = 0;
      if (NULL != pu8TxBuf)
      {
        u16Data = (uint16_t)pu8TxBuf[u16DataToSendCounter];
      }
      // Write the data to Transmit Data Register
      Mfs_WriteData(&MFS6, u16Data);
      u16DataToSendCounter++;
    }
    ...
```

```
    ...
    // To avoid inter-byte spikes on SOT, send out 2nd byte
immediately,
    //   if TDR is empty!
    if ((1 == u16DataToSendCounter) && (1 < u16TransferSize)) {
      do {
        // If Transmit Data Register (TDR) is empty
        u8Reg = Mfs_GetStatus(&MFS6, MFS_UART_SSR_TDRE);
      // Wait for TDR empty
      } while (0 == u8Reg);
      // If pu8TxData is NULL, dummy data is sent.
      u16Data = 0;
      if (NULL != pu8TxBuf) {
        u16Data = (uint16_t)pu8TxBuf[u16DataToSendCounter];
      }
      // Write the data to Transmit Data Register
      Mfs_WriteData(&MFS6, u16Data);
      u16DataToSendCounter++;
    }
    do {
      // Check reception
      u8Reg = Mfs_GetStatus(&MFS6,
                            (MFS_CSIO_SSR_ERR | MFS_CSIO_SSR_RDRF));
    // wait for reception finsihed
    } while (0 == u8Reg);
    if (0 != (u8Reg & MFS_CSIO_SSR_ERR)) {
      // Clear possible reception errors
      Mfs_ErrorClear(&MFS6);
    }
    if (0 != (u8Reg & MFS_CSIO_SSR_RDRF)) {
     // If Received Data Register (RDR) is full
      u16Data = Mfs_ReadData(&MFS6);
      if (NULL != pu8RxBuf) {
        pu8RxBuf[u16DataReceivedCounter] = u16Data;
      }
      u16DataReceivedCounter++;
    }
  }

  do {
    // Check the TX Bus status
    u8Reg = Mfs_GetStatus(&MFS6, MFS_CSIO_SSR_TBI);
  // Wait for TX bus idle
  } while (0 == u8Reg);

  // Disable serial timer
  Mfs_SetTxEnable(&MFS6, FALSE);
  Mfs_Csio_SetSerialTimerEnable(&MFS6, FALSE);
  Mfs_SetTxEnable(&MFS6, TRUE);

  return (Ok);
}
```

```
function
{
  uint8_t         au8RxBuf[32];
  uint8_t         au8TxData[4] = {0x00, 0x01, 0x02, 0x03};
  volatile uint32_t        u32Cnt;

  // Set CSIO Ch6_0 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR5 = FM4_GPIO->PFR5 | 0x00E0;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x00150000;

  // At first un-initialize CSIO
  (void)Mfs_Csio_DeInit(&MFS6);

  // Initialize MFS as CSIO
  if (Ok != Mfs_Csio_Init(&MFS6, (stc_mfs_csio_config_t
*)&stcMfsCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // Set serial timer prescale
  Mfs_Csio_SetTimerPrescale(&MFS6, MFS_SACSR_TDIV_256);
  // Enable synchronous transfer
  Mfs_Csio_SetSyncTransEnable(&MFS6, TRUE);
  // Set serial timer
  Mfs_Csio_SetCmpVal4SerialTimer(&MFS6, 62500);
  // Set transfer length
  Mfs_Csio_SetTxLength(&MFS6, 2);
  // Enable serial clock output
  Mfs_Csio_SetSckOutEnable(&MFS6, TRUE);
  // Clear possible reception errors
  Mfs_ErrorClear(&MFS6);
  // Enable TX function
  Mfs_SetTxEnable(&MFS6, TRUE);
  // Enable RX function
  Mfs_SetRxEnable(&MFS6, TRUE);

  // some code here ...

  while(1)
  {
    if (Ok == SampleMfsCsioReadWrite(au8TxData, au8RxBuf, 4))
    {
      // some code here ...
    }
  }
}
```

CSIO normal master mode with using both interrupt and serial timer

This example software excerpt shows an usage of the CSIO driver library for a normal master with using both interrupt and serial timer.

```
#include "mfs/mfs.h"

...
Configuration Structure of CSIO is same here
...
SampleMfsTxIrqHandler(),SampleMfsRxIrqHandler() and SampleMfsCsioRead()
are same as here.
```

```
static en_result_t SampleMfsCsioWrite(uint8_t*pu8TxBuf,
                                       uint16_t  u16WriteCnt
                                      )
{
  // Check for valid pointer
  if (NULL == pu8TxBuf)
  {
    return (ErrorInvalidParameter);
  }

  // Check if nothing to do
  if (0 == u16WriteCnt)
  {
    return (Ok);
  }

  // Enable serial timer
  Mfs_SetTxEnable(&MFS6, FALSE);
  Mfs_Csio_SetSerialTimerEnable(&MFS6, TRUE);
  Mfs_SetTxEnable(&MFS6, TRUE);

  // Disable transmit interrupt
  Mfs_SetTxIntEnable(&MFS6, FALSE);
  // Set tx data area
  pu8CsioTxBuf = pu8TxBuf;
  // Set tx data fill count (to transmit by interrupt)
  u16TxBufFillCount = u16WriteCnt;
  // Initialize output index
  u16TxBufOutIndex = 0;
  // Enable transmit interrupt
  Mfs_SetTxIntEnable(&MFS6, TRUE);
  // Wait until all data has been tranferred to the MFS
  // This is fully INT driven
  while (0 != u16TxBufFillCount);

  do
  {
    // Check the TX Bus status
    u8Reg = Mfs_GetStatus(&MFS6, MFS_CSIO_SSR_TBI);
  // Wait for TX bus idle
  } while (0 == u8Reg);

  // Disable serial timer
  Mfs_SetTxEnable(&MFS6, FALSE);
  Mfs_Csio_SetSerialTimerEnable(&MFS6, FALSE);
  Mfs_SetTxEnable(&MFS6, TRUE);

  return (Ok);
}

SampleMfsCsioReadWrite() are same as here.
```

```
function
{
  ...
  uint8_t          au8TxData[4] = {0x00, 0x01, 0x02, 0x03};
  ...

  // Set CSIO Ch6_0 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR5 = FM4_GPIO->PFR5 | 0x00E0;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x00150000;

  // At first un-initialize CSIO
  (void)Mfs_Csio_DeInit(&MFS6);
  // Initialize MFS as CSIO
  if (Ok != Mfs_Csio_Init(&MFS6, (stc_mfs_csio_config_t
*)&stcMfsCsioCfg))
  {
    // some code here ...
    while(1);
  }
  // Register interrupt handler and internal handle
  Mfs_SetTxIntCallBack(&MFS6, SampleMfsTxIrqHandler);
  Mfs_SetRxIntCallBack(&MFS6, SampleMfsRxIrqHandler);

  // Initialize variables for RX
  // (Variables for TX is initialized in SampleMfsCsioWrite())
  u16RxBufInIndex = 0;
  u16RxBufOutIndex = 0;
  u16RxBufFillCount = 0;
  u16RxBufFillCnt = 0;

  // Set serial timer prescale
  Mfs_Csio_SetTimerPrescale(&MFS6, MFS_SACSR_TDIV_256);
  // Enable synchronous transfer
  Mfs_Csio_SetSyncTransEnable(&MFS6, TRUE);
  // Set serial timer
  Mfs_Csio_SetCmpVal4SerialTimer(&MFS6, 62500);
  // Set transfer length
  Mfs_Csio_SetTxLength(&MFS6, 2);
  // Enable serial clock output
  Mfs_Csio_SetSckOutEnable(&MFS6, TRUE);
  // Clear possible reception errors
  Mfs_ErrorClear(&MFS6);
  // Enable TX function
  Mfs_SetTxEnable(&MFS6, TRUE);
  // Enable RX function
  Mfs_SetRxEnable(&MFS6, TRUE);
  // Enable RX interrupt
  Mfs_SetRxIntEnable(&MFS6, TRUE);
  // Init transmission interrupt
  Mfs_InitTxIrq(&MFS6);
  // Init reception interrupt
  Mfs_InitRxIrq(&MFS6);
 ...
```

```
  ...
  // some code here ...

  while(1)
  {
    if (Ok == SampleMfsCsioReadWrite(au8TxData, 4, au8RxBuf, 
&u16ReadCnt))
    {
      // some code here ...
    }
  }
}
```

CSIO normal slave mode with using interrupt

This example software excerpt shows an usage of the CSIO driver library for a normal slave with using interrupt.

```
#include "mfs/mfs.h"

...
static const stc_mfs_csio_config_t stcMfsCsioCfg = {
  2000000,                    // Baud rate (Not effective in slave mode)
  MfsCsioSlave,               // Slave mode
  MfsCsioActNormalMode,       // CSIO normal mode
  MfsSyncWaitZero,            // Non wait time insersion(Not effective in slave
                                 mode)
  MfsEightBits,               // 8 data bits
  FALSE,                      // LSB first
  TRUE                        // SCK Mark level High
};

...
static void SampleMfsTxIrqHandler(volatile stc_mfsn_t*pstcMfs,
                               void*              pvHandle
                               )
{
  // Put data from Buffer into Transmit Data Register
  Mfs_WriteData(pstcMfs, (uint16_t)au8CsioTxBuf[u16TxBufOutIndex]);
  // Update tail
  u16TxBufOutIndex++;
  if (SAMPLE_CSIO_TX_BUFFSIZE <= u16TxBufOutIndex)
  {
      u16TxBufOutIndex = 0;
  }
  u16TxBufFillCount--;

  // If no more bytes to sent ...
  if (0 == u16TxBufFillCount)
  {
    // Disable transmission interrupt
    Mfs_SetTxIntEnable(pstcMfs, FALSE);
  }
}

SampleMfsRxIrqHandler() and SampleMfsCsioRead() are same as here.
```

```
static en_result_t SampleMfsCsioWrite(uint8_t*pu8TxBuf,
                                    uint16_t  u16WriteCnt
                                   )
{
  uint16_t        u16Idx;
  uint16_t        u16DataSent;
  volatile uint8_t u8Reg;

  // Check for valid pointer
  if (NULL == pu8TxBuf)
  {
    return (ErrorInvalidParameter);
  }

  // Check if nothing to do
  if (0 == u16WriteCnt)
  {
    return (Ok);
  }

  // Check if ring buffer can take all bytes
  if (u16WriteCnt > (SAMPLE_CSIO_TX_BUFFSIZE - u16TxBufFillCount))
  {
    // not enough space left if non-blocking mode is requested
    return (ErrorBufferFull);
  }
  ...
```

```
  ...
  // Loop until all data has been sent (blocking only)
  // It is guaranteed here that the provided data will fit into the tx
buffer
  while (0 < u16WriteCnt)
  {
    // Check for transmission ongoing
    u8Reg = Mfs_GetStatus(&MFS6, MFS_CSIO_SSR_TDRE);
    // In case, a transmission is already pending
    if (0 != u8Reg)
    {
      // Disable transmit interrupt
      Mfs_SetTxIntEnable(&MFS6, FALSE);
    }

    // Copy data to provided destinaton buffer and save bytes been read
    // determine free size in TX buffer
    u16DataSent = MIN((SAMPLE_CSIO_TX_BUFFSIZE - u16TxBufFillCount),
                      u16WriteCnt);

    // store bytes in TX buffer
    for (u16Idx = 0; u16Idx < u16DataSent; u16Idx++)
    {
      au8CsioTxBuf[u16TxBufInIndex] = *pu8TxBuf;
      pu8TxBuf++;

      // Update in index
      u16TxBufInIndex++;
      if (SAMPLE_CSIO_TX_BUFFSIZE <= u16TxBufInIndex)
      {
        u16TxBufInIndex = 0;
      }
    }
    u16TxBufFillCount += u16DataSent;
    u16WriteCnt      -= u16DataSent;

    // Enable transmit interrupt
    Mfs_SetTxIntEnable(&MFS6, TRUE);
  }

  return (Ok);
}

SampleMfsCsioReadWrite() is same as here.
```

```
function
{
  uint8_t           au8RxData[128];
  uint16_tu16ReadCnt;
  uint8_t           u8TxData;

  // Set CSIO Ch6_0 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR5 = FM4_GPIO->PFR5 | 0x00E0;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x00150000;

  // At first un-initialize CSIO
  (void)Mfs_Csio_DeInit(&MFS6);
  // Initialize MFS as CSIO
  if (Ok != Mfs_Csio_Init(&MFS6, (stc_mfs_csio_config_t *)&stcMfsCsioCfg))
  {
    // some code here ...
    while(1);
  }
  // Register interrupt handler and internal handle
  Mfs_SetTxIntCallBack(&MFS6, SampleMfsTxIrqHandler);
  Mfs_SetRxIntCallBack(&MFS6, SampleMfsRxIrqHandler);

  // Initialize variables for RX
  // (Variables for TX is initialized in SampleMfsCsioWrite())
  u16TxBufInIndex = 0;
  u16TxBufOutIndex = 0;
  u16TxBufFillCount = 0;
  u16RxBufInIndex = 0;
  u16RxBufOutIndex = 0;
  u16RxBufFillCount = 0;
  u16RxBufFillCnt = 0;

  // Clear possible reception errors
  Mfs_ErrorClear(&MFS6);
  // Enable TX function
  Mfs_SetTxEnable(&MFS6, TRUE);
  // Enable RX function
  Mfs_SetRxEnable(&MFS6, TRUE);
  // Enable RX interrupt
  Mfs_SetRxIntEnable(&MFS6, TRUE);
  // Init transmission interrupt
  Mfs_InitTxIrq(&MFS6);
  // Init reception interrupt
  Mfs_InitRxIrq(&MFS6);

  // some code here ...

  while(1)
  {
    if (Ok == SampleMfsCsioReadWrite(&u8TxData, 1, au8RxData, &u16ReadCnt))
    {
      // some code here ...
    }
  }
}
```

CSIO SPI master mode without using interrupt and with using CS control by GPIO

This example software excerpt shows an usage of the CSIO SPI driver library for a master mode without using interrupt and with using CS control by GPIO.

```
#include "mfs/mfs.h"

...
static const stc_mfs_csio_config_t stcMfsCsioCfg = {
  2000000,                    // Baud rate
  MfsCsioMaster,              // Master mode
  MfsCsioActSpiMode,          // CSIO SPI mode
  MfsSyncWaitZero,            // Non wait time insersion
  MfsEightBits,               // 8 data bits
  FALSE,                      // LSB first
  TRUE                        // SCK Mark level High
};

...

static void SampleMfsSpiWait(volatile uint32_t u32Wait)
{
  // Wait specified count
  while (0 != (u32Wait--));
}

static void SampleMfsSpiEnableCs(void)
{
  // Enable CS
  FM4_GPIO->PDOR0_f.P0E = TRUE;
  // Insert wait
  SampleMfsSpiWait(40);
}

static void SampleMfsSpiDisableCs(void)
{
  // Insert wait
  SampleMfsSpiWait(40);
  // Disable CS
  FM4_GPIO->PDOR0_f.P0E = FALSE;
}

SampleMfsCsioReadWrite()) is same as here.
```

```
function
{
  en_result_t      enResult;
  uint8_t          u8RxData;
  uint8_t          u8TxData;
  ...

  // Set CSIO Ch6_1 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR0 = FM4_GPIO->PFR0 | 0x3800;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x002A0000;

  // At first un-initialize CSIO
  (void)Mfs_Csio_DeInit(&MFS6);

  // Initialize MFS as CSIO
  if (Ok != Mfs_Csio_Init(&MFS6, (stc_mfs_csio_config_t
*)&stcMfsCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // Chip select port is output
  FM4_GPIO->DDR0_f.P0E  = TRUE;
  FM4_GPIO->PDOR0_f.P0E = FALSE;

  // Enable serial clock output
  Mfs_Csio_SetSckOutEnable(&MFS6, TRUE);
  // Clear possible reception errors
  Mfs_ErrorClear(&MFS6);
  // Enable TX function
  Mfs_SetTxEnable(&MFS6, TRUE);
  // Enable RX function
  Mfs_SetRxEnable(&MFS6, TRUE);

  // some code here ...

  while(1)
  {
    // Enable CS
    SampleMfsSpiEnableCs();
    // Transmit and receive
    enResult = SampleMfsCsioReadWrite(&u8TxData, &u8RxData, 1);
    // Disable CS
    SampleMfsSpiDisableCs();
    if (Ok ==enResult)
    {
      // some code here ...
    }
  }
}
```

CSIO SPI master mode with using both interrupt and CS control by GPIO

This example software excerpt shows an usage of the CSIO SPI for a master with using both interrupt and CS control by GPIO.

```
#include "mfs/mfs.h"

...
Configuration Structure of CSIO is same as here
...
SampleMfsSpiWait(),SampleMfsSpiEnableCs() and SampleMfsSpDisableCs()
are same as here.

SampleMfsTxIrqHandler() and SampleMfsRxIrqHandler() are same as here.

static en_result_t SampleMfsSpiRead(uint8_t* pu8RxBuf,
                                    uint16_t*  pu16ReadCnt
                                   )
{
  Source code is same as SampleMfsCsioRead()here.
}

static en_result_t SampleMfsSpiWrite(uint8_t*         pu8TxBuf,
                                    uint16_t   u16WriteCnt
                                   )
{
  Source code is same as SampleMfsCsioWrite()here.
}
```

```
static en_result_t SampleMfsSpiReadWrite(uint8_t*     pu8TxBuf,
                                         uint16_t      u16WriteCnt,
                                         uint8_t*      pu8RxBuf,
                                         uint16_t*     pu16ReadCnt
                                         )
{
  uint8_t          au8CsioRxDummyBuf[SAMPLE_SPI_RX_BUFFSIZE];
  en_result_t      enResult;
  uint16_t         u16ReadCnt;

  // If Rx buffer specified NULL ...
  if (NULL == pu8RxBuf)
  {
    // Use internal buffer for dummy reading
    pu8RxBuf = au8CsioRxDummyBuf;
  }
  // Enable CS
  SampleMfsSpiEnableCs();
  // Write transmit data (blocking)
  enResult = SampleMfsSpiWrite(pu8TxBuf, u16WriteCnt);
  // Disable CS
  SampleMfsSpiDisableCs();
  if ((Ok == enResult) && (0 != u16WriteCnt))
  {
    // Wait to receive transmitted bytes.
    while (u16WriteCnt > u16RxBufFillCnt)
    {
      continue;
    }
    u16ReadCnt = u16RxBufFillCnt;
    // Read received data
    enResult = SampleMfsSpiRead(pu8RxBuf, &u16ReadCnt)
    if (Ok == enResult)
    {
      if (NULL != pu16ReadCnt)
      {
        // Set the received counts
        *pu16ReadCnt = u16ReadCnt;
      }
      // Update fill count of received buffer
      __disable_irq();
      u16RxBufFillCnt -= u16ReadCnt;
      __enable_irq();
    }
  }

  return (enResult);
}
```

```
function
{
  uint8_t           au8RxData[64];
  uint16_tu16ReadCnt;
  uint8_t           u8TxData;
  ...

  // Set CSIO Ch6_1 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR0 = FM4_GPIO->PFR0 | 0x3800;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x002A0000;
  // At first un-initialize CSIO
  (void)Mfs_Csio_DeInit(&MFS6);
  // Initialize MFS as CSIO
  if (Ok != Mfs_Csio_Init(&MFS6, (stc_mfs_csio_config_t *)&stcMfsCsioCfg))
  {
    // some code here ...
    while(1);
  }
  // Chip select port is output
  FM4_GPIO->DDR0_f.P0E  = TRUE;
  FM4_GPIO->PDOR0_f.P0E = FALSE;
  // Register interrupt handler and internal handle
  Mfs_SetTxIntCallBack(&MFS6, SampleMfsTxIrqHandler);
  Mfs_SetRxIntCallBack(&MFS6, SampleMfsRxIrqHandler);
  // Initialize variables for RX
  // (Variables for TX is initialized in SampleMfsCsioWrite())
  u16RxBufInIndex = 0;
  u16RxBufOutIndex = 0;
  u16RxBufFillCount = 0;
  u16RxBufFillCnt = 0;
  // Enable serial clock output
  Mfs_Csio_SetSckOutEnable(&MFS6, TRUE);
  // Clear possible reception errors
  Mfs_ErrorClear(&MFS6);
  // Enable TX function
  Mfs_SetTxEnable(&MFS6, TRUE);
  // Enable RX function
  Mfs_SetRxEnable(&MFS6, TRUE);
  // Enable RX interrupt
  Mfs_SetRxIntEnable(&MFS6, TRUE);
  // Init transmission interrupt
  Mfs_InitTxIrq(&MFS6);
  // Init reception interrupt
  Mfs_InitRxIrq(&MFS6);

  // some code here ...

  while(1)
  {
    if (Ok == SampleMfsCsioReadWrite(&u8TxData, 1, au8RxData, &u16ReadCnt))
    {
      // some code here ...
    }
  }
}
```

CSIO SPI master mode without using interrupt and with using CS control by peripheral function

This example software excerpt shows an usage of the CSIO SPI driver library for a master without using interrupt and with using CS control by peripheral function.

```
#include "mfs/mfs.h"

...
Configuration Structure of CSIO is same as here

static const stc_mfs_csio_cs_timing_t stcMfsCsioTiming = {
  0xFF,  // Timing for CS setup delay
  0xFF,  // Timing for CS hold delay
  20     // Minimum time from inactivation until activation of chip
            select
};
...
static en_result_t SampleMfsCsioReadWrite(uint8_t*    pu8TxBuf,
                                          uint8_t*    pu8RxBuf,
                                          uint16_t    u16TransferSize
                                         )
{
  uint16_t        u16DataToSendCounter;
  uint16_t        u16DataReceivedCounter;
  uint16_t        u16Data;
  volatile uint8_t u8Reg;

  // Check for valid pointers
  if (((NULL == pu8TxBuf) && (NULL == pu8RxBuf))
  // Check for 0 < transmission length
  ||  (0 == u16TransferSize)
    )
  {
    return (ErrorInvalidParameter);
  }

  // Check if transfer size is over maximum for TBYTE0
  if (MFS_CSIO_TBYTE_MAX < u16TransferSize)
  {
    return (ErrorInvalidParameter);
  }

  // Disable TX to set size of activation chip select an
  Mfs_SetTxEnable(&MFS6, FALSE);
  // Set size to active chip select
  Mfs_Csio_SetTxLength(&MFS6, (uint8_t)u16TransferSize);
  ...
```

```
if (TRUE == bCsHolding)
{
  // Hold chip select
  Mfs_Csio_SetCsHold(&MFS6, TRUE);
}
else
{
  // In-active chip select when transmit is end
  Mfs_Csio_SetCsHold(&MFS6, FALSE);
}
// Enable TX
Mfs_SetTxEnable(&MFS6, TRUE);

u16DataToSendCounter   = 0;
u16DataReceivedCounter = 0;

while (u16DataReceivedCounter != u16TransferSize)
{
  if (u16DataToSendCounter < u16TransferSize)
  {
    do
    {
      // If Transmit Data Register (TDR) is empty
      u8Reg = Mfs_GetStatus(&MFS6, MFS_CSIO_SSR_TDRE);
    // Wait for TDR empty
    } while (0 == u8Reg);
    // If pu8TxData is NULL, dummy data is sent.
    u16Data = 0;
    if (NULL != pu8TxBuf)
    {
      u16Data = (uint16_t)pu8TxBuf[u16DataToSendCounter];
    }
    // Write the data to Transmit Data Register
    Mfs_WriteData(&MFS6, u16Data);
    u16DataToSendCounter++;
  }
  ...
```

```
    ...
    // To avoid inter-byte spikes on SOT, send out 2nd byte
immediately,
    //   if TDR is empty!
    if ((1 == u16DataToSendCounter) && (1 < u16TransferSize))
    {
      do
      {
        // If Transmit Data Register (TDR) is empty
        u8Reg = Mfs_GetStatus(&MFS6, MFS_CSIO_SSR_TDRE);
      // Wait for TDR empty
      } while (0 == u8Reg);
      // If pu8TxData is NULL, dummy data is sent.
      u16Data = 0;
      if (NULL != pu8TxBuf)
      {
        u16Data = (uint16_t)pu8TxBuf[u16DataToSendCounter];
      }
      // Write the data to Transmit Data Register
      Mfs_WriteData(&MFS6, u16Data);
      u16DataToSendCounter++;
    }
    do
    {
      // Check reception
      u8Reg = Mfs_GetStatus(&MFS6, 0xFF);
    // wait for reception finsihed
    } while (0 == (u8Reg & MFS_CSIO_SSR_RDRF));
    // Check error (OVR)
    if (0 != (u8Reg & MFS_CSIO_SSR_ERR))
    {
      // Clear possible reception error
      Mfs_ErrorClear(&MFS6);
    }
    // If pu8RxData is NULL, dummy data is received.
    u16Data = Mfs_ReadData(&MFS6);
    if (NULL != pu8RxBuf)
    {
      pu8RxBuf[u16DataReceivedCounter] = u16Data;
    }
    u16DataReceivedCounter++;
  }

  return (Ok);
}
```

```
function
{
  ...
  uint8_t          u8RxData;
  uint8_t          u8TxData;

  // Set CSIO Ch6_1 Port (SIN, SOT, SCK, SCS)
  FM4_GPIO->PFR0    = FM4_GPIO->PFR0 | 0x7800;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x002A0000;
  FM4_GPIO->EPFR16 = FM4_GPIO->EPFR16 | 0x00000002;

  // At first un-initialize CSIO
  (void)Mfs_Csio_DeInit(&MFS6);

  // Initialize MFS as CSIO
  if (Ok != Mfs_Csio_Init(&MFS6, (stc_mfs_csio_config_t
*)&stcMfsCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // Enable Chip Select
  Mfs_Csio_SetChipSelectEnable(&MFS6, TRUE);
  // Enable or disable chip select output
  Mfs_Csio_SetChipSelectOutEnable(&MFS6, TRUE);
  // Set prescale for chip select timing
  Mfs_Csio_SetCsTimingPrescale(&MFS6, MFS_SCRCR_CDIV_64);
  // Set configuration of chip select timings
  Mfs_Csio_SetCsTimingConfig(&MFS6, (stc_mfs_csio_cs_timing_t
*)&stcMfsCsioTiming);
  // Enable serial clock output
  Mfs_Csio_SetSckOutEnable(&MFS6, TRUE);
  // Clear possible reception errors
  Mfs_ErrorClear(&MFS6);
  // Enable TX function
  Mfs_SetTxEnable(&MFS6, TRUE);
  // Enable RX function
  Mfs_SetRxEnable(&MFS6, TRUE);

  // some code here ...

  while(1)
  {
    if (Ok == SampleMfsCsioReadWrite(&u8TxData, &u8RxData, 1))
    {
      // some code here ...
    }
  }
}
```

CSIO SPI master mode with using both interrupt and CS control by peripheral function

This example software excerpt shows an usage of the CSIO SPI driver library for a master with using both interrupt and CS control by peripheral function.

```
#include "mfs/mfs.h"

...
Configuration Structure of CSIO is same as here

Serial Chip Select timing configuration structure is same as here.
...
SampleMfsTxIrqHandler() and SampleMfsRxIrqHandler() are same as here.

SampleMfsSpiRead() is same as here.
```

```
static en_result_t SampleMfsSpiWrite(uint8_t*pu8TxBuf,
                                  uint16_t        u16WriteCnt,
                                  boolean_t       bCsHolding
                                 )
{
  // Check for valid pointer
  if (NULL == pu8TxBuf)
  {
    return (ErrorInvalidParameter);
  }

  // Check if transfer size is over maximum for TBYTE0
  if (MFS_CSIO_TBYTE_MAX < u16WriteCnt)
  {
    return (ErrorInvalidParameter);
  }

  // Check if nothing to do
  if (0 == u16WriteCnt)
  {
    return (Ok);
  }

  // Disable transmit interrupt
  Mfs_SetTxIntEnable(&MFS6, FALSE);
  // Set tx data area
  pu8CsioTxBuf = pu8TxBuf;
  // Set tx data fill count (to transmit by interrupt)
  u16TxBufFillCount = u16WriteCnt;
  // Initialize output index
  u16TxBufOutIndex = 0;
  // Disable TX to set size of activation chip select and hold chip select
setting
  Mfs_SetTxEnable(&MFS6, FALSE);
  // Set size to active chip select
  Mfs_Csio_SetTxLength(&MFS6, (uint8_t)u16WriteCnt);
  if (TRUE == bCsHolding)
  {
    // Hold chip select
    Mfs_Csio_SetCsHold(&MFS6, TRUE);
  }
  else
  {
    // In-active chip select when transmit is end
    Mfs_Csio_SetCsHold(&MFS6, FALSE);
  }
  // Enable TX
  Mfs_SetTxEnable(&MFS6, TRUE);
  // Enable transmit interrupt
  Mfs_SetTxIntEnable(&MFS6, TRUE);
  // Wait until all data has been tranferred to the MFS
  // This is fully INT driven
  while (0 != u16TxBufFillCount);

  return (Ok);
}
```

```
static en_result_t SampleMfsSpiReadWrite(uint8_t*     pu8TxBuf,
                                         uint16_t     u16WriteCnt,
                                         uint8_t*     pu8RxBuf,
                                         uint16_t*    pu16ReadCnt
                                         )
{
  uint8_t          au8CsioRxDummyBuf[SAMPLE_SPI_RX_BUFFSIZE];
  en_result_t      enResult;
  uint16_t         u16ReadCnt;

  // If Rx buffer specified NULL ...
  if (NULL == pu8RxBuf)
  {
    // Use internal buffer for dummy reading
    pu8RxBuf = au8CsioRxDummyBuf;
  }
  // Write transmit data (blocking)
  enResult = SampleMfsSpiWrite(pu8TxBuf, u16WriteCnt);
  if ((Ok == enResult) && (0 != u16WriteCnt))
  {
    // Wait to receive transmitted bytes.
    while (u16WriteCnt > u16RxBufFillCnt)
    {
      continue;
    }
    u16ReadCnt = u16RxBufFillCnt;
    // Read received data
    enResult = SampleMfsSpiRead(pu8RxBuf, &u16ReadCnt)
    if (Ok == enResult)
    {
      if (NULL != pu16ReadCnt)
      {
        // Set the received counts
        *pu16ReadCnt = u16ReadCnt;
      }
      // Update fill count of received buffer
      __disable_irq();
      u16RxBufFillCnt -= u16ReadCnt;
      __enable_irq();
    }
  }

  return (enResult);
}
```

```
function
{
  uint8_t          au8RxData[64];
  ...
  uint16_t         u16ReadCnt;
  uint8_t          u8TxData;

  // Set CSIO Ch6_1 Port (SIN, SOT, SCK, SCS)
  FM4_GPIO->PFR0    = FM4_GPIO->PFR0 | 0x7800;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x002A0000;
  FM4_GPIO->EPFR16 = FM4_GPIO->EPFR16 | 0x00000002;

  // At first un-initialize CSIO
  (void)Mfs_Csio_DeInit(&MFS6);

  // Initialize MFS as CSIO
  if (Ok != Mfs_Csio_Init(&MFS6, (stc_mfs_csio_config_t
*)&stcMfsCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // Register interrupt handler and internal handle
  Mfs_SetTxIntCallBack(&MFS6, SampleMfsTxIrqHandler);
  Mfs_SetRxIntCallBack(&MFS6, SampleMfsRxIrqHandler);

  / Initialize variables for RX
  // (Variables for TX is initialized in SampleMfsSpiWrite())
  u16RxBufInIndex = 0;
  u16RxBufOutIndex = 0;
  u16RxBufFillCount = 0;
  u16RxBufFillCnt = 0;
  ...
```

```
  ...
  // Enable Chip Select
  Mfs_Csio_SetChipSelectEnable(&MFS6, TRUE);
  // Enable or disable chip select output
  Mfs_Csio_SetChipSelectOutEnable(&MFS6, TRUE);
  // Set prescale for chip select timing
  Mfs_Csio_SetCsTimingPrescale(&MFS6, MFS_SCRCR_CDIV_64);
  // Set configuration of chip select timings
  Mfs_Csio_SetCsTimingConfig(&MFS6, (stc_mfs_csio_cs_timing_t
*)&stcMfsCsioTiming);
  // Enable serial clock output
  Mfs_Csio_SetSckOutEnable(&MFS6, TRUE);
  // Clear possible reception errors
  Mfs_ErrorClear(&MFS6);
  // Enable TX function
  Mfs_SetTxEnable(&MFS6, TRUE);
  // Enable RX function
  Mfs_SetRxEnable(&MFS6, TRUE);
  // Enable RX interrupt
  Mfs_SetRxIntEnable(&MFS6, TRUE);
  // Init transmission interrupt
  Mfs_InitTxIrq(&MFS6);
  // Init reception interrupt
  Mfs_InitRxIrq(&MFS6);

  // some code here ...

  while(1)
  {
    if (Ok == SampleMfsSpiReadWrite(&u8TxData, 1, au8RxData,
&u16ReadCnt))
    {
      // some code here ...
    }
  }
```

CSIO SPI slave mode with using interrupt and without using CS control

This example software excerpt shows an usage of the CSIO SPI driver library for a slave with using interrupt and without using CS control.

```
#include "mfs/mfs.h"

...
static const stc_mfs_csio_config_t stcMfsCsioCfg = {
  2000000,                   // Baud rate (Not effective in slave mode)
  MfsCsioSlave,              // Slave mode
  MfsCsioActSpiMode,         // CSIO SPI mode
  MfsSyncWaitZero,           // Non wait time insersion(Not effective in
                                slave mode)
  MfsEightBits,              // 8 data bits
  FALSE,                     // LSB first
  TRUE                       // SCK Mark level High
};

...
SampleMfsTxIrqHandler() is same as here.

SampleMfsRxIrqHandler() and SampleMfsCsioRead() are same as here.

SampleMfsSpiRead() is same as here.

SampleMfsSpiWrite() is same as SampleMfsCsioWrite() of here.

SampleMfsSpiReadWrite() is same as here.
```

```
function
{
  uint8_t           au8RxData[64];
  uint16_tu16ReadCnt;
  uint8_t           u8TxData;

  // Set CSIO Ch6_1 Port (SIN, SOT, SCK)
  FM4_GPIO->PFR0    = FM4_GPIO->PFR0 | 0x3800;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x002A0000;

  // At first un-initialize CSIO
  (void)Mfs_Csio_DeInit(&MFS6);
  // Initialize MFS as CSIO
  if (Ok != Mfs_Csio_Init(&MFS6, (stc_mfs_csio_config_t *)&stcMfsCsioCfg))
  {
    // some code here ...
    while(1);
  }
  // Register interrupt handler and internal handle
  Mfs_SetTxIntCallBack(&MFS6, SampleMfsTxIrqHandler);
  Mfs_SetRxIntCallBack(&MFS6, SampleMfsRxIrqHandler);

  // Initialize variables for RX
  // (Variables for TX is initialized in SampleMfsCsioWrite())
  u16TxBufInIndex = 0;
  u16TxBufOutIndex = 0;
  u16TxBufFillCount = 0;
  u16RxBufInIndex = 0;
  u16RxBufOutIndex = 0;
  u16RxBufFillCount = 0;
  u16RxBufFillCnt = 0;

  // Clear possible reception errors
  Mfs_ErrorClear(&MFS6);
  // Enable TX function
  Mfs_SetTxEnable(&MFS6, TRUE);
  // Enable RX function
  Mfs_SetRxEnable(&MFS6, TRUE);
  // Enable RX interrupt
  Mfs_SetRxIntEnable(&MFS6, TRUE);
  // Init transmission interrupt
  Mfs_InitTxIrq(&MFS6);
  // Init reception interrupt
  Mfs_InitRxIrq(&MFS6);

  // some code here ...

  while(1)
  {
    if (Ok == SampleMfsSpiReadWrite(&u8TxData, 1, au8RxData, &u16ReadCnt))
    {
      // some code here ...
    }
  }
}
```

CSIO SPI slave mode with using both interrupt and CS control

This example software excerpt shows an usage of the CSIO SPI driver library for a slave with using both interrupt and CS control.

```
#include "mfs/mfs.h"

...
Configuration Structure of CSIO is same as here
};

...
SampleMfsTxIrqHandler(), SampleMfsRxIrqHandler(), SampleMfsCsioRead(),
SampleMfsSpiRead(), SampleMfsSpiWrite(), SampleMfsSpiReadWrite() are
same as here.
...
function
{
  uint8_t          au8RxData[64];
  uint16_t         u16ReadCnt;
  uint8_t          u8TxData;

  // Set CSIO Ch6_1 Port (SIN, SOT, SCK, SCS)
  FM4_GPIO->PFR0    = FM4_GPIO->PFR0 | 0x7800;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x002A0000;
  FM4_GPIO->EPFR16 = FM4_GPIO->EPFR16 | 0x00000002;

  // At first un-initialize CSIO
  (void)Mfs_Csio_DeInit(&MFS6);

  // Initialize MFS as CSIO
  if (Ok != Mfs_Csio_Init(&MFS6, (stc_mfs_csio_config_t
*)&stcMfsCsioCfg))
  {
    // some code here ...
    while(1);
  }

  // Register interrupt handler and internal handle
  Mfs_SetTxIntCallBack(&MFS6, SampleMfsTxIrqHandler);
  Mfs_SetRxIntCallBack(&MFS6, SampleMfsRxIrqHandler);

  / Initialize variables for RX
  //  (Variables for TX is initialized in SampleMfsSpiWrite())
  u16TxBufInIndex    = 0;
  u16TxBufOutIndex   = 0;
  u16TxBufFillCount  = 0;
  u16RxBufInIndex    = 0;
  u16RxBufOutIndex   = 0;
  u16RxBufFillCount = 0;
  u16RxBufFillCnt    = 0;
  ...
```

```
  ...
  // Enable Chip Select
  Mfs_Csio_SetChipSelectEnable(&MFS6, TRUE);
  // Set chip select in-active level (active level HIGH)
  Mfs_Csio_SetCsInActiveLevel(&MFS6, FALSE);
  // Clear possible reception errors
  Mfs_ErrorClear(&MFS6);
  // Enable TX function
  Mfs_SetTxEnable(&MFS6, TRUE);
  // Enable RX function
  Mfs_SetRxEnable(&MFS6, TRUE);
  // Enable RX interrupt
  Mfs_SetRxIntEnable(&MFS6, TRUE);
  // Init transmission interrupt
  Mfs_InitTxIrq(&MFS6);
  // Init reception interrupt
  Mfs_InitRxIrq(&MFS6);

  // some code here ...

  while(1)
  {
    if (Ok == SampleMfsSpiReadWrite(&u8TxData, 1, au8RxData,
&u16ReadCnt))
    {
      // some code here ...
    }
  }
}
```

7.22.3.2 I²C

I²C master mode with using blocking process

This example software excerpt shows an usage of the I²C driver library for a master with using blocking process.

```
#include "mfs/mfs.h"

...
static const stc_mfs_i2c_config_t stcMfsI2cCfg = {
  100000,                     // Baud rate
  MfsI2cMaster,               // Master mode
  MfsI2cNoizeFilterLess100M,//Noise filter setting (APB1:80MHz)
  0x00,                       // Slave address (This is not effective on
                                 master mode)
  FALSE                       // Disable Fast mode plus (Standard-mode)
};

...
```

```
static void SampleMfsTxIrqHandler(volatile stc_mfsn_t*pstcI2c,
                                              void* pvHandle)
{
  union
  {
    uint8_t                          u8SMR;
    stc_mfs_smr_field_t              stcSMR;
  } unSMR;
  union
  {
    uint8_t                          u8SSR;
    stc_mfs_ssr_field_t              stcSSR;
  } unSSR;
  union
  {
    uint8_t                          u8IBSR;
    stc_mfs_i2c_ibsr_field_t         stcIBSR;
  } unIBSR;
  union
  {
    uint8_t                          u8IBCR;
    stc_mfs_i2c_ibcr_field_t         stcIBCR;
  } unIBCR;
  union
  {
    uint8_t                          u8ISMK;
    stc_mfs_i2c_ismk_field_t         stcISMK;
  } unISMK;

  unIBSR.u8IBSR = Mfs_GetIBSR(pstcI2c);
  unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);

  /* stop condition */
  if (TRUE == unIBSR.stcIBSR.SPC)
  {
    /* Clear stop condition interrupt */
    unIBSR.stcIBSR.SPC = FALSE;
    Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);
    /* stop condition interrupt disable */
    unIBCR.stcIBCR.CNDE = FALSE;
    unIBCR.stcIBCR.ACT_SCC = FALSE;
    Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
    /* Clear IBSR:RACK */
    unISMK.u8ISMK = Mfs_GetISMK(pstcI2c);
    unISMK.stcISMK.EN = FALSE;
    Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);
    /* Restart */
    unISMK.stcISMK.EN = TRUE;
    Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);
    /* stop condition */
    u8Exec = SampleMfsExecStby;
  }
  ...
```

```
  ...
  /* TX complete or */
  else if (((u16BufferSize == u16OutIndex)
  &&     (TRUE == unIBCR.stcIBCR.INT))
  /* NACK received */
  ||      (TRUE == unIBSR.stcIBSR.RACK))
  {
    /* Disable master mode */
    unIBCR.stcIBCR.MSS = FALSE;
    unIBCR.stcIBCR.ACT_SCC = FALSE;
    /* Disable interrupt */
    unIBCR.stcIBCR.INTE = FALSE;
    /* Clear interrupt */
    unIBCR.stcIBCR.INT = FALSE;
    /* Enable stop condition interrupt */
    unIBCR.stcIBCR.CNDE = TRUE;
    Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
  }
  else
  {
    unSSR.u8SSR = Mfs_GetSSR(pstcI2c);
    /* Transmit Data Register is Empty */
    if (TRUE == unSSR.stcSSR.TDRE)
    {
      /* tx data to register */
      Mfs_WriteData(pstcI2c, (uint16_t)pu8Buffer[u16OutIndex]);
      u16OutIndex++;
      /* Complete to transmit */
      if (u16BufferSize == u16OutIndex)
      {
        /* tx interrupt disable */
        unSMR.u8SMR = Mfs_GetSMR(pstcI2c);
        unSMR.stcSMR.TIE = FALSE;
        Mfs_SetSMR(pstcI2c, unSMR.u8SMR);
      }
    }
    /* clear interrupt */
    unIBCR.stcIBCR.ACT_SCC = FALSE;
    unIBCR.stcIBCR.INT = FALSE;
    Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
  }
}
```

```
static void SampleMfsRxIrqHandler(volatile stc_mfsn_t*pstcI2c,
                                              void* pvHandle)
{
  union
  {
    uint8_t                         u8SSR;
    stc_mfs_ssr_field_t             stcSSR;
  } unSSR;
  union
  {
    uint8_t                         u8IBSR;
    stc_mfs_i2c_ibsr_field_t        stcIBSR;
  } unIBSR;
  union
  {
    uint8_t                         u8IBCR;
    stc_mfs_i2c_ibcr_field_t        stcIBCR;
  } unIBCR;
  union
  {
    uint8_t                         u8ISMK;
    stc_mfs_i2c_ismk_field_t        stcISMK;
  } unISMK;

  unIBSR.u8IBSR = Mfs_GetIBSR(pstcI2c);
  unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);
  unSSR.u8SSR = Mfs_GetSSR(pstcI2c);

  /* stop condition */
  if (TRUE == unIBSR.stcIBSR.SPC)
  {
    /* Clear stop condition interrupt */
    unIBSR.stcIBSR.SPC = FALSE;
    Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);

    /* Disable stop condition interrupt */
    unIBCR.stcIBCR.CNDE = FALSE;
    /* Disable interrupt */
    unIBCR.stcIBCR.INTE = FALSE;

    /* Clear IBSR:RACK */
    unISMK.u8ISMK = Mfs_GetISMK(pstcI2c);
    unISMK.stcISMK.EN = FALSE;
    Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);

    /* Restart */
    unISMK.stcISMK.EN = TRUE;
    Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);

    /* stop condition */
    u8Exec = SampleMfsExecStby;
  }
  ...
```

```
  ...
  /* Received data after second byte */
  else if ((TRUE == unSSR.stcSSR.RDRF) && (FALSE == nIBSR.stcIBSR.FBT))
  {
    /* Continue until specified data length is received */
    while (u16InIndex < u16BufferSize)
    {
      unSSR.u8SSR = Mfs_GetSSR(pstcI2c);
      if (TRUE == unSSR.stcSSR.RDRF)
      {
        pu8Buffer[u16InIndex] = (uint8_t)Mfs_ReadData(pstcI2c);
        u16InIndex++;
      }
      else
      {
        /* No data */
        break;
      }
    }
    /* Complete to receive */
    if (u16InIndex == u16BufferSize)
    {
      /* Stop condition */
      unIBCR.stcIBCR.MSS  = FALSE;
      /* NACK */
      unIBCR.stcIBCR.ACKE = FALSE;
      /* stop condition interrupt enable */
      unIBCR.stcIBCR.CNDE = TRUE;

      unIBSR.u8IBSR = Mfs_GetIBSR(pstcI2c);
      /* If restart condition was detected */
      if (TRUE == unIBSR.stcIBSR.RSC)
      {
        /* Clear restart condition */
        unIBSR.stcIBSR.RSC = FALSE;
        Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);
      }
    }
  }
  /* Overrun error */
  if (TRUE == unSSR.stcSSR.ORE)
  {
    /* Clear RX error */
    Mfs_ErrorClear(pstcI2c);
  }

  /* Clear interrupt */
  unIBCR.stcIBCR.ACT_SCC = FALSE;
  unIBCR.stcIBCR.INT = FALSE;
  Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
}
```

```
static void SampleMfsStsIrqHandler(volatile stc_mfsn_t* pstcI2c,
                                               void* pvHandle)
{
  /* Transmitting */
  if (SampleMfsExecTransmitting == u8Exec)
  {
    SampleMfsTxIrqHandler(pstcI2c, pvHandle);
  }
  /* Receiving */
  else
  {
    SampleMfsRxIrqHandler(pstcI2c, pvHandle);
  }
}

static en_result_t SampleMfsI2cWaitTxRxComplete(volatile stc_mfsn_t*
pstcI2c)
{
  en_result_t      enResult = ErrorTimeout;
  volatile uint32_t        u32Count;
  uint32_t         u32MaxCnt;

  /* Set maximum counter value */
  u32MaxCnt = Mfs_GetBusClock() / 10;
  u32Coun = 0;
  while (u32Count < u32MaxCnt)
  {
    /* Wait until tx or rx is completed */
    if (SampleMfsExecStby == u8Exec)
    {
      enResult = Ok;
      break;
    }
    u32Count++;
  }
  u8Exec = SampleMfsExecStby;

  return (enResult);
}
```

```
static en_result_t SampleMfsI2cRead(    uint8_t   u8SlvAddr,
                                                  uint8_t* pu8RxBuf,
                                                  uint16_t* pu16ReadCnt)
{
  en_result_t                           enResult;
  union
  {
    uint8_t                             u8SMR;
    stc_mfs_smr_field_t                 stcSMR;
  } unSMR;
  union
  {
    uint8_t                             u8IBCR;
    stc_mfs_i2c_ibcr_field_t  stcIBCR;
  } unIBCR;

  /* Check for valid pointers */
  if ((NULL == pu8RxBuf) || (NULL == pu16ReadCnt))
  {
    return (ErrorInvalidParameter);
  }
  /* Preset buffer */
  pu8Buffer = pu8RxBuf;
  u16BufferSize = *pu16ReadCnt;
  u16InIndex = 0;
  /* Change state */
  u8Exec = SampleMfsExecReceiving;
  /* Write slave address,  bit0 = 1 (rx) */
  Mfs_WriteData(&MFS2, (uint16_t)(((uint8_t)u8SlvAddr << 1) | MfsI2cRead));
  unIBCR.u8IBCR = Mfs_GetIBCR(&MFS2);
  /* Enable ACK */
  unIBCR.stcIBCR.ACKE = TRUE;
  /* wait select */
  unIBCR.stcIBCR.WSEL = TRUE;
  /* interrup enable */
  unIBCR.stcIBCR.INTE = TRUE;
  /* clear interrupt */
  unIBCR.stcIBCR.ACT_SCC = FALSE;
  unIBCR.stcIBCR.INT = FALSE;
  /* Set master mode */
  unIBCR.stcIBCR.MSS = TRUE;
  Mfs_SetIBCR(&MFS2, unIBCR.u8IBCR);
  /* Wait until TX is completed or error occur */
  enResult = SampleMfsI2cWaitTxRxComplete(&MFS2);
  if ((Ok != enResult) || (0 == u16InIndex))
  {
    *pu16ReadCnt = 0;
    return (ErrorTimeout);
  }
  /* Set received bytes */
  *pu16ReadCnt = u16InIndex;

  return (Ok);
}
```

```
static en_result_t SampleMfsI2cWrite(       uint8_t u8SlvAddr,
                                            uint8_t* pu8TxBuf,
                                            uint16_t u16WriteCnt)
{
  en_result_t                       enResult;
  union
  {
    uint8_t                         u8SMR;
    stc_mfs_smr_field_t             stcSMR;
  } unSMR;
  union
  {
    uint8_t                         u8IBCR;
    stc_mfs_i2c_ibcr_field_t        stcIBCR;
  } unIBCR;

  /* Check for valid pointer */
  if (NULL == pu8TxBuf)
  {
    return (ErrorInvalidParameter);
  }
  /* Check if nothing to do */
  if (0 == u16WriteCnt)
  {
    return (Ok);
  }

  /* Preset buffer */
  pu8Buffer = pu8TxBuf;
  u16BufferSize = u16WriteCnt;
  u16OutIndex = 0;
  /* Change state */
  u8Exec = SampleMfsExecTransmitting;
  /* Write slave address,  bit0 = 0 (tx) */
  Mfs_WriteData(&MFS2, (uint16_t)(((uint8_t)u8SlvAddr << 1) |
MfsI2cWrite));
  unIBCR.u8IBCR = Mfs_GetIBCR(&MFS2);
  /* Set master mode */
  unIBCR.stcIBCR.MSS = TRUE;
  /* Enable ACK */
  unIBCR.stcIBCR.ACKE = TRUE;
  /* Enable interrupt */
  unIBCR.stcIBCR.INTE = TRUE;
  unIBCR.stcIBCR.ACT_SCC = FALSE;
  /* wait select */
  unIBCR.stcIBCR.WSEL = FALSE;
  Mfs_SetIBCR(&MFS2, unIBCR.u8IBCR);
  /* tx interrupt enable : interruption occur */
  unSMR.u8SMR = Mfs_GetSMR(&MFS2);
  unSMR.stcSMR.TIE = TRUE;
  Mfs_SetSMR(&MFS2, unSMR.u8SMR);
  ...
```

```
  ...
  /* Wait until TX is completed */
  enResult = SampleMfsI2cWaitTxRxComplete(&MFS2);
  if (0 == u16OutIndex)
  {
      enResult = ErrorTimeout;
  }

  return (enResult);
}
```

```
function
{
  uint16_t u16TxRxCnt;
  ...

  // Set I2C Ch2_1 Port (SOT, SCK)
  FM4_GPIO->PFR2    = FM4_GPIO->PFR2 | 0x0060;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00280000;

  // At first un-initialize I2C
  (void)Mfs_I2c_DeInit(&MFS2);

  // Initialize MFS ch.2 as I2C master
  if (Ok != Mfs_I2c_Init(&MFS2, (stc_mfs_i2c_config_t *)&stcMfsI2cCfg))
  {
    // some code here ...
    while(1);
  }

  // Initialize state
  u8Exec = SampleMfsExecStby;
  // Register interrupt handler and internal handle
  Mfs_SetTxIntCallBack(&MFS2, SampleMfsTxIrqHandler);
  Mfs_SetRxIntCallBack(&MFS2, SampleMfsRxIrqHandler);
  Mfs_SetStsIntCallBack(&MFS2, SampleMfsStsIrqHandler);
  // Init transmission interrupt
  Mfs_InitTxIrq(&MFS2);
  // Init reception interrupt
  Mfs_InitRxIrq(&MFS2);

  // some code here ...

  while (1)
  {
    // Data send to slave
    if (Ok == SampleMfsI2cWrite(0x3E, (uint8_t *)au8TxData, 4))
    {
      // some code here ...
      u16TxRxCnt = 5;
      // Data receive from slave
      if (Ok == SampleMfsI2cRead(0x3E, au8RxData, &u16TxRxCnt))
      {
          // some code here ...
      }
      else
      {
          // some code here ...
      }
    }
  }
}
```

I²C master mode with using non-blocking process

This code excerpt shows how to use I²C master mode with using non-blocking process.

```
#include "mfs/mfs.h"

...
Configuration Structure of I2C is same as here
...

static void SampleMfsTxIrqHandler(volatile stc_mfsn_t*pstcI2c,
                                  void*pvHandle
                                  )
{
  union
  {
    uint8_t                        u8SMR;
    stc_mfs_smr_field_t            stcSMR;
  } unSMR;
  union
  {
    uint8_t                        u8SSR;
    stc_mfs_ssr_field_t            stcSSR;
  } unSSR;
  union
  {
    uint8_t                        u8IBSR;
    stc_mfs_i2c_ibsr_field_t       stcIBSR;
  } unIBSR;
  union
  {
    uint8_t                        u8IBCR;
    stc_mfs_i2c_ibcr_field_t       stcIBCR;
  } unIBCR;
  union
  {
    uint8_t                        u8ISMK;
    stc_mfs_i2c_ismk_field_t       stcISMK;
  } unISMK;

  unIBSR.u8IBSR = Mfs_GetIBSR(pstcI2c);
  unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);
  ...
```

```
...
/* stop condition */
if (TRUE == unIBSR.stcIBSR.SPC)
{
  /* Clear stop condition interrupt */
  unIBSR.stcIBSR.SPC = FALSE;
  Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);
  /* stop condition interrupt disable */
  unIBCR.stcIBCR.CNDE = FALSE;
  unIBCR.stcIBCR.ACT_SCC = FALSE;
  Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
  /* Clear IBSR:RACK */
  unISMK.u8ISMK = Mfs_GetISMK(pstcI2c);
  unISMK.stcISMK.EN = FALSE;
  Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);
  /* Restart */
  unISMK.stcISMK.EN = TRUE;
  Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);
  /* stop condition */
  u8Exec = SampleMfsExecStby;
  /* Set sent count */
  u16TxRxCnt = u16OutIndex;
}
/* TX complete or */
else if (((u16BufferSize == u16OutIndex)
&&     (TRUE == unIBCR.stcIBCR.INT))
/* NACK received */
||      (TRUE == unIBSR.stcIBSR.RACK))
{
  /* Disable master mode */
  unIBCR.stcIBCR.MSS = FALSE;
  unIBCR.stcIBCR.ACT_SCC = FALSE;
  /* Disable interrupt */
  unIBCR.stcIBCR.INTE = FALSE;
  /* Clear interrupt */
  unIBCR.stcIBCR.INT = FALSE;
  /* Enable stop condition interrupt */
  unIBCR.stcIBCR.CNDE = TRUE;
  Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
}
...
```

```
  ...
  else
  {
    unSSR.u8SSR = Mfs_GetSSR(pstcI2c);
    /* Transmit Data Register is Empty */
    if (TRUE == unSSR.stcSSR.TDRE)
    {
      /* tx data to register */
      Mfs_WriteData(pstcI2c, (uint16_t)pu8Buffer[u16OutIndex]);
      u16OutIndex++;
      /* Complete to transmit */
      if (u16BufferSize == u16OutIndex)
      {
        /* tx interrupt disable */
        unSMR.u8SMR = Mfs_GetSMR(pstcI2c);
        unSMR.stcSMR.TIE = FALSE;
        Mfs_SetSMR(pstcI2c, unSMR.u8SMR);
      }
    }
    /* clear interrupt */
    unIBCR.stcIBCR.ACT_SCC = FALSE;
    unIBCR.stcIBCR.INT = FALSE;
    Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
  }
}
```

```
static void SampleMfsRxIrqHandler(volatile stc_mfsn_t*pstcI2c,
                                             void* pvHandle)
{
  union
  {
    uint8_t                          u8SSR;
    stc_mfs_ssr_field_t              stcSSR;
  } unSSR;
  union
  {
    uint8_t                          u8IBSR;
    stc_mfs_i2c_ibsr_field_t         stcIBSR;
  } unIBSR;
  union
  {
    uint8_t                          u8IBCR;
    stc_mfs_i2c_ibcr_field_t         stcIBCR;
  } unIBCR;
  union
  {
    uint8_t                          u8ISMK;
    stc_mfs_i2c_ismk_field_t         stcISMK;
  } unISMK;

  unIBSR.u8IBSR = Mfs_GetIBSR(pstcI2c);
  unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);
  unSSR.u8SSR = Mfs_GetSSR(pstcI2c);

  /* stop condition */
  if (TRUE == unIBSR.stcIBSR.SPC)
  {
    /* Clear stop condition interrupt */
    unIBSR.stcIBSR.SPC = FALSE;
    Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);
    /* Disable stop condition interrupt */
    unIBCR.stcIBCR.CNDE = FALSE;
    /* Disable interrupt */
    unIBCR.stcIBCR.INTE = FALSE;
    /* Clear IBSR:RACK */
    unISMK.u8ISMK = Mfs_GetISMK(pstcI2c);
    unISMK.stcISMK.EN = FALSE;
    Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);
    /* Restart */
    unISMK.stcISMK.EN = TRUE;
    Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);
    /* stop condition */
    u8Exec = SampleMfsExecStby;
    /* Set received count */
    u16TxRxCnt = u16InIndex;
  }
  ...
```

```
  ...
  /* Received data after second byte */
  else if ((TRUE == unSSR.stcSSR.RDRF) && (FALSE ==nIBSR.stcIBSR.FBT))
  {
    /* Continue until specified data length is received */
    while (u16InIndex < u16BufferSize)
    {
      unSSR.u8SSR = Mfs_GetSSR(pstcI2c);
      if (TRUE == unSSR.stcSSR.RDRF)
      {
        pu8Buffer[u16InIndex] = (uint8_t)Mfs_ReadData(pstcI2c);
        u16InIndex++;
      }
      else
      {
        /* No data */
        break;
      }
    }
    /* Complete to receive */
    if (u16InIndex == u16BufferSize)
    {
      /* Stop condition */
      unIBCR.stcIBCR.MSS  = FALSE;
      /* NACK */
      unIBCR.stcIBCR.ACKE = FALSE;
      /* stop condition interrupt enable */
      unIBCR.stcIBCR.CNDE = TRUE;

      unIBSR.u8IBSR = Mfs_GetIBSR(pstcI2c);
      /* If restart condition was detected */
      if (TRUE == unIBSR.stcIBSR.RSC)
      {
        /* Clear restart condition */
        unIBSR.stcIBSR.RSC = FALSE;
        Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);
      }
    }
  }
  /* Overrun error */
  if (TRUE == unSSR.stcSSR.ORE)
  {
    /* Clear RX error */
    Mfs_ErrorClear(pstcI2c);
  }

  /* Clear interrupt */
  unIBCR.stcIBCR.ACT_SCC = FALSE;
  unIBCR.stcIBCR.INT = FALSE;
  Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
}

SampleMfsStsIrqHandler() is same as here.
```

```
static en_result_t SampleMfsI2cWaitTxRxComplete(
                             volatile stc_mfsn_t* pstcI2c,
                                    uint32_t u32MaxCnt)
{
  /* If tx or rx is completed, return OK */
  if (SampleMfsExecStby == u8Exec)
  {
    enResult = Ok;
  }
  else
  {
    enResult = ErrorOperationInProgress;
    u32I2cProcCnt++;
    /* If tx or rx is proceeding, polling counter counts */
    if (u32MaxCnt <= u32I2cProcCnt)
    {
      enResult = ErrorTimeout;
      u8Exec = SampleMfsExecStby;
    }
  }

  return (enResult);
}
```

```
static en_result_t SampleMfsI2cRead(uint8_t u8SlvAddr,
                                    uint8_t* pu8RxBuf,
                                    uint16_t* pu16ReadCnt)
{
  union
  {
    uint8_t                          u8SMR;
    stc_mfs_smr_field_t              stcSMR;
  } unSMR;
  union
  {
    uint8_t                          u8IBCR;
    stc_mfs_i2c_ibcr_field_t         stcIBCR;
  } unIBCR;

  /* Check for valid pointers */
  if ((NULL == pu8RxBuf) || (NULL == pu16ReadCnt))
  {
    return (ErrorInvalidParameter);
  }
  /* Preset buffer */
  pu8Buffer = pu8RxBuf;
  u16BufferSize = *pu16ReadCnt;
  u16InIndex = 0;

  /* Change state */
  u8Exec = SampleMfsExecReceiving;
  u32I2cProcCnt = 0;

  /* Write slave address,  bit0 = 1 (rx) */
  Mfs_WriteData(&MFS2, (uint16_t)(((uint8_t)u8SlvAddr << 1) |
MfsI2cRead));

  unIBCR.u8IBCR = Mfs_GetIBCR(&MFS2);
  /* Enable ACK */
  unIBCR.stcIBCR.ACKE = TRUE;
  /* wait select */
  unIBCR.stcIBCR.WSEL = TRUE;
  /* interrup enable */
  unIBCR.stcIBCR.INTE = TRUE;
  /* clear interrupt */
  unIBCR.stcIBCR.ACT_SCC = FALSE;
  unIBCR.stcIBCR.INT = FALSE;
  /* Set master mode */
  unIBCR.stcIBCR.MSS = TRUE;
  Mfs_SetIBCR(&MFS2, unIBCR.u8IBCR);

  return (Ok);
}
```

```
static en_result_t SampleMfsI2cWrite(uint8_t u8SlvAddr,
uint8_t* pu8TxBuf,
          uint16_t u16WriteCnt)
{
  union
  {
    uint8_t                            u8SMR;
    stc_mfs_smr_field_t                stcSMR;
  } unSMR;
  union
  {
    uint8_t                            u8IBCR;
    stc_mfs_i2c_ibcr_field_t  stcIBCR;
  } unIBCR;

  /* Check for valid pointer */
  if (NULL == pu8TxBuf)
  {
    return (ErrorInvalidParameter);
  }

  /* Check if nothing to do */
  if (0 == u16WriteCnt)
  {
    return (Ok);
  }

  /* Preset buffer */
  pu8Buffer = pu8TxBuf;
  u16BufferSize = u16WriteCnt;
  u16OutIndex = 0;
  /* Change state */
  u8Exec = SampleMfsExecTransmitting;
  u32I2cProcCnt = 0;
  /* Write slave address,  bit0 = 0 (tx) */
  Mfs_WriteData(&MFS2, (uint16_t)(((uint8_t)u8SlvAddr << 1) | MfsI2cWrite));
  unIBCR.u8IBCR = Mfs_GetIBCR(&MFS2);
  /* Set master mode */
  unIBCR.stcIBCR.MSS = TRUE;
  /* Enable ACK */
  unIBCR.stcIBCR.ACKE = TRUE;
  /* Enable interrupt */
  unIBCR.stcIBCR.INTE = TRUE;
  unIBCR.stcIBCR.ACT_SCC = FALSE;
  /* wait select */
  unIBCR.stcIBCR.WSEL = FALSE;
  Mfs_SetIBCR(&MFS2, unIBCR.u8IBCR);
  /* tx interrupt enable : interruption occur */
  unSMR.u8SMR = Mfs_GetSMR(&MFS2);
  unSMR.stcSMR.TIE = TRUE;
  Mfs_SetSMR(&MFS2, unSMR.u8SMR);

  return (Ok);
}
```

```
function
{
  en_result_t enResult;
  ...

  // Set I2C Ch2_1 Port (SOT, SCK)
  FM4_GPIO->PFR2    = FM4_GPIO->PFR2 | 0x0060;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00280000;

  ...

  // At first un-initialize I2C
  (void)Mfs_I2c_DeInit(&MFS2);

  // Initialize MFS ch.2 as I2C master
  if (Ok != Mfs_I2c_Init(&MFS2, (stc_mfs_i2c_config_t *)&stcMfsI2cCfg))
  {
    // some code here ...
    while(1);
  }

  // Initialize state
  u8Exec = SampleMfsExecStby;
  // Register interrupt handler and internal handle
  Mfs_SetTxIntCallBack(&MFS2, SampleMfsTxIrqHandler);
  Mfs_SetRxIntCallBack(&MFS2, SampleMfsRxIrqHandler);
  Mfs_SetStsIntCallBack(&MFS2, SampleMfsStsIrqHandler);
  // Init transmission interrupt
  Mfs_InitTxIrq(&MFS2);
  // Init reception interrupt
  Mfs_InitRxIrq(&MFS2);

  // some code here ...

  // Initialize state
  u8I2cState = SAMPLE_I2C_STATUS_STBY;

  while (1)
  {
    switch (u8I2cState)
    {
    case SAMPLE_I2C_STATUS_STBY:
      // some code here ...
      // Data send to slave
      if (Ok == SampleMfsI2cWrite(0x3E, (uint8_t *)au8TxData, 4))
      {
        u8I2cState = SAMPLE_I2C_STATUS_TX;
      }
      else
      {
        u8I2cState = SAMPLE_I2C_STATUS_RX_RQ;
        // some code here ...
      }
      break;
    ...
```

```
case SAMPLE_I2C_STATUS_STBY:
  // some code here ...
  // Data send to slave
  if (Ok == SampleMfsI2cWrite(0x3E, (uint8_t *)au8TxData, 4))
  {
    u8I2cState = SAMPLE_I2C_STATUS_TX;
  }
  else
  {
    u8I2cState = SAMPLE_I2C_STATUS_RX_RQ;
    // some code here ...
  }
  break;
...

case SAMPLE_I2C_STATUS_TX:
  // Check to complete TX (This is for fail safe)
  enResult = SampleMfsI2cWaitTxRxComplete(&MFS2, 10000000);
  if (Ok == enResult)
  {
    // some code here ...
    u8I2cState = SAMPLE_I2C_STATUS_RX_RQ;
  }
  else
  {
    if (ErrorOperationInProgress != enResult)
    {
      u8I2cState = SAMPLE_I2C_STATUS_RX_RQ;
      // some code here ...
    }
  }
  break;

case SAMPLE_I2C_STATUS_RX_RQ:
  // some code here ...
  u16TxRxCnt = 5;
  // Data receive from slave
  if (Ok == SampleMfsI2cRead(0x3E, au8RxData, &u16TxRxCnt))
  {
    u8I2cState = SAMPLE_I2C_STATUS_RX;
  }
  else
  {
    u8I2cState = SAMPLE_I2C_STATUS_STBY;
    // some code here ...
  }
  break;
...
```

```
    case SAMPLE_I2C_STATUS_RX:
      // Check to complete RX (This is for fail safe)
      enResult = SampleMfsI2cWaitTxRxComplete(&MFS2, 10000000);
      if (Ok == enResult)
      {
        u8I2cState = SAMPLE_I2C_STATUS_STBY;
      }
      else
      {
        if (ErrorOperationInProgress != enResult)
        {
          u8I2cState = SAMPLE_I2C_STATUS_STBY;
          // some code here ...
        }
      }
      break;

    // Fail safe
    default:
      u8I2cState = SAMPLE_I2C_STATUS_STBY;
      break;
    }
  }
}
```

I²C slave mode with using blocking process

This code excerpt shows how to use I²C slave mode with using blocking process.

```
#include "mfs/mfs.h"

...
static const stc_mfs_i2c_config_t stcMfsI2cCfg = {
  100000,                            // Baud rate (Not effective in
                                        slave mode)
  MfsI2cSlave,                       // Master mode
  MfsI2cNoizeFilterLess100M,         // Noise filter setting(APB1:80MHz)
  0x3E,                              // Slave address
  FALSE                              // Disable Fast mode plus
                                        (Standard-mode)
};
...

static uint8_t SampleMfsI2cSlvStCb(uint8_t u8Status)
{
  uint8_t u8Data = 0u;

  /* Set I2C direction status */
  u8I2cStatus = u8Status;

  /* TX */
  if (MfsI2cWrite == u8Status)
  {
    /* some code here ... */
  }

  return (u8Data);
}
```

```
static void SampleMfsTxIrqHandler(volatile stc_mfsn_t*pstcI2c,
                                               void* pvHandle)
{
  union
  {
    uint8_t                          u8SMR;
    stc_mfs_smr_field_t              stcSMR;
  } unSMR;
  union
  {
    uint8_t                          u8SSR;
    stc_mfs_ssr_field_t              stcSSR;
  } unSSR;
  union
  {
    uint8_t                          u8IBSR;
    stc_mfs_i2c_ibsr_field_t         stcIBSR;
  } unIBSR;
  union
  {
    uint8_t                          u8IBCR;
    stc_mfs_i2c_ibcr_field_t         stcIBCR;
  } unIBCR;
  union
  {
    uint8_t                          u8ISMK;
    stc_mfs_i2c_ismk_field_t         stcISMK;
  } unISMK;

  unIBSR.u8IBSR = Mfs_GetIBSR(pstcI2c);
  ...
```

```
...
/* end of data or */
if ((u16BufferSize == u16OutIndex)
/* Stop condition */
||  (TRUE == unIBSR.stcIBSR.SPC))
{
  /* Disable TX interrupt */
  unSMR.u8SMR = Mfs_GetSMR(pstcI2c);
  unSMR.stcSMR.TIE = FALSE;
  Mfs_SetSMR(pstcI2c, unSMR.u8SMR);

  /* Stop condition */
  if (TRUE == unIBSR.stcIBSR.SPC)
  {
    /* Clear IBSR:RACK */
    unISMK.u8ISMK = Mfs_GetISMK(pstcI2c);
    unISMK.stcISMK.EN = FALSE;
    Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);

    /* Restart */
    unISMK.stcISMK.EN = TRUE;
    Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);

    /* Clear stop condition interrupt */
    unIBSR.stcIBSR.SPC = FALSE;
    Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);

    /* Stop condition */
    u8Exec = SampleMfsExecStby;
  }

  /* Clear interrupt */
  unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);
  unIBCR.stcIBCR.ACT_SCC = FALSE;
  unIBCR.stcIBCR.INT = FALSE;
  Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);

  /* Clear RSC interrupt */
  unIBSR.stcIBSR.RSC = FALSE;
  Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);
}
...
```

```
  ...
  else
  {
    unSSR.u8SSR = Mfs_GetSSR(pstcI2c);
    if (TRUE == unSSR.stcSSR.TDRE)
    {
      /* Set data to TX FIFO */
      Mfs_WriteData(pstcI2c, pu8Buffer[u16OutIndex]);
      u16OutIndex += 1u;
    }
    /* Clear interrupt */
    unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);
    unIBCR.stcIBCR.ACT_SCC = FALSE;
    unIBCR.stcIBCR.INT = FALSE;
    Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
    /* Received NACK */
    if (TRUE == unIBSR.stcIBSR.RACK)
    {
      /* Clear RSC interrupt */
      unIBSR.stcIBSR.RSC = FALSE;
      Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);
    }
  }
}
```

```
static void SampleMfsI2cPreStartSlave(volatile stc_mfsn_t* pstcI2c)
{
  union
  {
    uint8_t                         u8SMR;
    stc_mfs_smr_field_t             stcSMR;
  } unSMR;
  union
  {
    uint8_t                         u8IBSR;
    stc_mfs_i2c_ibsr_field_t        stcIBSR;
  } unIBSR;
  union
  {
    uint8_t                         u8IBCR;
    stc_mfs_i2c_ibcr_field_t        stcIBCR;
  } unIBCR;
  union
  {
    uint8_t                         u8ISMK;
    stc_mfs_i2c_ismk_field_t        stcISMK;
  } unISMK;
  uint8_t                           u8Data;

  unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);
  unIBSR.u8IBSR = Mfs_GetIBSR(pstcI2c);
  ...
```

```
  ...
  /* Chk Slave Address Received */
  if ( (FALSE == unIBCR.stcIBCR.MSS) &&
     (TRUE == unIBCR.stcIBCR.ACT_SCC) &&
     (TRUE == unIBSR.stcIBSR.FBT)
    )
  {
    unSMR.u8SMR = Mfs_GetSMR(pstcI2c);
    /* TX */
    if (TRUE == unIBSR.stcIBSR.TRX)
    {
      /* Callback for the I2C slave starting(TX) and get 1st data to
         send */
      u8Data = SampleMfsI2cSlvStCb(MfsI2cWrite);
      /* Send 1st data */
      Mfs_WriteData(pstcI2c, u8Data);
      /* Disable TX interrupt */
      unSMR.stcSMR.TIE = FALSE;
      unIBCR.stcIBCR.WSEL = FALSE;
    }
    /* RX */
    else
    {
      /* Enable ACK */
      unIBCR.stcIBCR.ACKE = TRUE;
      /* Disable RX interrupt */
      unSMR.stcSMR.RIE = FALSE;
      unIBCR.stcIBCR.WSEL = TRUE;
      /* Callback for the I2C slave starting(RX) */
      SampleMfsI2cSlvStCb(MfsI2cRead);
    }
    Mfs_SetSMR(pstcI2c, unSMR.u8SMR);
  }

  /* Clear interrupt */
  unIBCR.stcIBCR.ACT_SCC = FALSE;
  unIBCR.stcIBCR.INT = FALSE;
  Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
}
```

```
static void SampleMfsI2cDataRxSlave(volatile stc_mfsn_t* pstcI2c)
{
  union
  {
    uint8_t                          u8SSR;
    stc_mfs_ssr_field_t              stcSSR;
  } unSSR;
  union
  {
    uint8_t                          u8IBSR;
    stc_mfs_i2c_ibsr_field_t         stcIBSR;
  } unIBSR;
  union
  {
    uint8_t                          u8IBCR;
    stc_mfs_i2c_ibcr_field_t         stcIBCR;
  } unIBCR;
  union
  {
    uint8_t                          u8ISMK;
    stc_mfs_i2c_ismk_field_t         stcISMK;
  } unISMK;
  uint8_t                            u8Data;

  unIBSR.u8IBSR = Mfs_GetIBSR(pstcI2c);
  unSSR.u8SSR = Mfs_GetSSR(pstcI2c);
  ...
```

```
...
/* Stop condition */
if (TRUE == unIBSR.stcIBSR.SPC)
{
  /* Clear stop condition interrupt */
  unIBSR.stcIBSR.SPC = FALSE;
  Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);

  /* Clear IBSR:RACK */
  unISMK.u8ISMK = Mfs_GetISMK(pstcI2c);
  unISMK.stcISMK.EN = FALSE;
  Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);

  /* Restart */
  unISMK.stcISMK.EN = TRUE;
  Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);

  /* Stop condition */
  u8Exec = SampleMfsExecStby;
}
/* Received NACK */
else if (TRUE == unIBSR.stcIBSR.RACK)
{
  /* Clear interrupt */
  unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);
  unIBCR.stcIBCR.ACT_SCC = FALSE;
  unIBCR.stcIBCR.INT = FALSE;
  Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
}
...
```

```
  ...
  /* Check receiving */
  else
  {
    unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);

    /* Received data */
    if ((TRUE == unSSR.stcSSR.RDRF) && (FALSE == unIBSR.stcIBSR.FBT))
    {
      /* Continue until specified data length is received */
      while (u16InIndex < u16BufferSize)
      {
        unSSR.u8SSR = Mfs_GetSSR(pstcI2c);
        if (TRUE == unSSR.stcSSR.RDRF)
        {
          pu8Buffer[u16InIndex] = (uint8_t)Mfs_ReadData(pstcI2c);
          u16InIndex += 1u;
        }
        else
        {
          /* No data */
          break;
        }
      }
      /* Complete to receive */
      if (u16InIndex == u16BufferSize)
      {
        /* Send NACK */
        unIBCR.stcIBCR.ACKE = FALSE;
      }
      else
      {
        /* Send ACK */
        unIBCR.stcIBCR.ACKE = TRUE;
      }
    }
    /* Clear interrupt */
    unIBCR.stcIBCR.ACT_SCC = FALSE;
    unIBCR.stcIBCR.INT = FALSE;
    Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
  }
  /* Overrun error */
  if (TRUE == unSSR.stcSSR.ORE)
  {
    /* Clear RX error */
    Mfs_ErrorClear(pstcI2c);
  }
}
```

```
static void SampleMfsRxIrqHandler(volatile stc_mfsn_t*pstcI2c,
                                               void* pvHandle)
{
  /* If status is standby... */
  if (SampleMfsExecStby == u8Exec)
  {
    SampleMfsI2cPreStartSlave(pstcI2c);
  }
  else
  {
    SampleMfsI2cDataRxSlave(pstcI2c);
  }
}

static void SampleMfsRxIrqHandler(volatile stc_mfsn_t*pstcI2c,
                                void*                  pvHandle
                                )
{
  switch (u8Exec)
  {
    /* Standby */
    case SampleMfsExecStby:
      SampleMfsI2cPreStartSlave(pstcI2c);
      break;
    /* Receiving (after slave address was received) */
    case SampleMfsExecReceiving:
      SampleMfsI2cDataRxSlave(pstcI2c);
      break;
    /* Transmitting */
    default:
      SampleMfsTxIrqHandler(pstcI2c, NULL);
      break;
  }
}
```

```
static en_result_t SampleMfsI2cStartSlave(volatile stc_mfsn_t* pstcI2c)
{
  en_result_t                          enResult;
  union
  {
    uint8_t                            u8SMR;
    stc_mfs_smr_field_t                stcSMR;
  } unSMR;
  union
  {
    uint8_t                            u8IBCR;
    stc_mfs_i2c_ibcr_field_t  stcIBCR;
  } unIBCR;
  union
  {
    uint8_t                            u8IBSR;
    stc_mfs_i2c_ibsr_field_t  stcIBSR;
  } unIBSR;
  unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);
  /* Slave is active */
  if ((TRUE == unIBCR.stcIBCR.ACT_SCC) && (FALSE == unIBCR.stcIBCR.MSS))
  {
    /* Set ACK */
    unIBCR.stcIBCR.ACKE = TRUE;
    unIBCR.stcIBCR.ACT_SCC = FALSE;
    Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
    unIBSR.u8IBSR = Mfs_GetIBSR(pstcI2c);
    /* Direction is TX */
    if (TRUE == unIBSR.stcIBSR.TRX)
    {
      /* tx interrupt enable */
      unSMR.u8SMR = Mfs_GetSMR(pstcI2c);
      unSMR.stcSMR.TIE = TRUE;
      Mfs_SetSMR(pstcI2c, unSMR.u8SMR);
    }
    /* Direction is RX */
    else
    {
      /* rx interrupt enable */
      unSMR.u8SMR = Mfs_GetSMR(pstcI2c);
      unSMR.stcSMR.RIE = TRUE;
      Mfs_SetSMR(pstcI2c, unSMR.u8SMR);
    }
    enResult = Ok;
  }
  else
  {
    u8Exec = SampleMfsExecStby;
    enResult = ErrorInvalidMode;
  }

  return (enResult);
}

SampleMfsI2cWaitTxRxComplete() is same as here.
```

```
static en_result_t SampleMfsI2cRead(uint8_t* pu8RxBuf,
                                    uint16_t* pu16ReadCnt)
{
  en_result_t enResult;

  /* Check for valid pointers */
  if ((NULL == pu8RxBuf) || (NULL == pu16ReadCnt))
  {
    return (ErrorInvalidParameter);
  }

  /* Preset buffer */
  pu8Buffer = pu8RxBuf;
  u16BufferSize = *pu16ReadCnt;
  u16InIndex = 0;

  /* Change state */
  u8Exec = SampleMfsExecReceiving;

  /* rx */
  SampleMfsI2cStartSlave(&MFS2);

  /* Wait until TX is completed */
  enResult = SampleMfsI2cWaitTxRxComplete(&MFS2);

  if ((Ok != enResult) || (0 == u16InIndex))
  {
    *pu16ReadCnt = 0;
    return (ErrorTimeout);
  }

  /* Set received bytes */
  *pu16ReadCnt = u16InIndex;

  return (Ok);
}
```

```
static en_result_t SampleMfsI2cWrite(uint8_t* pu8TxBuf,
                                     uint16_t u16WriteCnt)
{
  en_result_t                           enResult;
  union
  {
    uint8_t                             u8SMR;
    stc_mfs_smr_field_t                 stcSMR;
  } unSMR;
  union
  {
    uint8_t                             u8IBCR;
    stc_mfs_i2c_ibcr_field_t  stcIBCR;
  } unIBCR;

  /* Check for valid pointer */
  if (NULL == pu8TxBuf)
  {
      return (ErrorInvalidParameter);
  }
  /* Check if nothing to do */
  if (0 == u16WriteCnt)
  {
      return (Ok);
  }
  /* Preset buffer */
  pu8Buffer = pu8TxBuf;
  u16BufferSize = u16WriteCnt;
  u16OutIndex = 0;
  /* Change state */
  u8Exec = SampleMfsExecTransmitting;
  /* tx */
  SampleMfsI2cStartSlave(&MFS2);
  /* Wait until TX is completed or error occur */
  enResult = SampleMfsI2cWaitTxRxComplete(&MFS2);
  if (0 == u16OutIndex)
  {
      enResult = ErrorTimeout;
  }

  return (enResult);
}

static uint8_t SampleMfsGetI2cSlvStatus(void)
{
    uint8_t u8Status;

    __disable_irq();
    u8Status = u8I2cStatus;/* Set the slave status for return */
    u8I2cStatus = 0xee;     /* Clear slave status */
    __enable_irq();

    return (u8Status);
}
```

```
function
{
  uint16_t u16RxCnt;

  // Set I2C Ch2_1 Port (SOT, SCK)
  FM4_GPIO->PFR2   = FM4_GPIO->PFR2 | 0x0060;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00280000;
  // At first un-initialize I2C
  (void)Mfs_I2c_DeInit(&MFS2);
  // Initialize MFS ch.2 as I2C master
  if (Ok != Mfs_I2c_Init(&MFS2, (stc_mfs_i2c_config_t *)&stcMfsI2cCfg))
  {
    // some code here ...
    while(1);
  }
  ...
  // Initialize state
  u8Exec = SampleMfsExecStby;
  u8I2cStatus = 0xee;
  // Register interrupt handler and internal handle
  Mfs_SetTxIntCallBack(&MFS2, SampleMfsTxIrqHandler);
  Mfs_SetRxIntCallBack(&MFS2, SampleMfsRxIrqHandler);
  Mfs_SetStsIntCallBack(&MFS2, SampleMfsStsIrqHandler);
  /* Enable stop condition interrupt */
  Mfs_I2c_SetCondDetIntEnable(&MFS2, TRUE);
  /* Enable interrupt */
  Mfs_I2c_SetIntEnable(&MFS2, TRUE);
  // Init transmission interrupt
  Mfs_InitTxIrq(&MFS2);
  // Init reception interrupt
  Mfs_InitRxIrq(&MFS2);

  while (1)
  {
    // Get I2C status (check the request from master)
    switch (SampleMfsGetI2cSlvStatus())
    {
      // Write (Request to read from master)
      case MfsI2cWrite:
        if (Ok == SampleMfsI2cWrite(au8TxData, 4))
        ...
        // some code here ...
        break;
      // Read (Request to write from master)
      case MfsI2cRead:
        u16RxCnt = 4;
        if (Ok == SampleMfsI2cRead(au8RxData, &u16RxCnt))
        ...
        // some code here ...
        break;
    }
  }
}
```

I²C slave mode with using non-blocking process

This code excerpt shows how to use I²C slave mode with using non-blocking process.

```
#include "mfs/mfs.h"

...
Configuration Structure of I2C is same as here
...

SampleMfsI2cSlvStCb() is same as Error! Reference source not found.here.

static void SampleMfsTxIrqHandler(volatile stc_mfsn_t*pstcI2c,
                                  void* pvHandle)
{
  union
  {
    uint8_t                          u8SMR;
    stc_mfs_smr_field_t              stcSMR;
  } unSMR;
  union
  {
    uint8_t                          u8SSR;
    stc_mfs_ssr_field_t              stcSSR;
  } unSSR;
  union
  {
    uint8_t                          u8IBSR;
    stc_mfs_i2c_ibsr_field_t         stcIBSR;
  } unIBSR;
  union
  {
    uint8_t                          u8IBCR;
    stc_mfs_i2c_ibcr_field_t         stcIBCR;
  } unIBCR;
  union
  {
    uint8_t                          u8ISMK;
    stc_mfs_i2c_ismk_field_t         stcISMK;
  } unISMK;

  unIBSR.u8IBSR = Mfs_GetIBSR(pstcI2c);
  ...
```

```
...
/* end of data or */
if ((u16BufferSize == u16OutIndex)
/* Stop condition */
||  (TRUE == unIBSR.stcIBSR.SPC))
{
  /* Disable TX interrupt */
  unSMR.u8SMR = Mfs_GetSMR(pstcI2c);
  unSMR.stcSMR.TIE = FALSE;
  Mfs_SetSMR(pstcI2c, unSMR.u8SMR);

  /* Stop condition */
  if (TRUE == unIBSR.stcIBSR.SPC)
  {
    /* Clear IBSR:RACK */
    unISMK.u8ISMK = Mfs_GetISMK(pstcI2c);
    unISMK.stcISMK.EN = FALSE;
    Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);

    /* Restart */
    unISMK.stcISMK.EN = TRUE;
    Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);

    /* Clear stop condition interrupt */
    unIBSR.stcIBSR.SPC = FALSE;
    Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);
    /* Stop condition */
    u8Exec = SampleMfsExecStby;
    /* Set sent count */
    u16TxRxCnt = u16OutIndex;
  }

  /* Clear interrupt */
  unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);
  unIBCR.stcIBCR.ACT_SCC = FALSE;
  unIBCR.stcIBCR.INT = FALSE;
  Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);

  /* Clear RSC interrupt */
  unIBSR.stcIBSR.RSC = FALSE;
  Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);
}
...
```

```
  ...
  else
  {
    unSSR.u8SSR = Mfs_GetSSR(pstcI2c);
    if (TRUE == unSSR.stcSSR.TDRE)
    {
      /* Set data to TX FIFO */
      Mfs_WriteData(pstcI2c, pu8Buffer[u16OutIndex]);
      u16OutIndex += 1u;
    }
    /* Clear interrupt */
    unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);
    unIBCR.stcIBCR.ACT_SCC = FALSE;
    unIBCR.stcIBCR.INT = FALSE;
    Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
    /* Received NACK */
    if (TRUE == unIBSR.stcIBSR.RACK)
    {
      /* Clear RSC interrupt */
      unIBSR.stcIBSR.RSC = FALSE;
      Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);
    }
  }
}

SampleMfsI2cPreStartSlave() is same as here.
```

```
static void SampleMfsI2cDataRxSlave(volatile stc_mfsn_t* pstcI2c)
{
  union
  {
    uint8_t                         u8SSR;
    stc_mfs_ssr_field_t             stcSSR;
  } unSSR;
  union
  {
    uint8_t                         u8IBSR;
    stc_mfs_i2c_ibsr_field_t        stcIBSR;
  } unIBSR;
  union
  {
    uint8_t                         u8IBCR;
    stc_mfs_i2c_ibcr_field_t        stcIBCR;
  } unIBCR;
  union
  {
    uint8_t                         u8ISMK;
    stc_mfs_i2c_ismk_field_t        stcISMK;
  } unISMK;
  uint8_t                           u8Data;

  unIBSR.u8IBSR = Mfs_GetIBSR(pstcI2c);
  unSSR.u8SSR = Mfs_GetSSR(pstcI2c);
  ...
```

```
...
/* Stop condition */
if (TRUE == unIBSR.stcIBSR.SPC)
{
  /* Clear stop condition interrupt */
  unIBSR.stcIBSR.SPC = FALSE;
  Mfs_SetIBSR(pstcI2c, unIBSR.u8IBSR);

  /* Clear IBSR:RACK */
  unISMK.u8ISMK = Mfs_GetISMK(pstcI2c);
  unISMK.stcISMK.EN = FALSE;
  Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);

  /* Restart */
  unISMK.stcISMK.EN = TRUE;
  Mfs_SetISMK(pstcI2c, unISMK.u8ISMK);

  /* Stop condition */
  u8Exec = SampleMfsExecStby;

  /* Set received length */
  u16TxRxCnt = u16InIndex;
}
/* Received NACK */
else if (TRUE == unIBSR.stcIBSR.RACK)
{
  /* Clear interrupt */
  unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);
  unIBCR.stcIBCR.ACT_SCC = FALSE;
  unIBCR.stcIBCR.INT = FALSE;
  Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
}
...
```

```
  ...
  /* Check receiving */
  else
  {
    unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);

    /* Received data */
    if ((TRUE == unSSR.stcSSR.RDRF) && (FALSE == unIBSR.stcIBSR.FBT))
    {
      /* Continue until specified data length is received */
      while (u16InIndex < u16BufferSize)
      {
        unSSR.u8SSR = Mfs_GetSSR(pstcI2c);
        if (TRUE == unSSR.stcSSR.RDRF)
        {
          pu8Buffer[u16InIndex] = (uint8_t)Mfs_ReadData(pstcI2c);
          u16InIndex += 1u;
        }
        else
        {
          /* No data */
          break;
        }
      }
      /* Complete to receive */
      if (u16InIndex == u16BufferSize)
      {
        /* Send NACK */
        unIBCR.stcIBCR.ACKE = FALSE;
      }
      else
      {
        /* Send ACK */
        unIBCR.stcIBCR.ACKE = TRUE;
      }
    }
    /* Clear interrupt */
    unIBCR.stcIBCR.ACT_SCC = FALSE;
    unIBCR.stcIBCR.INT = FALSE;
    Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
  }
  /* Overrun error */
  if (TRUE == unSSR.stcSSR.ORE)
  {
    /* Clear RX error */
    Mfs_ErrorClear(pstcI2c);
  }
}

SampleMfsRxIrqHandler() and SampleMfsStsIrqHandler() are same as here.
```

```
static en_result_t SampleMfsI2cStartSlave(volatile stc_mfsn_t* pstcI2c)
{
  en_result_t                         enResult;
  union
  {
    uint8_t                           u8SMR;
    stc_mfs_smr_field_t               stcSMR;
  } unSMR;
  union
  {
    uint8_t                           u8IBCR;
    stc_mfs_i2c_ibcr_field_t  stcIBCR;
  } unIBCR;
  union
  {
    uint8_t                           u8IBSR;
    stc_mfs_i2c_ibsr_field_t  stcIBSR;
  } unIBSR;
  unIBCR.u8IBCR = Mfs_GetIBCR(pstcI2c);
  /* Slave is active */
  if ((TRUE == unIBCR.stcIBCR.ACT_SCC) && (FALSE == unIBCR.stcIBCR.MSS))
  {
    /* Set ACK */
    unIBCR.stcIBCR.ACKE = TRUE;
    unIBCR.stcIBCR.ACT_SCC = FALSE;
    Mfs_SetIBCR(pstcI2c, unIBCR.u8IBCR);
    unIBSR.u8IBSR = Mfs_GetIBSR(pstcI2c);
    /* Direction is TX */
    if (TRUE == unIBSR.stcIBSR.TRX)
    {
      /* Change state */
      u8Exec = SampleMfsExecTransmitting;
      /* tx interrupt enable */
      unSMR.u8SMR = Mfs_GetSMR(pstcI2c);
      unSMR.stcSMR.TIE = TRUE;
      Mfs_SetSMR(pstcI2c, unSMR.u8SMR);
    }
    /* Direction is RX */
    else
    {
      /* Change state */
      u8Exec = SampleMfsExecReceiving;
      /* rx interrupt enable */
      unSMR.u8SMR = Mfs_GetSMR(pstcI2c);
      unSMR.stcSMR.RIE = TRUE;
      Mfs_SetSMR(pstcI2c, unSMR.u8SMR);
    }
    u32I2cProcCnt = 0;
    enResult = Ok;
  }
  else
  {
    u8Exec = SampleMfsExecStby;
    enResult = ErrorInvalidMode;
  }
  return (enResult);
}
```

```
SampleMfsI2cWaitTxRxComplete() is same as here.

static en_result_t SampleMfsI2cRead(    uint8_t* pu8RxBuf,
                                        uint16_t u16ReadCnt)
{
  en_result_t enResult;

  /* Check for valid pointers */
  if ((NULL == pu8RxBuf)
  ||   (0 == u16ReadCnt)
     )
  {
    return (ErrorInvalidParameter);
  }

  /* Preset buffer */
  pu8Buffer = pu8RxBuf;
  u16BufferSize = u16ReadCnt;
  u16InIndex = 0;
  /* rx */
  enResult = SampleMfsI2cStartSlave(&MFS2);

  return (enResult);
}

static en_result_t SampleMfsI2cWrite(uint8_t*     pu8TxBuf,
                                 uint16_t         u16WriteCnt
                                 )
{
  en_result_t                          enResult;

  /* Check for valid pointer */
  if (NULL == pu8TxBuf)
  {
    return (ErrorInvalidParameter);
  }

  /* Check if nothing to do */
  if (0 == u16WriteCnt)
  {
    return (Ok);
  }

  /* Preset buffer */
  pu8Buffer = pu8TxBuf;
  u16BufferSize = u16WriteCnt;
  u16OutIndex = 0;
  /* tx */
  enResult = SampleMfsI2cStartSlave(&MFS2);

  return (enResult);
}

SampleMfsGetI2cSlvStatus() is same as here.
```

```
function
{
  en_result_t enResult;

  // Set I2C Ch2_1 Port (SOT, SCK)
  FM4_GPIO->PFR2    = FM4_GPIO->PFR2 | 0x0060;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00280000;
  ...
  // At first un-initialize I2C
  (void)Mfs_I2c_DeInit(&MFS2);
  // Initialize MFS ch.2 as I2C slave
  if (Ok != Mfs_I2c_Init(&MFS2, (stc_mfs_i2c_config_t *)&stcMfsI2cCfg))
  {
    // some code here ...
    while(1);
  }
  ...
  // Initialize state
  u8Exec = SampleMfsExecStby;
  u8I2cStatus = 0xee;
  u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_STANDBY;
  // Register interrupt handler and internal handle
  Mfs_SetTxIntCallBack(&MFS2, SampleMfsTxIrqHandler);
  Mfs_SetRxIntCallBack(&MFS2, SampleMfsRxIrqHandler);
  Mfs_SetStsIntCallBack(&MFS2, SampleMfsStsIrqHandler);
  /* Enable stop condition interrupt */
  Mfs_I2c_SetCondDetIntEnable(&MFS2, TRUE);
  /* Enable interrupt */
  Mfs_I2c_SetIntEnable(&MFS2, TRUE);
  // Init transmission interrupt
  Mfs_InitTxIrq(&MFS2);
  // Init reception interrupt
  Mfs_InitRxIrq(&MFS2);

  while (1)
  {
    // I2C is standby
    if (SAMPLE_MFS_I2C_STATUS_STANDBY == u8TxRxStatus)
    {
      // Get I2C status (check the request from master)
      switch (SampleMfsGetI2cSlvStatus())
      {
        // Write (Request to read from master)
        case MfsI2cWrite:
          enResult = SampleMfsI2cWrite(au8TxData, 4);
          if (Ok == enResult)
          {
            u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_TRANS;
          }
          // some code here ...
          break;
      ...
```

```
        ...
        // Read (Request to write from master)
        case MfsI2cRead:
          enResult = SampleMfsI2cRead(au8RxData, 4);
          if (Ok == enResult)
          {
            u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_RCV;
          }
          // some code here ...
          break;

        default:
          break;
      }
    }
    // TX or RX proccessing
    if (SAMPLE_MFS_I2C_STATUS_STANDBY != u8TxRxStatus)
    {
      // Transmitting
      if(SAMPLE_MFS_I2C_STATUS_TRANS == u8TxRxStatus)
      {
        // Check to complete TX (This is for fail safe)
        enResult = SampleMfsI2cWaitTxRxComplete(&MFS2, 10000000);
        if (Ok == enResult)
        {
          u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_STANDBY;
          // some code here ...
        }
        else
        {
          if (ErrorOperationInProgress != enResult)
          {
            u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_STANDBY;
            // some code here ...
          }
        }
      }
      ...
```

```
      ....
      // Receiving
      else
      {
        // Check to complete RX (This is for fail safe)
        enResult = SampleMfsI2cWaitTxRxComplete(&MFS2, 10000000);
        if (Ok == enResult)
        {
          u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_STANDBY;
          // some code here ...
        }
        else
        {
          if (ErrorOperationInProgress != enResult)
          {
            u8TxRxStatus = SAMPLE_MFS_I2C_STATUS_STANDBY;
          }
        }
      }
    }
  }
}
```

7.22.3.3 LIN

This example software excerpt shows an usage of the LIN driver library for a master and a slave (with using interrupt).

```
#include "mfs/mfs.h"

...
// Configuration for master
static const stc_mfs_lin_config_t stcMfsLinCfg1 = {
  19200,                     // Baud rate
  FALSE,                     // Un-use external wake-up function
  FALSE,                     // Disable LIN break reception interrupt
  MfsLinMaster,              // Master mode
  MfsOneStopBit,             // 1 stop bit
  MfsLinBreakLength16,       // Lin Break Length 16 Bit Times
  MfsLinDelimiterLength1     // Lin Break Delimiter Length 1 Bit Time
};

// Configuration for slav
static const stc_mfs_lin_config_t stcMfsLinCfg2 = {
  19200,                     // Baud rate (Not effective in slave mode)
  FALSE,                     // Un-use external wake-up function
  TRUE,                      // Enable LIN break reception interrupt
  MfsLinSlave,               // Slave mode
  MfsOneStopBit,             // 1 stop bit
  MfsLinBreakLength16,       // Lin Break Length 16 Bit Times
  MfsLinDelimiterLength1     // Lin Break Delimiter Length 1 Bit Time
};

...
static void SampleMfsTxIrqHandler1(volatile stc_mfsn_t* pstcMfs,
                                          void* pvHandle)
{
  // Put data from Buffer into Transmit Data Register
  Mfs_WriteData(pstcMfs, (uint16_t) pu8LinTxBuf1[u16TxBufOutIndex1]);
  // Update tail
  u16TxBufOutIndex1++;
  u16TxBufFillCount1--;

  // If no more bytes to sent ...
  if (0 == u16TxBufFillCount1)
  {
    // Disable transmission interrupt
    Mfs_SetTxIntEnable(pstcMfs, FALSE);
  }
}
```

```
static void SampleMfsRxIrqHandler1(volatile  stc_mfsn_t* pstcMfs,
                                             void* pvHandle)
{
  uint16_t        u16Data;
  volatile uint8_t u8Ssr;

  // Check Overrun error
  u8Ssr = Mfs_GetStatus(pstcMfs, MFS_LIN_SSR_ERR);
  if (0 != u8Ssr)
  {
    // Clear possible reception errors
    Mfs_ErrorClear(pstcMfs);
  }

  // Read data from Read Data Register
  u16Data = Mfs_ReadData(pstcMfs);
  // If there is empty space in RX buffer
  if (u16RxBufFillCount1 < SAMPLE_LIN_RX_BUFFSIZE)
  {
    // Store read data to RX buffer
    au8LinRxBuf1[u16RxBufInIndex1] = (uint8_t)u16Data;
    // Update input index
    u16RxBufInIndex1++;
    if (SAMPLE_LIN_RX_BUFFSIZE <= u16RxBufInIndex1)
    {
      u16RxBufInIndex1 = 0;
    }
    // Count bytes in RX-FIFO
    u16RxBufFillCount1++;
    u16RxBufFillCnt1 = u16RxBufFillCount1;
  }
}

static void SampleMfsStsIrqHandler1(volatile stc_mfsn_t* pstcMfs,
       void* pvHandle)
{
  volatile uint8_t u8DummyData;
  volatile uint8_t u8Reg;

  // LIN break detected?
  u8Reg = Mfs_GetStatus(pstcMfs, MFS_LIN_SSR_LBD);
  if (0 != u8Reg)
  {
    // Dummy read
    u8DummyData = (uint8_t)Mfs_ReadData(pstcMfs)
    // Clear LIN break detection
    Mfs_Lin_ClearBreakDetFlag(pstcMfs);
    // Clear possible reception errors
    Mfs_ErrorClear(pstcMfs);
    u8Dummy++;
  }
}
```

```
static en_result_t SampleMfsLinInit1(void)
{
  en_result_t      enResult;

  // At first un-initialize LIN
  (void)Mfs_Lin_DeInit(&MFS0);
  // Initialize MFS as LIN
  enResult = Mfs_Lin_Init(&MFS0, (stc_mfs_lin_config_t
*)&stcMfsLinCfg1);
  if (Ok == enResult)
  {
    // Register interrupt handler
    Mfs_SetTxIntCallBack(&MFS0, SampleMfsTxIrqHandler1);
    Mfs_SetRxIntCallBack(&MFS0, SampleMfsRxIrqHandler1);
    Mfs_SetStsIntCallBack(&MFS0, SampleMfsStsIrqHandler1);

    // Initialize variables for RX
    // (Variables for TX is initialized in SampleMfsLinWrite1())
    u16RxBufOutIndex1 = 0;
    u16RxBufInIndex1 = 0;
    u16RxBufFillCount1 = 0;
    u16RxBufFillCnt1 = 0;

    // Clear possible reception errors
    Mfs_ErrorClear(&MFS0);
    // Enable TX function
    Mfs_SetTxEnable(&MFS0, TRUE);
    // Enable RX function
    Mfs_SetRxEnable(&MFS0, TRUE);
    // Enable RX interrupt
    Mfs_SetRxIntEnable(&MFS0, TRUE);
    // Init transmission interrupt
    Mfs_InitTxIrq(&MFS0);
    // Init reception interrupt
    Mfs_InitRxIrq(&MFS0);
  }

  return (enResult);
}
```

```
static en_result_t SampleMfsLinWrite1(        uint8_t* pu8TxBuf,
                                              uint16_t u16WriteCnt)
{
  // Check for valid pointer
  if (NULL == pu8TxBuf)
  {
    return (ErrorInvalidParameter);
  }

  // Check if nothing to do
  if (0 == u16WriteCnt)
  {
    return (Ok);
  }

  // Disable transmit interrupt
  Mfs_SetTxIntEnable(&MFS0, FALSE);

  // Set tx data area
  pu8LinTxBuf1 = pu8TxBuf;
  // Set tx data fill count (to transmit by interrupt)
  u16TxBufFillCount1 = u16WriteCnt;
  // Initialize output index
  u16TxBufOutIndex1 = 0;
  // Enable transmit interrupt
  Mfs_SetTxIntEnable(&MFS0, TRUE);
  // Wait until all data has been tranferred to the MFS
  // This is fully INT driven
  while (0 != u16TxBufFillCount1);

  return (Ok);
}
```

```
static en_result_t SampleMfsLinRead1(        uint8_t* pu8RxBuf,
                                             uint16_t* pu16ReadCnt)
{
  uint16_t        u16Idx;
  uint16_t        u16Length;
  uint16_t        u16BytesToReadLeft;

  // Check for valid pointers
  if ((NULL == pu8RxBuf) || (NULL == pu16ReadCnt))
  {
      return (ErrorInvalidParameter);
  }
  // Save Read Count for later use
  u16BytesToReadLeft = *pu16ReadCnt;
  *pu16ReadCnt = 0;   // Preset to default

  u16Idx = 0;
  // Read all available bytes from ring buffer, blocking.
  while (0 < u16BytesToReadLeft)
  {
    if (0 == u16RxBufFillCount1)
    {
      return (Ok);
    }
    // Disable reception interrupt
    Mfs_SetRxIntEnable(&MFS0, FALSE);
    // Copy data to destinaton buffer and save no. of bytes been read
    // get number of bytes to read
    u16Length = MIN(u16RxBufFillCount1, u16BytesToReadLeft);
    // if there are any bytes left to read
    if (0 != u16Length)
    {
      // read bytes out of RX buffer
      for (u16Idx = *pu16ReadCnt; u16Idx < (u16Length + *pu16ReadCnt);
u16Idx++)
      {
        pu8RxBuf[u16Idx] = au8LinRxBuf1[u16RxBufOutIndex1];
        // Update out index
        u16RxBufOutIndex1++;
        if (SAMPLE_LIN_RX_BUFFSIZE <= u16RxBufOutIndex1)
        {
          u16RxBufOutIndex1 = 0;
        }
      }
      u16RxBufFillCount1 -= u16Length;        // Update fill counter
    }
    *pu16ReadCnt += u16Length;                // Provide no. of read to
the caller
    u16BytesToReadLeft -= u16Length;          // Some data processed
    // Enable reception interrupt
    Mfs_SetRxIntEnable(&MFS0, TRUE);
  }

  return (Ok);
}
```

```
SampleMfsTxIrqHandler2() is same as SampleMfsTxIrqHandler1() here (for
ch.6 (MFS6)).

SampleMfsRxIrqHandler2() is same as SampleMfsRxIrqHandler1() here (for
ch.6 (MFS6)).

SampleMfsStsIrqHandler2() is same as SampleMfsStsIrqHandler1() here
(for ch.6 (MFS6)).

SampleMfsLinInit2() is same as SampleMfsLinInit1() here (for ch.6
(MFS6)).

SampleMfsLinWrite2() is same as SampleMfsLinWrite1() here (for ch.6
(MFS6)).

SampleMfsLinRead2() is same as SampleMfsLinRead1() here (for ch.6
(MFS6)).

static void SampleMfsLinWait(volatile uint32_t u u32WaitCount)
{
  // Wait specified count
  while (0 != (u32WaitCount--));
}
```

```
static void SampleMfsLin(void)
{
  // Only for example! Do not use this in your application!
  // Temporarily disable Slave RX interrupt. Will be reenabled by Mfs_Read()
  Mfs_SetRxIntEnable(&MFS6, FALSE);
  u8LinBrk2 = FALSE;
  // LIN Master Task
  au8WriteBuffer1[0] = 0x55;  // Synch Field
  au8WriteBuffer1[1] = 'M';   // Header (no LIN meaning, just test byte)
  au8WriteBuffer1[2] = 'S';   // Data (no LIN meaning, just test byte)
  au8WriteBuffer1[3] = 'T';   // Checksum (no LIN meaning, just test byte)
  u16RxBufFillCnt2 = 0;
  // First Set LIN Break
  Mfs_Lin_SetBreak(&MFS0);
  while (FALSE == u8LinBrk2); // wait for break received by slave
  // Prepare Read LIN Slave (Enable reception interrupt)
  Mfs_SetRxIntEnable(&MFS6, TRUE);
  // Write rest of LIN Frame
  SampleMfsLinWrite1(au8WriteBuffer1, 4);
  // Wait for all data read in MFS6 (slave)
  while (u16RxBufFillCnt2 < 4);
  // Transfer LIN slave data from internal buffer to Main_ReadBuffer2
  SampleMfsLinRead2(au8ReadBuffer2, (uint16_t *)&u16RxBufFillCnt2);
  // Wait time for next frame (usually done by scheduler timer)
  SampleMfsLinWait(20000);
  // Temporarily disable Slave RX interrupt. Will be reenabled by Mfs_Read()
  Mfs_SetRxIntEnable(&MFS6, FALSE);
  u8LinBrk2 = FALSE;
  // LIN Slave Task
  au8WriteBuffer1[0] = 0x55;  // Synch Field
  au8WriteBuffer1[1] = 'S';   // Header (no LIN meaning, just test byte)
  au8WriteBuffer2[0] = 'L';   // Data (no LIN meaning, just test byte)
  au8WriteBuffer2[1] = 'V';   // Checksum (no LIN meaning, just test byte)
  u16RxBufFillCnt2 = 0;
  // First Set LIN Break
  Mfs_Lin_SetBreak(&MFS0);
  while (FALSE == u8LinBrk2); // wait for break received by slave
  // Prepare Read LIN Slave (Enable reception interrupt)
  Mfs_SetRxIntEnable(&MFS6, TRUE);
  // Write synch field and header of LIN Frame
  SampleMfsLinWrite1(au8WriteBuffer1, 2);
  // Wait for all data read in MFS6 (slave)
  while (u16RxBufFillCnt2 < 2);
  // Transfer LIN slave data from internal buffer to Main_ReadBuffer2
  SampleMfsLinRead2(au8ReadBuffer2, (uint16_t *)&u16RxBufFillCnt2);
  u16RxBufFillCnt1 = 0;
  // Prepare Read LIN Master (Enable reception interrupt)
  Mfs_SetRxIntEnable(&MFS0, TRUE);
  // Write synch field and header of LIN Frame
  SampleMfsLinWrite2(au8WriteBuffer2, 2);
  // Wait for all data read back in MFS0 (master)
  while (u16RxBufFillCnt1 < 2);
  // Transfer LIN master data from internal buffer to Main_ReadBuffer
  SampleMfsLinRead1(au8ReadBuffer1 + 2, (uint16_t *)&u16RxBufFillCnt1);
}
```

```
function
{
  en_result_t enResult;

  // Disable Analog input (P21:SIN0_0/AN17, P22:SOT0_0/AN16)
  FM4_GPIO->ADE = 0;

  // Set LIN Ch0_0 Port (SIN, SOT)
  FM4_GPIO->PFR2 = FM4_GPIO->PFR2 | 0x0006;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00000040;
  // Set LIN Ch6_0 Port (SIN, SOT)
  FM4_GPIO->PFR5 = FM4_GPIO->PFR5 | 0x0060;
  FM4_GPIO->EPFR08 = FM4_GPIO->EPFR08 | 0x00050000;

  // some code here ...

  // Initialize MFS ch.0 as LIN master mode
  enResult = SampleMfsLinInit1();
  if (Ok == enResult)
  {
    // Initialize MFS ch.6 as LIN slave mode
    enResult = SampleMfsLinInit2();
    if (Ok != enResult)
    {
      (void)Mfs_Lin_DeInit(&MFS0);
    }
  }

  if (Ok != enResult)
  {
    // some code here ...
    while(1);
  }

  // If initilazation is successful, LIN master and slave activation
sample is performed.
  SampleMfsLin();

  while (1);
}
```

7.22.3.4 UART

UART without using interrupt

This example software excerpt shows an usage of the UART driver library without using interrupt.

```
#include "mfs/mfs.h"

...
static const stc_mfs_uart_config_t stcMfsUartCfg = {
  115200,                   // Baud rate
  MfsUartNormal,  // Normal mode
  MfsParityNone,  // Non parity
  MfsOneStopBit,  // 1 stop bit
  MfsEightBits,   // 8 character bits
  FALSE,          // LSB first
  FALSE,          // NRZ
  FALSE           // Not use Hardware Flow
};

...
```

```
static en_result_t SampleMfsUartRead(   uint8_t* pu8RxBuf,
                                         uint16_t* pu16ReadCnt)
{
  uint16_t          16Idx;
  uint16_t          u16BytesToReadLeft;
  volatile uint8_t            u8Reg;

  // Check for valid pointers
  if (( NULL == pu8RxBuf) || (NULL == pu16ReadCnt))
  {
    return (ErrorInvalidParameter);
  }

  // Check for nothing to do
  if (0u == *pu16ReadCnt)
  {
    return (Ok);
  }
  // Save Read Count for later use
  u16BytesToReadLeft = *pu16ReadCnt;

  u16Idx = 0;
  // Read all available bytes from ring buffer, blocking.
  while (0 < u16BytesToReadLeft)
  {
    // Read status
    u8Reg = Mfs_GetStatus(&MFS0,
                          (MFS_UART_SSR_ERR | MFS_UART_SSR_RDRF));
    // Overrun/Framing/Parity error
    if (0 != (u8Reg & MFS_UART_SSR_ERR))
    {
      // Clear possible reception errors
      Mfs_ErrorClear(pstcUart);
    }
    // If received data is full...
    if (0 != (u8Reg & MFS_UART_SSR_RDRF))
    {
      // Store Received Data Register into buffer
      pu8RxBuf[u16Idx] = (uint8_t)Mfs_ReadData(&MFS0);
      ++u16Idx;
      u16BytesToReadLeft--;
    }
    // If received data is none...
    else
    {
      break;
    }
  }
  // Write variable for number of bytes to been read
  *pu16ReadCnt = u16Idx;

  return (Ok);
}
```

```
static en_result_t SampleMfsUartWrite(       uint8_t* pu8TxBuf,
                                             uint16_t u16WriteCnt)
{
  uint16_t         u16Idx;
  volatile uint8_t u8Reg;

  // Check for valid pointer
  if (NULL == pu8TxBuf)
  {
    return (ErrorInvalidParameter);
  }

  // Check if nothing to do
  if (0 == u16WriteCnt)
  {
    return (Ok);
  }

  u16Idx = 0;

  // Loop until all data has been sent
  while (0 != u16WriteCnt)
  {
    // If Transmit Data Register (TDR) is empty
    u8Reg = Mfs_GetStatus(&MFS0, MFS_UART_SSR_TDRE);
    if (0 != u8Reg)
    {
      // Write the contents of the buffer to Transmit Data Register
      Mfs_WriteData(&MFS0, (uint16_t)pu8TxBuf[u16Idx]);
      u16Idx++;
      u16WriteCnt--;
    }
  }

  return (Ok);
}
```

```
function
{
  uint8_t au8ReadBuf[128];
  uint16_t        u16ReadCnt;

  // Disable Analog input (P21:SIN0_0/AN17, P22:SOT0_0/AN16)
  FM4_GPIO->ADE = 0;

  // Set UART Ch0_0 Port (SIN, SOT)
  FM4_GPIO->PFR2 = FM4_GPIO->PFR2 | 0x0006;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00000040;

  // At first un-initialize UART
  (void)Mfs_Uart_DeInit(&MFS0);

  // Initialize MFS as UART
  if (Ok != Mfs_Uart_Init(&MFS0, (stc_mfs_uart_config_t
*)&stcMfsUartCfg))
  {
    // some code here ...
    while(1);
  }

  // Clear possible reception errors
  Mfs_ErrorClear(&MFS0);
  // Enable TX function
  Mfs_SetTxEnable(&MFS0, TRUE);
  // Enable RX function
  Mfs_SetRxEnable(&MFS0, TRUE);

  SampleMfsUartWrite("UART (MFS) Test\r\n", 17);

  while (1)
  {
    // Receive data from UART asynchrnously (non-blocking)
    u16ReadCnt = 128;
    if (Ok == SampleMfsUartRead(au8ReadBuf, &u16ReadCnt))
    {
      // If data is received from UART,
      if (0 < u16ReadCnt)
      {
        // Send received data to UART (Echo)
        SampleMfsUartWrite(au8ReadBuf, u16ReadCnt);
      }
    }
  }
}
```

UART with using interrupt

This example software excerpt shows an usage of the UART driver library with using interrupt.

```
#include "mfs/mfs.h"

#define SAMPLE_UART_RX_BUFFSIZE              (256)
...
static uint8_t*             pu8UartTxBuf;
static uint8_t              au8UartRxBuf[SAMPLE_UART_RX_BUFFSIZE];
static uint16_t             u16TxBufOutIndex;
static volatile uint16_t    u16TxBufFillCount;
static uint16_t             u16RxBufInIndex;
static uint16_t             u16RxBufOutIndex;
static volatile uint16_t    u16RxBufFillCount;

Configuration structure like in the here

...

static en_result_t SampleMfsTxIrqHandler(volatilestc_mfsn_t* pstcMfs,
                                         void* pvHandle )
{
  // Put data from Buffer into Transmit Data Register
  Mfs_WriteData(pstcMfs, (uint16_t)pu8UartTxBuf[u16TxBufOutIndex]);
  // Update tail
  u16TxBufOutIndex++;
  u16TxBufFillCount--;

  // If no more bytes to sent ...
  if (0 == u16TxBufFillCount)
  {
    // Disable transmission interrupt
    Mfs_SetTxIntEnable(pstcMfs, FALSE);
  }
}
```

```
static void SampleMfsRxIrqHandler(  volatile stc_mfsn_t* pstcMfs,
                                    void* pvHandle )
{
  uint16_t        u16Data;
  volatile uint8_t u8Ssr;

  // Check Overrun error
  u8Ssr = Mfs_GetStatus(pstcMfs, MFS_UART_SSR_ERR);
  if (0 != u8Ssr)
  {
    // Clear possible reception errors
    Mfs_ErrorClear(pstcMfs);
  }

  // Read data from Read Data Register
  u16Data = Mfs_ReadData(pstcMfs);

  // If there is empty space in RX buffer
  if (u16RxBufFillCount < SAMPLE_UART_RX_BUFFSIZE)
  {
    // Store read data to RX buffer
    au8UartRxBuf[u16RxBufInIndex] = (uint8_t)u16Data;
    // Update input index
    u16RxBufInIndex++;
    if (SAMPLE_UART_RX_BUFFSIZE <= u16RxBufInIndex)
    {
      u16RxBufInIndex = 0;
    }
    // Count bytes in RX buffer
    u16RxBufFillCount++;
  }
}
```

```
static en_result_t SampleMfsUartRead(        uint8_t* pu8RxBuf,
                                             uint16_t* pu16ReadCnt)
{
  uint16_t u16Idx;
  uint16_t u16Length;
  uint16_t u16BytesToReadLeft;

  // Check for valid pointers
  if ((NULL == pu8RxBuf) || (NULL == pu16ReadCnt)) {
    return (ErrorInvalidParameter);
  }

  // Save Read Count for later use
  u16BytesToReadLeft = *pu16ReadCnt;
  *pu16ReadCnt = 0;   // Preset to default

  u16Idx = 0;
  // Read all available bytes from ring buffer, blocking.
  while (0 < u16BytesToReadLeft) {
    if (0 == u16RxBufFillCount) {
      return (Ok);
    }
    // Disable reception interrupt
    Mfs_SetRxIntEnable(&MFS0, FALSE);

    // Copy data to destinaton buffer and save no. of bytes been read
    // get number of bytes to read
    u16Length = MIN(u16RxBufFillCount, u16BytesToReadLeft);

    // if there are any bytes left to read
    if (0 != u16Length) {
      // read bytes out of RX buffer
      for (u16Idx = *pu16ReadCnt; u16Idx < (u16Length + *pu16ReadCnt);
u16Idx++) {
        pu8RxBuf[u16Idx] = au8UartRxBuf[u16RxBufOutIndex];
        // Update out index
        u16RxBufOutIndex++;
        if (SAMPLE_UART_RX_BUFFSIZE <= u16RxBufOutIndex)
        {
          u16RxBufOutIndex = 0;
        }
      }
      u16RxBufFillCount -= u16Length;         // Update fill counter
    }

    *pu16ReadCnt += u16Length;                // Provide no. of read to
the caller
    u16BytesToReadLeft = u16Length; // Some data processed

    // Enable reception interrupt
    Mfs_SetRxIntEnable(&MFS0, TRUE);
  }

  return (Ok);
}
```

```
static en_result_t SampleMfsUartWrite(        uint8_t* pu8TxBuf,
                                              uint16_t u16WriteCnt )
{
  // Check for valid pointer
  if (NULL == pu8TxBuf)
  {
    return (ErrorInvalidParameter);
  }

  // Check if nothing to do
  if (0 == u16WriteCnt)
  {
    return (Ok);
  }

  // Disable transmit interrupt
  Mfs_SetTxIntEnable(&MFS0, FALSE);

  // Set tx data area
  pu8UartTxBuf = pu8TxBuf;
  // Set tx data fill count (to transmit by interrupt)
  u16TxBufFillCount = u16WriteCnt;
  // Initialize output index
  u16TxBufOutIndex = 0;
  // Enable transmit interrupt
  Mfs_SetTxIntEnable(&MFS0, TRUE);
  // Wait until all data has been tranferred to the MFS
  // This is fully INT driven
  while (0 != u16TxBufFillCount);

  return (Ok);
}
```

```
function
{
  uint8_t au8ReadBuf[128];
  uint16_t        u16ReadCnt;

  // Disable Analog input (P21:SIN0_0/AN17, P22:SOT0_0/AN16)
  FM4_GPIO->ADE = 0;
  // Set UART Ch0_0 Port (SIN, SOT)
  FM4_GPIO->PFR2 = FM4_GPIO->PFR2 | 0x0006;
  FM4_GPIO->EPFR07 = FM4_GPIO->EPFR07 | 0x00000040;
  // At first un-initialize UART
  (void)Mfs_Uart_DeInit(&MFS0);
  // Initialize MFS as UART
  if (Ok != Mfs_Uart_Init(&MFS0, (stc_mfs_uart_config_t
*)&stcMfsUartCfg))
  {
    // some code here ...
    while(1);
  }
  // Register interrupt handler
  Mfs_SetTxIntCallBack(&MFS0, SampleMfsTxIrqHandler);
  Mfs_SetRxIntCallBack(&MFS0, SampleMfsRxIrqHandler);
  // Initialize variables for RX
  // (Variables for TX is initialized in Sample_Mfs_Uart_Write())
  u16RxBufInIndex = 0;
  u16RxBufOutIndex = 0;
  u16RxBufFillCount = 0;
  // Clear possible reception errors
  Mfs_ErrorClear(&MFS0);
  // Enable TX function
  Mfs_SetTxEnable(&MFS0, TRUE);
  // Enable RX function
  Mfs_SetRxEnable(&MFS0, TRUE);
  // Enable RX interrupt
  Mfs_SetRxIntEnable(&MFS0, TRUE);
  // Init transmission interrupt
  Mfs_InitTxIrq(&MFS0);
  // Init reception interrupt
  Mfs_InitRxIrq(&MFS0);
  SampleMfsUartWrite("UART (MFS) Test\r\n", 17);
  while (1)
  {
    // Receive data from UART asynchrnously (non-blocking)
    u16ReadCnt = 128;
    if (Ok == SampleMfsUartRead(au8ReadBuf, &u16ReadCnt))
    {
      // If data is received from UART,
      if (0 < u16ReadCnt)
      {
        // Send received data to UART (Echo)
        SampleMfsUartWrite(au8ReadBuf, u16ReadCnt);
      }
    }
  }
}
```

7.23 (PPG) Programmable Pulse Generator

Ppg_Init() must be used for configuration of a PPG couple channel with a structure of the type stc_ppg_config_t. Three ways can trigger the PPG start:

- Triggered by software
- Triggered by up counter
- Triggered by GATE signal from MFT

A PPG interrupt can be enabled by the function Ppg_EnableInt().This function can set callback function for each channel too.

With Ppg_SetLevelWidth() the PPG low/high level width is set to the value given in the parameter Ppg_SetLevelWidth#u8LowWidth and Ppg_SetLevelWidth#u8LowWidth. Ppg_GetUpCntxStatus() can get the operation status of up counter (x=0,1,2).

If use software to start PPG, calling Ppg_StartSoftwareTrig() will start PPG.

If use up counter to start PPG, initialize up counter with Ppg_ConfigUpCntx() with a structure of the type stc_ppg_upcntx_config_t, start the up counter with Ppg_StartUpCnt1() and if count value matchs with compare value, the according PPG channel will start.

If use GATE signal to trigger PPG, set the valid level with Ppg_SelGateLevel(), and if GATE signal from MFT becomes valid level, PPG will start.

With interrupt mode, when the interrupt occurs, the interrupt flag will be cleared and run into user interrupt callback function.

With polling mode, user can use Ppg_GetIntFlag() to check if the interrupt occurs, and clear the interrupt flag by Ppg_ClrIntFlag().

When stopping the PPG, if PPG is triggered by software, use Ppg_StopSoftwareTrig() to stop PPG output; If PPG is triggered by up counter, use Ppg_DisableTimerGen0StartTrig to stop PPG output, if PPG is triggered by GATE, set the GATE signal to invalid level in the MFT module.

IGBT mode is also supported by PPG. Ppg_InitIgbt() must be used for configuration of IGBT mode with a structure of the type stc_ppg_igbt_config_t.

Ppg_EnableIgbtMode() is used to enable IGBT mode and Ppg_DisableIgbtMode() is used to disable IGBT mode.

Type Definition	-
Configuration Types	stc_ppg_config_t stc_ppg_upcnt0_config_t stc_timer0_gen_ch_t stc_ppg_upcnt1_config_t stc_timer1_gen_ch_t stc_ppg_upcnt2_config_t stc_timer2_gen_ch_t stc_ppg_igbt_config_t
Address Operator	-

7.23.1 Configuration Structure

A PPG configure instance uses the following configuration structure of the type of `stc_ppg_config_t`:

Type	Field	Possible Values	Description
en_ppg_opt_mode_t	enMode	Ppg8Bit8Bit Ppg8Bit8Pres Ppg16Bit Ppg16Bit16Pres	PPG mode configuration: Even channel: 8bit PPG, odd channel: 8bit PPG. Even channel: 8bit PPG, odd channel: 8bit prescaler 16bit PPG 16bit PPG + 16 prescaler
en_ppg_clock_t	enEvenClock	PpgPclkDiv1 PpgPclkDiv4 PpgPclkDiv16 PpgPclkDiv64	Clock prescaler of even channel: PPG count clock prescaler: 1 PPG count clock prescaler: 1/4 PPG count clock prescaler: 1/16 PPG count clock prescaler: 1/64
en_ppg_clock_t	enOddClock	Same above	Clock prescaler of odd channel.
en_ppg_level_t	enEvenLevel	PpgNormalLevel PpgReverseLevel	Output level of even channel: Inital level: Low Inital level: High
en_ppg_level_t	enOddLevel	Same above	Output level of odd channel:
en_ppg_trig_t	enTrig	PpgSoftwareTrig PpgMftGateTrig PpgTimingGenTrig	PPG trigger mode configuration: Software trigger GATE signal from MFT trigger Timing Generator trigger

An up counter0 configuration instance uses the following configuration structure of the type of `stc_ppg_upcnt0_config_t`:

Type	Field	Possible Values	Description
en_ppg_upcnt_clk_t	enClk	PpgUpCntPclkDiv2 PpgUpCntPclkDiv8 PpgUpCntPclkDiv32 PpgUpCntPclkDiv64	Up counter clock prescaler: prescaler: 1/2 prescaler: 1/8 prescaler: 1/32 prescaler: 1/64
uint8_t	u8CmpValue0	-	Up counter compare value for Ch0
uint8_t	u8CmpValue2	-	Up counter compare value for Ch2
uint8_t	u8CmpValue4	-	Up counter compare value for Ch4
uint8_t	u8CmpValue6	-	Up counter compare value for Ch6

An up counter0 stop channels selection instance uses the following configuration structure of the type of `stc_timer0_gen_ch_t`:

Type	Field	Possible Values	Description
boolean_t	bPpgCh0	TRUE FALSE	Select Ch.0 Not selected
boolean_t	bPpgCh2	TRUE FALSE	Select Ch.2 Not selected
boolean_t	bPpgCh4	TRUE FALSE	Select Ch.4 Not selected
boolean_t	bPpgCh6	TRUE FALSE	Select Ch.6 Not selected

An up counter1 onfiguration instance uses the following configuration structure of the type of `stc_ppg_upcnt1_config_t`:

Type	Field	Possible Values	Description
en_ppg_upcnt_clk_t	enClk	PpgUpCntPclkDiv2 PpgUpCntPclkDiv8 PpgUpCntPclkDiv32 PpgUpCntPclkDiv64	Up counter clock prescaler: prescaler: 1/2 prescaler: 1/8 prescaler: 1/32 prescaler: 1/64
uint8_t	u8CmpValue8	-	Up counter compare value for Ch8
uint8_t	u8CmpValue10	-	Up counter compare value for Ch10
uint8_t	u8CmpValue12	-	Up counter compare value for Ch12
uint8_t	u8CmpValue14	-	Up counter compare value for Ch14

An up counter1 stop channels selection instance uses the following configuration structure of the type of `stc_timer1_gen_ch_t`:

Type	Field	Possible Values	Description
boolean_t	bPpgCh8	TRUE FALSE	Select Ch.8 Not selected
boolean_t	bPpgCh10	TRUE FALSE	Select Ch.10 Not selected
boolean_t	bPpgCh12	TRUE FALSE	Select Ch.12 Not selected
boolean_t	bPpgCh14	TRUE FALSE	Select Ch.14 Not selected

An up counter2 onfiguration instance uses the following configuration structure of the type of `stc_ppg_upcnt2_config_t`:

Type	Field	Possible Values	Description

`en_ppg_upcnt_clk_t`	`enClk`	`PpgUpCntPclkDiv2`
		`PpgUpCntPclkDiv8`
		`PpgUpCntPclkDiv32`
		`PpgUpCntPclkDiv64`
			Up counter clock prescaler:
			prescaler: 1/2
			prescaler: 1/8
			prescaler: 1/32
			prescaler: 1/64

`uint8_t`	`u8CmpValue16`	-	Up counter compare value for Ch16
`uint8_t`	`u8CmpValue18`	-	Up counter compare value for Ch18
`uint8_t`	`u8CmpValue20`	-	Up counter compare value for Ch20
`uint8_t`	`u8CmpValue22`	-	Up counter compare value for Ch22

An up counter2 stop channels selection instance uses the following configuration structure of the type of `stc_timer2_gen_ch_t`:

Type	Field	Possible Values	Description
`boolean_t`	`bPpgCh16`	`TRUE` `FALSE`	Select Ch.16 Not selected
`boolean_t`	`bPpgCh18`	`TRUE` `FALSE`	Select Ch.18 Not selected
`boolean_t`	`bPpgCh20`	`TRUE` `FALSE`	Select Ch.20 Not selected
`boolean_t`	`bPpgCh22`	`TRUE` `FALSE`	Select Ch.22 Not selected

A PPG IGBT configuration instance uses the following configuration structure of the type of `stc_ppg_igbt_config_t`:

Type	Field	Possible Values	Description
`en_igbt_prohibition_mode_t`	`enMode`	`IgbtNormalMode` `IgbtStopProhibitionMode`	prohibition mode: Normal mode Stop prohibition mode in output active

`en_igbt_filter_width_t`	`enWidth`	`IgbtNoFilter`
		`IgbtFilter4Pclk`
		`IgbtFilter8Pclk`
		`IgbtFilter16Pclk`
		`IgbtFilter32Pclk`
			noise filter width:
			No noise filter
			noise filter width: 4PCLK
			noise filter width: 8PCLK
			noise filter width: 16PCLK
			noise filter width: 32PCLK

`en_igbt_level_t`	`enTrigInputLevel`	`IgbtLevelNormal` `IgbtLevelInvert`	Trigger input level: Normal Invert
`en_igbt_level_t`	`enIgbt0OutputLevel`	`Same above`	IGBT0 output level (PPG0):
`en_igbt_level_t`	`enIgbt1OutputLevel`	`Same above`	IGBT1 output level (PPG4):

7.23.2 PPG API

7.23.2.1 Ppg_Init ()

This function initializes PPG.

Prototype	
`en_result_t Ppg_Init( uint8_t u8CoupleCh, stc_ppg_config_t *pstcConfig);`	
Parameter Name	**Description**
`[in] u8CoupleCh`	A couple PPG channels.
`[in] pstcConfig`	Pointer to PPG configuration structure.
Return Values	**Description**
`Ok`	Configure the PPG successfully.
`ErrorInvalidParameter`	`u8CoupleCh` param invalid `pstcConfig` param invalid

7.23.2.2 Ppg_StartSoftwareTrig ()

This function starts PPG by software trigger.

Prototype	
`en_result_t Ppg_StartSoftwareTrig(uint8_t u8Ch);`	
Parameter Name	**Description**
`[in] u8Ch`	PPG channel number.
Return Values	**Description**
`Ok`	Start PPG by software done.
`ErrorInvalidParameter`	`u8Ch > PPG_CH23`

7.23.2.3 Ppg_StopSoftwareTrig ()

This function stops PPG by software trigger.

Prototype	
`en_result_t Ppg_StopSoftwareTrig(uint8_t u8Ch);`	
Parameter Name	**Description**
`[in] u8Ch`	PPG channel number.
Return Values	**Description**
`Ok`	Stop PPG by software done.
`ErrorInvalidParameter`	`u8Ch > PPG_CH23`

7.23.2.4 *Pg_SelGateLevel ()*

This function sets the valid level of GATE signal.

Prototype	
`en_result_t Ppg_SelGateLevel(uint8_t u8EvenCh, en_ppg_gate_level_t enLevel);`	
Parameter Name	**Description**
`[in] u8EvenCh`	An even channel of PPG.
`[in] enLevel`	GATE level
Return Values	**Description**
`Ok`	Set the valid level of GATE signal successfully.
`ErrorInvalidParameter`	`u8EvenCh` param invalid `enLevel` param invalid

7.23.2.5 *Ppg_ConfigUpCnt0 ()*

This function configures the up counter of Timing Generator 0.

Prototype	
`en_result_t  Ppg_ConfigUpCnt0(stc_ppg_upcnt0_config_t* pstcConfig);`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to up counter 0 configuration structure.
Return Values	**Description**
`Ok`	Configure the up counter successfully.
`ErrorInvalidParameter`	`pstcConfig` param invalid

7.23.2.6 *Ppg_StartUpCnt0 ()*

This function starts the up counter of Timing Generator 0.

Prototype	
`void Ppg_StartUpCnt0(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	

7.23.2.7 *Ppg_GetUpCnt0Status ()*

This function gets the work status of up counter of Timing Generator 0.

Prototype	
`en_stat_flag_t Ppg_GetUpCnt0Status(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
`PdlSet`	Up counter is counting.
`PdlClr`	Up counter stops.

7.23.2.8 *Ppg_DisableTimerGen0StartTrig ()*

This function disables start trigger of Timing Generator 0.

Prototype	
`en_result_t Ppg_DisableTimerGen0StartTrig(stc_timer0_gen_ch_t* pstcTimer0GenCh);`	
Parameter Name	**Description**
`[in] pstcTimer0GenCh`	Pointer to the structure of selected channels.
Return Values	**Description**
`Ok`	Disable start trigger.
`ErrorInvalidParameter`	`pstcTimer0GenCh` param invalid.

7.23.2.9 *Ppg_ConfigUpCnt1 ()*

This function configures the up counter of Timing Generator 1.

Prototype	
`en_result_t Ppg_ConfigUpCnt1(stc_ppg_upcnt1_config_t* pstcConfig);`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to up counter 1 configuration structure.
Return Values	**Description**
`Ok`	Configure the up counter successfully.
`ErrorInvalidParameter`	`pstcConfig` param invalid

7.23.2.10 Ppg_StartUpCnt1 ()

This function starts the up counter of Timing Generator 1.

Prototype	
`void Ppg_StartUpCnt1(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	

7.23.2.11 Ppg_GetUpCnt1Status ()

This function gets the work status of up counter of Timing Generator 1.

Prototype	
`en_stat_flag_t  Ppg_GetUpCnt1Status(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
PdlSet	Up counter is counting.
PdlClr	Up counter stops.

7.23.2.12 Ppg_DisableTimerGen1StartTrig ()

This function disables start trigger of Timing Generator 1.

Prototype	
`en_result_t  Ppg_DisableTimerGen1StartTrig(stc_timer1_gen_ch_t* pstcTimer1GenCh);`	
Parameter Name	**Description**
[in] pstcTimer1GenCh	Pointer to the structure of selected channels.
Return Values	**Description**
Ok	Disable start trigger.
ErrorInvalidParameter	pstcTimer0GenCh param invalid.

7.23.2.13 Ppg_ConfigUpCnt2 ()

This function configures the up counter of Timing Generator 2.

Prototype	
`en_result_t Ppg_ConfigUpCnt2(stc_ppg_upcnt2_config_t* pstcConfig);`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to up counter 2 configuration structure.
Return Values	**Description**
`Ok`	Configure the up counter successfully.
`ErrorInvalidParameter`	`pstcConfig` param invalid

7.23.2.14 Ppg_StartUpCnt2 ()

This function starts the up counter of Timing Generator 2.

Prototype	
`void Ppg_StartUpCnt2(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	

7.23.2.15 Ppg_GetUpCnt2Status ()

This function gets the work status of up counter of Timing Generator 2.

Prototype	
`en_stat_flag_t Ppg_GetUpCnt2Status(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
`PdlSet`	Up counter is counting.
`PdlClr`	Up counter stops.

7.23.2.16 Ppg_DisableTimerGen2StartTrig ()

This function disables start trigger of Timing Generator 2.

Prototype	
en_result_t Ppg_DisableTimerGen2StartTrig(stc_timer2_gen_ch_t* pstcTimer2GenCh);	
Parameter Name	**Description**
[in] pstcTimer2GenCh	Pointer to the structure of selected channels.
Return Values	**Description**
Ok	Disable start trigger.
ErrorInvalidParameter	pstcTimer0GenCh param invalid.

7.23.2.17 Ppg_EnableInt ()

This function enables PPG interrupt.

Prototype	
en_result_t Ppg_EnableInt(uint8_t u8Ch, en_ppg_int_mode_t enIntMode, func_ptr_t pfnCallback);	
Parameter Name	**Description**
[in] u8Ch	PPG channnel.
[in] enIntMode	Interrupt mode.
[in] pfnCallback	Pointer to interrupt callback function.
Return Values	**Description**
Ok	Enable PPG interrupt successfully.
ErrorInvalidParameter	u8Ch param invalid enIntMode > PpgHighAndLowUnderflow pfnCallback == NULL

7.23.2.18 Ppg_DisableInt ()

This function disables PPG interrupt.

Prototype	
en_result_t Ppg_DisableInt(uint8_t u8Ch);	
Parameter Name	**Description**
[in] u8Ch	PPG channnel.
Return Values	**Description**
Ok	Disable PPG interrupt successfully.
ErrorInvalidParameter	u8Ch param invalid

7.23.2.19 *Ppg_GetIntFlag ()*

This function gets the interrupt flag of PPG.

Prototype	
`en_int_flag_t Ppg_GetIntFlag(uint8_t u8Ch);`	
Parameter Name	**Description**
`[in] u8Ch`	PPG channnel.
Return Values	**Description**
`PdlSet`	Interrupt flag set.
`PdlClr`	Interrupt flag clear `u8Ch param invalid`

7.23.2.20 *Ppg_ClrIntFlag ()*

This function clears the interrupt flag of PPG.

Prototype	
`en_result_t Ppg_ClrIntFlag(uint8_t u8Ch);`	
Parameter Name	**Description**
`[in] u8Ch`	PPG channnel.
Return Values	**Description**
`Ok`	Clear PPG interrupt flag successfully.
`ErrorInvalidParameter`	`u8Ch param invalid`

7.23.2.21 *Ppg_SetLevelWidth ()*

This function sets the pulse width of PPG.

Prototype	
`en_result_t Ppg_SetLevelWidth(uint8_t u8Ch, uint8_t u8LowWidth, uint8_t u8HighWidth);`	
Parameter Name	**Description**
`[in] u8Ch`	PPG channnel.
`[in] u8LowWidth`	Low level width of PPG.
`[in] u8HighWidth`	High level width of PPG.
Return Values	**Description**
`Ok`	Set the pulse width of PPG successfully.
`ErrorInvalidParameter`	`u8Ch > PPG_CH23`

7.23.2.22 Ppg_InitIgbt ()

This function initializes IGBT mode.

Prototype	
`en_result_t  Ppg_InitIgbt(stc_ppg_igbt_config_t* pstcPpgIgbt);`	
Parameter Name	Description
`[in] pstcPpgIgbt`	Pointer to IBGT configuration structure.
Return Values	Description
`Ok`	Initialize IGBT mode successfully.
`ErrorInvalidParameter`	`pstcPpgIgbt param invalid`

7.23.2.23 Ppg_EnableIgbtMode ()

This function enables IGBT mode.

Prototype	
`void Ppg_EnableIgbtMode(void);`	
Parameter Name	Description
-	Only PPG0 and PP4 supports IGBT mode.
Return Values	Description
-	

7.23.2.24 Ppg_DisableIgbtMode ()

This function disables IGBT mode.

Prototype	
`void Ppg_DisableIgbtMode(void);`	
Parameter Name	Description
-	PPG0 and PPG4 outputs normal PPG wave.
Return Values	Description
-	

7.23.2.25 Ppg_IrqHandler ()

This function is PPG interrupt service routine.

Prototype	
`void Ppg_IrqHandler(en_ppg_irq_ch_t u8Ch);`	
Parameter Name	Description
`[in] u8Ch`	Irq channel of PPG.
Return Values	Description
-	

7.24 (QPRC) Quad Position and Revolution Counter

Type Definition	-
Configuration Types	stc_qprc_int_sel_t stc_qprc_int_cb_t stc_qprc_filter_t stc_qprc_config_t stc_qprc_intern_data_t stc_qprc_instance_data_t
Address Operator	-

Qprc_Init() must be used for configuration of a QPRC channel with a structure of the type stc_qprc_config_t.

A QPRC interrupt can be enabled by the function Qprc_EnableInt().This function can set callback function for each channel too.

With Qprc_SetPcCompareValue() the PC compare value is set to the value given in the parameter Qprc_SetPcCompareValue#u16PcValue. And PC compare value is read by Qprc_GetPcCompareValue().

With Qprc_SetPcRcCompareValue() the PC and RC compare value is set to the value given in the parameter Qprc_SetPcRcCompareValue#u16PcRcValue. And PC and RC compare value is read by Qprc_GetPcRcCompareValue(). Whether PC and RC compare value compares to PC count and RC count depends on the setting in paramter of the Qprc_Init().

The initial PC count value can be set by Qprc_SetPcCount(). And current PC count can be read by Qprc_GetPcCount().

The initial RC count value can be set by Qprc_SetRcCount(). And current RC count can be read by Qprc_GetRcCount()

The maximum PC count value can be set by Qprc_SetPcMaxValue(). And the maximum PC count value can be read by Qprc_GetPcMaxValue(). If PC count exceeds this value, a PC overflow interrupt flag will be set.

After above setting, Qprc_ConfigPcMode() can start PC with following mode:

- Up-down count mode(PC mode 1)
- Phase different count mode (PC mode 2)
- Directional count mode (PC mode 3)

Qprc_ConfigRcMode() can start RC with following mode:

- ZIN trigger mode(RC mode 1)
- PC underflow or overflow detection trigger mode (RC mode 2)
- ZIN or PC underflow or overflow detection trigger mode(RC mode 3)

With interrupt mode, when the interrupt occurs, the interrupt flag will be cleared and run into user interrupt callback function.

With polling mode, user can use Mft_Qprc_GetIntFlag() to check if the interrupt occurs, and clear the interrupt flag by Mft_Qprc_ClrIntFlag().

Qprc_GetPcOfUfDir() can get the PC direction after a PC overflow or underflow occurs.

Qprc_GetPcDir() can get the current PC direction when PC is counting.

Qprc_StopPcCount() can stop PC when PC is counting. And Qprc_RestartPcCount() will restart PC when PC is in stop status.

When stopping the QPRC, disable PC by using Qprc_ConfigPcMode() to set PC to PC mode0 and disable RC by using Qprc_ConfigRcMode() to set RC to RC mode0. Use Mft_QPRC_DisableInt() to disable QPRC interrupt.

7.24.1 Configuration Structure

A QPRC interrupt selection instance uses the following configuration structure of the type of `stc_qprc_int_sel_t`:

Type	Field	Possible Values	Description
boolean_t	bQprcPcOfUfZeroInt	TRUE FALSE	Overflow, undowflow, zero match interrupt Selected Not selected
boolean_t	bQprcPcMatchInt	TRUE FALSE	PC match interrupt of position counter Selected Not selected
boolean_t	bQprcPcRcMatchInt	TRUE FALSE	PC and RC match interrupt Selected Not selected
boolean_t	bQprcPcMatchRcMatchInt	TRUE FALSE	PC match and RC match interrupt Selected Not selected
boolean_t	bQprcPcCountInvertInt	TRUE FALSE	PC invert interrupt Selected Not selected
boolean_t	bQprcRcOutrangeInt	TRUE FALSE	RC outrange interrupt Selected Not selected

A QPRC interrupt callback function instance uses the following configuration structure of the type of `stc_qprc_int_cb_t`:

Type	Field	Possible Values	Description
func_ptr_arg1_t	pfnPcOfUfZeroIntCallback	-	Overflow, undowflow, zero match interrupt callback function of position counter
func_ptr_t	pfnPcMatchIntCallback	-	PC match interrupt callback function of position counter
func_ptr_t	pfnPcRcMatchIntCallback	-	PC and RC match interrupt callback function
func_ptr_t	pfnPcMatchRcMatchIntCallback	-	PC match and RC match interrupt callback function
func_ptr_t	pfnPcCountInvertIntCallback	-	PC invert interrupt callback function
func_ptr_t	pfnRcOutrangeIntCallback	-	RC outrange interrupt callback function

A QPRC filter configuration instance uses the following configuration structure of the type of `stc_qprc_filter_t`:

Type	Field	Possible Values	Description
boolean_t	bInputMask	 TRUE FALSE	Input mask setting: Set mask Not set mask
boolean_t	bInputInvert	 TRUE FALSE	Input invert setting: Set invert Not set invert
en_qprc_filter_width_t	enWidth	 QprcNoFilter QprcFilterWidth4Pclk QprcFilterWidth8Pclk QprcFilterWidth16Pclk QprcFilterWidth32Pclk QprcFilterWidth64Pclk QprcFilterWidth128Pclk QprcFilterWidth256Pclk	QPRC filter width setting: No filter 4 PCLK 8 PCLK 16 PCLK 32 PCLK 64 PCLK 128 PCLK 256 PCLK

A QPRC configuration instance uses the following configuration structure of the type of `stc_qprc_config_t`:

Type	Field	Possible Values	Description
boolean_t	bSwapAinBin	TRUE FALSE	Swap AIN and BIN. Swap Not swap
en_qprc_compmode_t	enComapreMode	QprcComapreWithPosition QprcComapreWithRevolution	compare object of QPRCR register Compares the value of the QPRC Position and Revolution Counter Compare Register (QPRCR) with that of the position counter. Compares the value of the QPRC Position and Revolution Counter Compare Register (QPRCR) with that of the revolution counter.
en_qprc_zinedge_t	enZinEdge	QprcZinDisable QprcZinFallingEdge QprcZinRisingEdge QprcZinBothEdges QprcZinLowLevel QprcZinHighLevel	ZIN pin mode. Disables edge and level detection ZIN active at falling edge ZIN active at rising edge ZIN active at falling or rising edge ZIN active at low level detected ZIN active at high level detected
en_qprc_binedge_t	enBinEdge	QprcBinDisable QprcBinFallingEdge QprcBinRisingEdge QprcBinBothEdges	Detection mode of the BIN pin Disables edge detection BIN active at falling edge BIN active at rising edge BIN active at falling or rising edge
en_qprc_ainedge_t	enAinEdge	QprcAinDisable QprcAinFallingEdge QprcAinRisingEdge QprcAinBothEdges	Detection mode of the AIN pin Disables edge detection AIN active at falling edge AIN active at rising edge AIN active at falling or rising edge
en_qprc_pcresetmask_t	enPcResetMask	QprcResetMaskDisable QprcResetMask2Times QprcResetMask4Times QprcResetMask8Times	reset mask times of position counter no reset mask 2 times 4 times 8 times
boolean_t	b8KValue	TRUE FALSE	Outrange mode from 0 to 0x7FFF Outrange mode from 0 to 0xFFFF
stc_qprc_filter_t	stcAinFilter	-	AIN noise filter configuration See "stc_qprc_filter_t" for details.
stc_qprc_filter_t	stcBinFilter	-	BIN noise filter configuration

Type	Field	Possible Values	Description
			See "stc_qprc_filter_t" for details.
stc_qprc_filter_t	stcCinFilter	-	CIN noise filter configuration See "stc_qprc_filter_t" for details.

A QPRC internal data instance uses the following configuration structure of the type of `stc_qprc_intern_data_t`:

Type	Field	Possible Values	Description
func_ptr_arg1_t	pfnPcUfOfZeroCallback	-	Pointer to overflow, undowflow, zero match interrupt callback function of position counter
func_ptr_t	enComapreMode	-	Pointer to PC match interrupt callback function of position counter
func_ptr_t	pfnPcRcMatchCallback	-	Pointer to PC and RC match interrupt callback function
func_ptr_t	pfnPcMatchRcMatchCallback	-	Pointer to PC match and RC match interrupt
func_ptr_t	pfnPcCountInvertCallback	-	Pointer to PC invert interrupt
func_ptr_t	pfnRcOutrangeCallback	-	Pointer to RC outrange interrupt callback function

A QPRC instance data instance uses the following configuration structure of the type of `stc_qprc_instance_data_t`:

Type	Field	Possible Values	Description
volatile stc_qprcn_t*	pstcInstance	-	Pointer to registers of an instance
volatile stc_qprc_nfn_t*	pstcInstanceNf	-	Pointer to registers of a QPRC-NF instance
stc_qprc_intern_data_t	stcInternData	-	module internal data of instance

7.24.2 QPRC API

7.24.2.1 *Qprc_Init ()*

This function initializes QPRC.

Prototype	
`en_result_t Qprc_Init( volatile stc_qprcn_t* pstcQprc,stc_qprc_config_t*  pstcConfig );`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
`[in] pstcConfig`	Pointer to a QPRC module configuration.
Return Values	**Description**
`Ok`	QPRC has been inited successfully.

`ErrorInvalidParameter`	`pstcQprc == NULL`
	`pstcConfig == NULL`
	`pstcConfig` param invalid.
	Other invalid configuration.

7.24.2.2 *Qprc_StopPcCount ()*

This function enables Position Counter.

Prototype	
`en_result_t Qprc_StopPcCount(volatile stc_qprcn_t *pstcQprc);`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
Return Values	**Description**
`Ok`	Enable Position Counter counting successfully.
`ErrorInvalidParameter`	`pstcQprc == NULL`

7.24.2.3 *Qprc_RestartPcCount ()*

This function disables Position Counter.

Prototype	
`en_result_t Qprc_RestartPcCount(volatile stc_qprcn_t *pstcQprc);`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
Return Values	**Description**
`Ok`	Stops Position Counter counting successfully.
`ErrorInvalidParameter`	`pstcQprc == NULL`

7.24.2.4 Qprc_SetPcCount ()

This function sets count value of Position counter.

Prototype	
`en_result_t Qprc_SetPcCount ( volatile stc_qprcn_t *pstcQprc,uint16_t u16PcValue );`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
`[in] u16PcValue`	Count value.
Return Values	**Description**
`Ok`	Count value has been setup.
`ErrorInvalidParameter`	`pstcQprc == NULL`

7.24.2.5 Qprc_GetPcCount ()

This function gets count value of Position counter.

Prototype	
`uint16_t Qprc_GetPcCount ( volatile stc_qprcn_t *pstcQprc );`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
Return Values	**Description**
`value`	Position counter value.
`ErrorInvalidParameter`	`pstcQprc == NULL`

7.24.2.6 Qprc_SetRcCount ()

This function sets count value of Revolution Counter.

Prototype	
`en_result_t Qprc_SetRcCount ( volatile stc_qprcn_t *pstcQprc,uint16_t u16RcValue );`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
`[in] u16PcValue`	Count value.
Return Values	**Description**
`Ok`	Count value has been setup.
`ErrorInvalidParameter`	`pstcQprc == NULL`

7.24.2.7 *Qprc_GetRcCount ()*

This function gets count value of Revolution Counter.

Prototype	
`uint16_t Qprc_GetRcCount ( volatile stc_qprcn_t *pstcQprc );`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
Return Values	**Description**
`Value`	Count value.
`ErrorInvalidParameter`	`pstcQprc == NULL`

7.24.2.8 *Qprc_SetPcMaxValue ()*

This function set maximum count value of Position Counter.

Prototype	
`en_result_t Qprc_SetPcMaxValue( volatile stc_qprcn_t *pstcQprc,uint16_t u16PcMaxValue );`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
`[in] u16PcMaxValue`	Maximum count value.
Return Values	**Description**
`Ok`	Value has been setup.
`ErrorInvalidParameter`	`pstcQprc == NULL`

7.24.2.9 *Qprc_GetPcMaxValue ()*

This function gets maximum count value of Position Counter.

Prototype	
`uint16_t Qprc_GetPcMaxValue(volatile stc_qprcn_t *pstcQprc);`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
Return Values	**Description**
`value`	PcMaxValue.
`ErrorInvalidParameter`	`pstcQprc == NULL`

7.24.2.10 Qprc_SetPcCompareValue ()

This function sets compare value of Position Counter.

Prototype	
`en_result_t Qprc_SetPcCompareValue( volatile stc_qprcn_t *pstcQprc,uint16_t u16PcValue );`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
`[in] u16PcValue`	Compare value.
Return Values	**Description**
`Ok`	Compare value has been setup.
`ErrorInvalidParameter`	`pstcQprc == NULL`

7.24.2.11 Qprc_GetPcCompareValue ()

This function gets compare value of Position Counter.

Prototype	
`uint16_t Qprc_GetPcCompareValue( volatile stc_qprcn_t *pstcQprc);`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
Return Values	**Description**
`value`	Compare value.
`ErrorInvalidParameter`	`pstcQprc == NULL`

7.24.2.12 Qprc_SetPcRcCompareValue ()

This function sets compare value of Position and Revolution Counter.

Prototype	
`en_result_t Qprc_SetPcRcCompareValue( volatile stc_qprcn_t *pstcQprc,uint16_t u16PcRcValue );`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
`[in] u16PcRcValue`	Compare value.
Return Values	**Description**
`Ok`	Compare value has been setup.
`ErrorInvalidParameter`	`pstcQprc == NULL`

7.24.2.13 Qprc_GetPcRcCompareValue ()

This function gets compare value of Position and Revolution Counter.

Prototype	
`uint16_t Qprc_GetPcRcCompareValue(volatile stc_qprcn_t *pstcQprc);`	
Parameter Name	**Description**
[in] pstcQprc	Pointer to a QPRC instance.
Return Values	**Description**
value	Compare value.
ErrorInvalidParameter	pstcQprc == NULL

7.24.2.14 Qprc_ConfigPcMode ()

This function sets Position Counter mode.

Prototype	
`en_result_t Qprc_ConfigPcMode( volatile stc_qprcn_t *pstcQprc,en_qprc_pcmode_t enMode );`	
Parameter Name	**Description**
[in] pstcQprc	Pointer to a QPRC instance.
[in] enMode	Position Counter mode.
Return Values	**Description**
Ok	Mode has been setup.
ErrorInvalidParameter	pstcQprc == NULL enMode > QprcPcMode3

7.24.2.15 Qprc_ConfigRcMode ()

This function sets Revolution Counter mode.

Prototype	
`en_result_t Qprc_ConfigRcMode( volatile stc_qprcn_t *pstcQprc,en_qprc_rcmode_t enMode );`	
Parameter Name	**Description**
[in] pstcQprc	Pointer to a QPRC instance.
[in] enMode	Revolution Counter mode.
Return Values	**Description**
Ok	Mode has been setup.
ErrorInvalidParameter	pstcQprc == NULL enMode > QprcPcMode3

7.24.2.16 Qprc_EnableInt ()

This function enables PC match interrupt.

Prototype	
`en_result_t Qprc_EnableInt( volatile stc_qprcn_t* pstcQprc, stc_qprc_int_sel_t* pstcIntSel, stc_qprc_int_cb_t* pstcIntCallback );`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
`[in] pstcIntSel`	Pointer to interrupt selection type.
`[in] pfnOfufZeroCallback`	Pointer to interrupt callback functions.
Return Values	**Description**
`Ok`	PC match interrupt enable bit has been setup.
`ErrorInvalidParameter`	`pstcQprc == NULL` `pstcIntSel` parameter invalid. Other invalid configuration.

7.24.2.17 Qprc_DisableInt ()

This function disables PC match interrupt.

Prototype	
`en_result_t Qprc_DisableInt( volatile stc_qprcn_t* pstcQprc, stc_qprc_int_sel_t* pstcIntSel );`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
`[in] pstcIntSel`	Pointer to interrupt selection.
Return Values	**Description**
`Ok`	PC match interrupt disable bit has been set and clear callback function.
`ErrorInvalidParameter`	`pstcQprc == NULL` `pstcIntSel` param invalid. Other invalid configuration.

7.24.2.18 Qprc_GetIntFlag ()

This function gets interrupt flag of QPRC.

Prototype	
`en_int_flag_t Qprc_GetIntFlag( volatile stc_qprcn_t *pstcQprc,en_qprc_int_t enIntType);`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
`[in] enIntType`	Interrupt type.
Return Values	**Description**
`PdlClr`	Don't occur interrupt which corresponds with enIntType.
`PdlSet`	Occur interrupt which corresponds with enIntType.

7.24.2.19 Qprc_ClrIntFlag ()

This function clears interrupt flag of QPRC.

Prototype	
`en_result_t Qprc_ClrIntFlag( volatile stc_qprcn_t *pstcQprc,en_qprc_int_t enIntType);`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
`[in] enIntType`	Interrupt type.
Return Values	**Description**
`Ok`	Cleared interrupt flag which corresponds with enIntType.
`ErrorInvalidParameter`	`pstcQprc == NULL` `pstcQprc` invalid. Other invalid configuration.

7.24.2.20 Qprc_GetPcOfUfDir ()

This function gets last position counter flow direction.

Prototype	
`en_stat_flag_t Qprc_GetPcOfUfDir( volatile stc_qprcn_t *pstcQprc );`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
Return Values	**Description**
`PdlClr`	The position counter was incremented.
`PdlSet`	The position counter was decremented.

7.24.2.21 Qprc_GetPcDir ()

This function gets last position counter direction.

Prototype	
`en_stat_flag_t Qprc_GetPcDir( volatile stc_qprcn_t *pstcQprc );`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
Return Values	**Description**
`PdlClr`	The position counter was incremented. `pstcQprc == NULL`
`PdlSet`	The position counter was decremented.

7.24.2.22 Qprc_IrqHandler ()

This function is QPRC instance interrupt service routine.

Prototype	
`void Qprc_IrqHandler ( volatile stc_qprcn_t *pstcQprc,stc_qprc_intern_data_t *pstcQprcInternData );`	
Parameter Name	**Description**
`[in] pstcQprc`	Pointer to a QPRC instance.
`[in] pstcQprcInternData`	Pointer to internal data.
Return Values	**Description**
–	

7.25 (RESET) Reset Cause

Type Definition	–
Configuration Type	–
Address Operator	–

This module provides access to the Reset Cause register and a global reset cause variable `stc_reset_result_t stcStoredResetCause`. This driver does not need any configuration.

7.25.1 RESET API

The following API functions are used for handling the Low Power Modes.

7.25.1.1 Reset_GetCause()

This function reads the Reset Cause Register and stores the cause bits in the result structure pointer. It copies the result to the global variable `stc_reset_result_t stcStoredResetCause`.

Attention:
Calling this function clears all bits in the Reset Cause Register `RST_STR`! `Reset_GetCause`() should only be called after Start-up code!

Prototype	
`en_result_t Reset_GetCause( stc_reset_result_t* pstcResult );`	
Parameter Name	**Description**
`[out] pstcResult`	Pointer to cause structure to be written to
Return Values	**Description**
`Ok`	Successfully read

Structure of the type of `stc_reset_result_t`.

Element Type	Element Name	Description
`boolean_t`	`bPowerOn`	`TRUE`: Power on reset occurred
`boolean_t`	`bInitx`	`TRUE` : `INITX` (external) reset occurred
`boolean_t`	`bLowVoltageDetection`	`TRUE` : Low Voltage Detection reset occurred (only applicable for Type3 and 7, always `FALSE` otherwise)
`boolean_t`	`bSoftwareWatchdog`	`TRUE` : Software Watchdog reset occurred
`boolean_t`	`bHardwareWatchdog`	`TRUE` : Hardware Watchdog reset occurred
`boolean_t`	`bClockSupervisor`	`TRUE` : Clock Supervisor reset occurred
`boolean_t`	`bAnomalousFrequency`	`TRUE` : Anomalous Frequency reset occurred
`boolean_t`	`bSoftware`	`TRUE` : Software reset ocurred

7.25.1.2 Reset_GetStoredCause()

This function copies the global variable global variable `stc_reset_result_t stcStoredResetCause` to a result structure pointer.

Prototype	
`en_result_t Reset_GetStoredCause( stc_reset_result_t* pstcResult );`	
Parameter Name	**Description**
`[out] pstcResult`	Pointer to cause structure to be written to
Return Values	**Description**
`Ok`	Successfully read

The structure for pstcResult is the same as described in the paragraph above (7.25.1.1).

7.25.2 RESET Example

The PDL example folder does not provide a REST example.

7.26 (RTC) Real Time Clock

Type Definition	-
Configuration Type	`stc_rtc_config_t`
Address Operator	-

`Rtc_Init()` initializes the RTC block with given time and date. It also initializes the NVIC, if specidfied.

`Rtc_SetDateTime()` sets a new date and time.

`Rtc_SetAlarmDateTime()` sets a new alarm date and time

`Rtc_EnableDisableInterrupts()` enables/disables interrupt configurations.

`Rtc_EnableDisableAlarmRegisters()` adjusts the time.

`Rtc_ReadDateTimePolling()` retrieves recent date and time to the members of the structure.

`Rtc_ReadClockStatus()` reads out the status of the RTC.

`Rtc_TimerSet()` sets the timer value for its interval or one-shot counting.

`Rtc_TimerStart()` starts and `Rtc_TimerStop()` stops the RTC timer.

`Rtc_TimerStatusRead()` reads out the RTC timer status.

`Rtc_TransStatusRead()` reads out the transmission status.

`Rtc_DeInit()` deinitializes all RTC functions and interrupts. Also the NVIC registers can be set.

`Rtc_GetRawTime()` calculates the 'raw' time (UNIX time) from the RTC time structure `stc_rtc_time_t`.

`Rtc_SetDayOfWeek()` sets the day of the week calculated from the date and time given in `stc_rtc_time_t`.

`Rtc_SetTime()` calculates the RTC time structure from the 'raw' time.

`Rtc_WriteBkupReg8()`, `Rtc_WriteBkupReg16()` and `Rtc_WriteBkupReg32()` puts a single byte, 16-bit or 32-bit data to given backup register area.

`Rtc_ReadBkupReg8()`, `Rtc_ReadBkupReg16()` and `Rtc_ReadBkupReg32()` read a single byte, 16-bit or 32-bit word from a given backup register area.

Notes: Before this driver initializes, VBAT domain has to be initialized and SUB clock has to be enabled. This driver uses the standard C library *`time.h`*.

7.26.1 RTC Configuration Structures

7.26.1.1 RTC Configuration

The RTC driver library uses the following structure of configuration. the type `stc_rtc_config_t`:

Type	Field	Possible Values	Description
`boolean_t`	`bUseFreqCorr`	TRUE FALSE	Use Frequency correction value Don't use Frequency correction value
`boolean_t`	`bUseDivider`	TRUE FALSE	Enable Divider for Divider Ratio Disable Divider for Divider Ratio
`uint16_t`	`u16FreqCorrValue`	-	Frequency correction value (10bit for WTCAL)
`uint8_t`	`u8FreqCorrValue0`	-	Frequency correction value (Lower 8bit for WTCAL0)
`uint8_t`	`u8FreqCorrValue1`	-	Frequency correction value (Upper 2bit for WTCAL1)
`uint8_t`	`u8DividerRatio`	RtcDivRatio …1 …2 …4 …8 …16 …32 …64 …128 …256 …512 …1024 …2048 …4096 …8192 …16384 …32768	RIN clock division ration: No division Divided by 2 Divided by 4 Divided by 8 Divided by 16 Divided by 32 Divided by 64 Divided by 128 Divided by 256 Divided by 512 Divided by 1024 Divided by 2048 Divided by 4096 Divided by 8192 Divided by 16384 Divided by 32768
`uint8_t`	`u8FreqCorrCycle`	0 - 0x3F	Frequency correction cycle value
`uint8_t`	`u8CoSIgnalDiv`	RtcCoDiv1 RtcCoDiv2	CO signal of RTC count part is output CO signal of RTC divided by 2 is output
`stc_rtc_... interrupts_t`	`stcRtcInterrupt… Enable`		
`uint8_t`	`u8AllInterrupts`		

Type	Field	Possible Values	Description
stc_rtc_... alarm_... enable_t	stcAlarmRegister.. .Enable		
func_ptr_... rtc_arglist_ t	pfnReadCallback	-	Callback function pointer for read completion Interrupt
func_ptr_t	pfnTimeWrtErr... Callback	-	Callback function pointer for Timer writing error Interrupt
func_ptr_t	pfnAlarmCallback	-	Callback function pointer for Alarm Interrupt
func_ptr_t	pfnTimerCallback	-	Callback function pointer for Timer Interrupt
func_ptr_t	pfnHalfSecond... Callback	-	Callback function pointer for 0.5-Second Interrupt
func_ptr_t	pfnOneSecond... Callback	-	Callback function pointer for One-Second Interrupt
func_ptr_t	pfnOneMinute... Callback	-	Callback function pointer for One-Minute Interrupt
func_ptr_t	pfnOneHour... Callback	-	Callback function pointer for One-Hour Interrupt

7.26.1.2 RTC Interrupts Enable Structure

The structure of the type `stc_rtc_interrupts_t` used by the configuration has following bit fields:

Field	Possible Values	Description
ReadCompletionIrqEn	0 1	RTC Read Completion interrupt disabled RTC Read Completion interrupt enabled
TimeRewriteErrorIrqEn	0 1	Time rewrite error interrupt disabled Time rewrite error interrupt enabled
AlarmIrqEn	0 1	RTC alarm interrupt disabled RTC alarm interrupt enabled
TimerIrqEn	0 1	RTC timer interrupt disabled RTC timer interrupt disabled
OneHourIrqEn	0 1	One-Hour interrupt disabled One-Hour interrupt Enabled
OneMinuteIrqEn	0 1	One-Minute interrupt disabled One-Minute interrupt enabled
OneSecondIrqEn	0 1	One-Second interrupt disabled One-Second interrupt enabled
HalfSecondIrqEn	0 1	Half-Second interrupt disabled Half-Second interrupt enabled

7.26.1.3 Alarm Register Enable Structure

The structure of the type `stc_rtc_alarm_enable_t` used by the configuration has following bit fields:

Field	Possible Values	Description
`AlarmYearEnable`	0 1	RTC alarm year register disabled RTC alarm year register enabled
`AlarmMonthEnable`	0 1	RTC alarm month register disabled RTC alarm month register enabled
`AlarmDayEnable`	0 1	RTC alarm day register disabled RTC alarm day register enabled
`AlarmHourEnable`	0 1	RTC alarm hour register disabled RTC alarm hour register enabled
`AlarmMinuteEnable`	0 1	RTC alarm minute register disabled RTC alarm minute register enabled

7.26.1.4 RTC Time structure

The structure of the type `stc_rtc_time_t` has following fields:

Type	Field	Possible Values	Description
`uint8_t`	`u8Second`	0...59	RTC second
`uint8_t`	`u8Minute`	0...59	RTC minute
`uint8_t`	`u8Hour`	0...23	RTC hour
`uint8_t`	`u8Day`	1...31	RTC day
`uint8_t`	`u8DayOfWeek`	0...6	RTC day of week (0 == Sunday)
`uint8_t`	`u8Month`	1...12	RTC month
`uint8_t`	`u8Year`	1...99 (+2000)	RTC year

7.26.1.5 RTC Alarm structure

The structure of the type `stc_rtc_alarm_t` has following fields:

Type	Field	Possible Values	Description
uint8_t	u8AlarmMinute	0...59	Alarm minute
uint8_t	u8AlarmHour	0...23	Alarm hour
uint8_t	u8AlarmDay	1...31	Alarm day
uint8_t	u8AlarmMonth	1...12	Alarm month
uint8_t	u8AlarmYear	1...99 (+2000)	Alarm year

7.26.1.6 RTC Timer Configuration

The RTC’s timer configuration of the type `stc_rtc_timer_config_t` has following fields:

Type	Field	Possible Values	Description
`uint32_t`	`u32TimerValue`	`1…172799`	18-bit value for RTC timer
`uint8_t`	`u8TimerValue0`	-	RTC Timer value bit7-0 for WTTR0
`uint8_t`	`u8TimerValue1`	-	RTC Timer value bit15-8 for WTTR0
`uint8_t`	`u8TimerValue2`	-	RTC Timer value bit17,16 for WTTR2
`boolean_t`	`bTimerIntervalEnable`	`0` `1`	Timer is set in count interval mode Timer is set in one-shot mode

7.26.2 RTC Definitions

For Month and Day of the week values, *rtc.h* provides definitions as shown as below. Also the Year can be stated as YYYY format by using `Rtc_Year()` macro.

```
/**
 ******************************************************************************
 ** \brief Year calculation macro for adjusting RTC year format

******************************************************************************/
#define Rtc_Year(a) (a - 2000)

...

/**
 ******************************************************************************
 ** \brief Month name definitions (not used in driver - to be used by
 **        user applciation)

******************************************************************************/
typedef enum en_rtc_month
{
    Rtc_January     = 1,
    Rtc_Febuary     = 2,
    Rtc_March       = 3,
    Rtc_April       = 4,
    Rtc_May         = 5,
    Rtc_June        = 6,
    Rtc_July        = 7,
    Rtc_August      = 8,
    Rtc_September   = 9,
    Rtc_October     = 10,
    Rtc_November    = 11,
    Rtc_December    = 12
} en_rtc_month_t;

/**
 ******************************************************************************
 ** \brief Day of week name definitions (not used in driver - to be used by
 **        user applciation)

******************************************************************************/
typedef enum en_rtc_day_of_week
{
    Rtc_Sunday      = 0,
    Rtc_Monday      = 1,
    Rtc_Tuesday     = 2,
    Rtc_Wednesday   = 3,
    Rtc_Thursday    = 4,
    Rtc_Friday      = 5,
    Rtc_Saturday    = 6
} en_rtc_day_of_week_t;
```

7.26.3 API Reference

7.26.3.1 Rtc_Init()

Initializes the RTC block and sets up the internal data structures. The VBAT domain initialization, the sub clock enabling and stabilization should be done before this function is called. The user should define the recent time in the structure of the type `stc_rtc_time_t`.
Also the structure of the type `stc_rtc_alarm_t` should be defined even if the alarm feature is not used. The structure fields may contain NULL in this case.

Prototype	

`en_result_t Rtc_Init(stc_rtc_config_t* pstcConfig,`
`stc_rtc_time_t* pstcTime,`
`stc_rtc_alarm_t* pstcAlarm,`
`boolean_t bTouchNvic )`	

Parameter Name	**Description**
[in] `pstcConfig`	A pointer to a structure of the RTC configuration
[in] `pstcTime`	A pointer to a structure of the RTC Time
[in] `pstcAlarm`	A Pointer to a structure of the RTC Alarm This can be set to NULL
[in] `bTouchNvic`	`TRUE` = touch shared NVIC registers `FALSE` = do not touch NVIC
Return Values	**Description**
`Ok`	Initializing the RTC block ended with no error

`ErrorInvalidParameter`	`pstcConfig == NULL`
	`pstcTime == NULL`
	Invalid value is set to one of members of `pctcConfig`
	Invalid value is set to one of members of `pctcTime`
	Invalid value is set to one of members of `pctcAlamr`

ErrorTimeout	Timeout to complete transmission

7.26.3.2 Rtc_DeInit()

Deinitializes the RTC block.

Prototype	
`en_result_t Rtc_DeInit(void)`	
Return Values	**Description**
`Ok`	Deinitializing the RTC block ended with no error
`ErrorTimeout`	Timeout to complete transmission

7.26.3.3 Rtc_SetDateTime()

Sets the date and time to the RTC registers.

Prototype	
`en_result_t Rtc_SetDateTime(stc_rtc_time_t* pstcRtcTime)`	
Parameter Name	**Description**
[in] `pstcRtcTime`	A pointer to the RTC Time structure
Return Values	**Description**
`Ok`	Setting the date and time ended with no error
`ErrorInvalidParameter`	`pstcRtcTime == NULL` Invalid value is set to one of members of `pctcRtcTime`
`ErrorTimeout`	Timeout to complete transmission

7.26.3.4 Rtc_SetAlarmDateTime()

Sets the RTC alarm time.

Prototype	
`en_result_t Rtc_SetAlarmDateTime(stc_rtc_alarm_t* pstcRtcAlarm)`	
Parameter Name	**Description**
[in] `pstcRtcAlarm`	A pointer to the RTC Alarm structure
Return Values	**Description**
`Ok`	Setting the alarm date and time ended with no error
`ErrorInvalidParameter`	`pstcRtcAlarm == NULL` Invalid value is set to one of members of `pctcRtcAlarm`
`ErrorTimeout`	Timeout to complete transmission

7.26.3.5 Rtc_EnableDisableInterrupts()

Enable or disable the RTC interrupts.

Prototype	
`en_result_t Rtc_EnableDisableInterrupts(stc_rtc_config_t* pstcConfig)`	
Parameter Name	**Description**
[in] `pstcConfig`	A pointer to a structure of the RTC configuration
Return Values	**Description**
`Ok`	Enableing/disabling the RTC interrupts ended with no error
`ErrorInvalidParameter`	`pstcRtcConfig == NULL`

7.26.3.6 Rtc_EnableDisableAlarmRegisters()

Enable or disable the alarm registers.

Prototype	
`en_result_t Rtc_EnableDisableAlarmRegisters(stc_rtc_alarm_enable_t* pstcRtcAlarmEn)`	
Parameter Name	**Description**
[in] `pstcRtcAlarmEn`	A pointer to a structure of the RTC alarm enable or disable
Return Values	**Description**
`Ok`	Enabling/disabling RTC alarm ended with no error
`ErrorInvalidParameter`	`pstcRtcAlarmEn == NULL`
`ErrorTimeout`	Timeout to complete transmission

7.26.3.7 Rtc_ReadDateTimePolling()

Reads a recent time. The recent time is read out to a pointer of a structure of the RTC time.

Notes: This function disables the interreuption, CRI.

Prototype	
`en_result_t Rtc_ReadDateTimePolling(stc_rtc_time_t* pstcRtcTime)`	
Parameter Name	**Description**
[in] `pstcRtcTime`	A pointer to a structure of the RTC time
Return Values	**Description**
`Ok`	Reading out date and time to `pstcRtcTime` ended with no error
`ErrorInvalidParameter`	`pstcRtcTime == NULL`
`ErrorTimeout`	Timeout occurs

7.26.3.8 Rtc_RequestDateTime()

Reads a recent time and copy it to the RTC registers. In the RTC ISR the callback function `stc_rtc_config_t::pfnReadCallback` with all of the 7 arguments for date and time is called.

Notes: This function needs `INTCRIE` bit to be set to '1' by `Rtc_Init()` or `Rtc_EnableDisableInterrupts()`.

Prototype	
`en_result_t Rtc_RequestDateTime(void)`	
Return Values	**Description**
`Ok`	Request started with no error
`ErrorNotReady`	Request was not accepted because previous request is on going

7.26.3.9 *Rtc_TimerSet()*

Sets a mode and a timer value to the RTC block. The RTC block has to be initialized with `Rtc_Init()` before calling this function.

Prototype	
`en_result_t Rtc_TimerSet(stc_rtc_timer_config_t* pstcConfig)`	
Parameter Name	**Description**
[in] `pstcConfig`	A pointer to a structure of the RTC configuration
Return Values	**Description**
`Ok`	Setting a mode and a timer value endded with no error
`ErrorInvalidParameter`	`pstcConfig == NULL` Invalid value is set to one of members of pctcConfig
`ErrorTimeout`	Timeout occurs

7.26.3.10 *Rtc_TimerStart()*

Starts the Timer of the RTC timer. The RTC block has to be initialized with `Rtc_Init()` before calling this function.

Prototype	
`en_result_t Rtc_TimerStart(void)`	
Return Values	**Description**
`Ok`	The timer has started with no error
`ErrorTimeout`	Timeout occurs

7.26.3.11 *Rtc_TimerStop()*

Stops a Timer of the RTC block. The RTC block has to be initialized with `Rtc_Init()` before calling this function.

Prototype	
`en_result_t Rtc_TimerStop(void)`	
Return Values	**Description**
`Ok`	Timer successfully stopped
`ErrorTimeout`	Timeout to complete transmission

7.26.3.12 Rtc_TimerStatusRead()

Rrovides a status of the `TMRUN` in the `WTCR21` register. It returns the type `en_rtc_timer_status_t` as describes below.

Prototype	
`en_rtc_timer_status_t Rtc_TimerStatusRead(void)`	
Return Values	**Description**
`RtcTimerNoOperation`	The timer was not in operation
`RtcTimerInOperation`	The timer was in operation

7.26.3.13 Rtc_TransStatusRead()

Povides a status of the `TRANS` in the `WTCR10` register.

Prototype	
`en_rtc_timer_status_t Rtc_TransStatusRead(void)`	
Return Values	**Description**
`RtcTransNoOperation`	Transmission is not operating
`RtcTransInOperation`	Transmission is operating

7.26.3.14 Rtc_GetRawTime()

Calculates the “raw” time (‘UNIX time’). It uses `mktime()` of the *`time.h`* library.

Prototype	
`time_t Rtc_GetRawTimer(stc_rtc_time_t* pstcRtcTime)`	
Parameter Name	**Description**
[in] `pstcRtcTime`	A pointer to a structure of the RTC time
Return Values	**Description**
`time_t`	Calculated time or -1 on error

7.26.3.15 Rtc_SetDayOfWeek()

Calculates a day of the week from YY-MM-DD in the Time structure. It uses `mktime()` of *`time.h`* library.

Prototype	
`en_result_t Rtc_SetDayOfWeek(stc_rtc_time_t* pstcRtcTime)`	
Parameter Name	**Description**
[in] `pstcRtcTime`	A pointer to a structure of the RTC time
Return Values	**Description**
`Ok`	Setting day of the week ended with no error
`ErrorInvalidParameter`	`pstcRtcTImer == NULL` Calling `mktime()` failed

7.26.3.16 Rtc_SetTime()

Sets the RTC time to a time structure. It uses `localtime()` in the `time.h` library.

Prototype		
`en_result_t Rtc_SetTime(`	`stc_rtc_time_t*` `time_t`	`pstcRtcTime,` `tRawTime)`

Parameter Name	Description
[in] `pstcRtcTime`	A pointer to a structure of the RTC time
[in] `tRawTime`	“raw” time

Return Values	Description
`Ok`	Setting a RTC time structure ended with no error
`ErrorInvalidParameter`	`pstcRtcTImer == NULL` Calling `localtime()` failed

7.26.3.17 Rtc_WriteBkupReg8()

Writes data to the backup register by byte access.

Prototype		
`en_result_t Rtc_WriteBkupReg8(`	`uint8_t` `uint8_t`	`u8Data,` `u8Area)`

Parameter Name	Description
[in] `u8Data`	Data for backup
[in] `u8Area`	Backup register area. (*)

Return Values	Description
`Ok`	Setting data to a backup area ended with no error
`ErrorInvalidParameter`	Backup registers are out of the range
`ErrorTimeout`	Timeout occurs

(*)This should be specified by `RtcBkupRegAreaXX`.

7.26.3.18 *Rtc_WriteBkupReg16()*

Writes data to the Backup Register (with 16-byte access).

Prototype	
`en_result_t Rtc_WriteBkupReg16( uint16_t u16Data, uint8_t u8Area)`	
Parameter Name	**Description**
[in] `u16Data`	Data for backup
[in] `u8Area`	Backup register area. (*)
Return Values	**Description**
`Ok`	Setting data to a backup area ended with no error
`ErrorInvalidParameter`	Backup registers are out of the range
`ErrorTimeout`	Timeout occurs

(*)This should be specified by “`RtcBkupRegAreaXX`”

7.26.3.19 *Rtc_WriteBkupReg32()*

Writes data to the Backup Register (with 32-bit access).

Prototype	
`en_result_t Rtc_WriteBkupReg32( uint32_t u32Data, uint8_t u8Area)`	
Parameter Name	**Description**
[in] `u32Data`	Data for backup
[in] `u8Area`	Backup register area (*)
Return Values	**Description**
`Ok`	Setting data to a backup area ended with no error
`ErrorInvalidParameter`	Backup register area is out of range
`ErrorTimeout`	Timeout to complete transmission

(*)This should be specified by “`RtcBkupRegAreaXX`”

7.26.3.20 *Rtc_ReadBkupReg8()*

Reads out data from Backup Register (with byte access).

Prototype	
`uint8_t Rtc_ReadBkupReg8(uint8_t u8Area)`	
Parameter Name	**Description**
[in] `u8Area`	Backup register area (*)
Return Values	**Description**
uint8_t	Backup register value

(*)This should be specified by “`RtcBkupRegAreaXX`”.

7.26.3.21 *Rtc_ReadBkupReg16()*

Reads out data from Backup Register (with 16-bit access).

Prototype	
`uint16_t Rtc_ReadBkupReg16(uint8_t u8Area)`	
Parameter Name	**Description**
[in] `u8Area`	Backup register area (*)
Return Values	**Description**
`uint16_t`	Backup register value

(*)This should be specified by “`RtcBkupRegAreaXX`”.

7.26.3.22 *Rtc_ReadBkupReg32()*

Reads out data from Backup Register (with 32-bit access).

Prototype	
`uint32_t Rtc_ReadBkupReg32(uint8_t u8Area)`	
Parameter Name	**Description**
[in] `u8Area`	Backup register area (*)
Return Values	**Description**
`uint32_t`	Backup register value

(*)This should be specified by “`RtcBkupRegAreaXX`”.

7.26.3.23 Callback functions

The callback functions is registered by `Rtc_Init()`. There are eight callbacks.

Callback function for read completion Interrupt

The callback function is called when read completion interrupt (`INTCRI`) is generated.

Prototype

```
void (*func_ptr_rtc_arglist_t)(   uint8_t     u8Second,
           uint8_t     u8Minute,
           uint8_t     u8Hour,
           uint8_t     u8Day,
           uint8_t     u8DayOfWeek,
           uint8_t     u8Month
           uint8_t     u8Year)
```

Parameter Name	Description
[in] `u8Second`	RTC second
[in] `u8Minute`	RTC minute
[in] `u8Hour`	RTC hour
[in] `u8Day`	RTC day
[in] `u8DayOfWeek`	RTC day of the week (`0` == Sunday)
[in] `u8Month`	RTC month
[in] `u8Year`	RTC year

Other callbacks

There are 7 other callbacks:

The callback is called when time rewriting error interrupt (`INTERI`) was generated.

The callback is called when alarm coincidence interrupt (`INTALI`) was generated.

The callback is called when Timer underflow detection interrupt (`INTTMI`) was generated.

The callback is called when every hour interrupt (`INTHI`) was generated.

The callback is called when every minute interrupt (`INTMI`) was generated.

The callback is called when every second interrupt (`INTSI`) was generated.

The callback is called when every half-second interrupt (`INTSSI`) was generated.

These callbacks are registered individually.

Prototype

```
void (*func_prt_t)(void)
```

7.26.4 Example Code

The example software is in *\example\rtc*.

Folder *\example\rtc\..*	**Summary**
rtc_time_count	RTC time count and alarm action
rtc_timer_interval	RTC time count and RTC timer using interval mode
rtc_timer_oneshot	RTC time count and RTC timer using one-shot mode

7.26.4.1 RTC

This example software excerpt shows time count and alarm usage.

```
#include "rtc/rtc.h"

static stc_rtc_config_t    stcRtcConfig;// recommend to be global
static stc_rtc_time_t      stcRtcTime; // recommend to be global
static stc_rtc_alarm_t     stcRtcAlarm;// recommend to be global
...
static void SampleRtcReadCb(         uint8_t u8Sec,
       uint8_t u8Min,
       uint8_t u8Hour,
       uint8_t u8Day,
       uint8_t u8DayOfWeek,
       uint8_t u8Month,
       uint8_t u8Year)
{
  // some code here ...
}

static void SampleRtcTimeWrtErrCb(void)
{
  // some code here ...
}

static void SampleRtcAlarmCb(void)
{
  // some code here ...
}

static void SampleRtcHalfSencondCb(void)
{
  // some code here ...
}

static void SampleRtcOneSencondCb(void)
{
  // some code here ...
}
```

```
function
{
  en_result_t enResult;
  ...
  // Set the RTC configuration
  stcRtcConfig.bUseFreqCorr          = FALSE;
  stcRtcConfig.bUseDivide   = FALSE;
  stcRtcConfig.u16FreqCorrValue      = 0;
  stcRtcConfig.u8DividerRatio        = RtcDivRatio1;
  stcRtcConfig.u8FreqCorrCycle       = 0x13;
  stcRtcConfig.u8CoSignalDiv         = RtcCoDiv1;
  stcRtcConfig.u8AllInterrupts       = 0xC7;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmYearEnable  = TRUE;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmMonthEnable= TRUE;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmDayEnable   = TRUE;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmHourEnable  = TRUE;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmMinuteEnable= TRUE;
  stcRtcConfig.pfnReadCallback       = SampleRtcReadCb;
  stcRtcConfig.pfnTimeWrtErrCallback= SampleRtcTimeWrtErrCb;
  stcRtcConfig.pfnAlarmCallback      = SampleRtcAlarmCb;
  stcRtcConfig.pfnTimerCallback      = NULL;
  stcRtcConfig.pfnHalfSecondCallback= SampleRtcHalfSencondCb;
  stcRtcConfig.pfnOneSecondCallback = SampleRtcOneSencondCb;
  stcRtcConfig.pfnOneMinuteCallback = NULL;
  stcRtcConfig.pfnOneHourCallback    = NULL;

  // Default RTC setting (23:59:00 31th of May 2013)
  stcRtcTime.u8Second       = 0;      // Second         : 00
  stcRtcTime.u8Minute       = 59;     // Minutes        : 59
  stcRtcTime.u8Hour= 23;              // Hour           : 23
  stcRtcTime.u8Day = 31;              // Date           : 31th
  stcRtcTime.u8Month        = Rtc_May;// Month          May
  stcRtcTime.u8Year= Rtc_Year(2013);// Year           : 2013
  (void)Rtc_SetDayOfWeek(&stcRtcTime);// Set Day of the Week in
stcRtcTime

  // Alarm setting (00:00:00 1st of June 2013)
  stcRtcAlarm.u8AlarmMinute= 0;       // Minutes        : 00
  stcRtcAlarm.u8AlarmHour= 0;         // Hour  : 00
  stcRtcAlarm.u8AlarmDay              = 1;     // Date  : 1st
  stcRtcAlarm.u8AlarmMonth = Rtc_June;         // Month : June
  stcRtcAlarm.u8AlarmYear= Rtc_Year(2013);     // Year  : 2013
  ...
```

```
  ...
  // Initialize the RTC
  enResult = Rtc_Init(&stcRtcConfig, &stcRtcTime, &stcRtcAlarm, TRUE);
  if (Ok != enResult)
  {
    // some code here ...
    while (1);
  }

  while (1)
  {
    // some code here ...
  }
}
```

7.26.4.2 RTC Timer

This example software code excerpt shows interval mode.

```
#include "rtc/rtc.h"

static stc_rtc_config_t    stcRtcConfig;      // recommend to be global
static stc_rtc_time_t      stcRtcTime;        // recommend to be global
...
static void SampleRtcReadCb(uint8_t u8Sec,
                            uint8_t u8Min,
                            uint8_t u8Hour,
                            uint8_t u8Day,
                            uint8_t u8DayOfWeek,
                            uint8_t u8Month,
                            uint8_t u8Year)
{
  // some code here ...
}

static void SampleRtcTimeWrtErrCb(void)
{
  // some code here ...
}

static void SampleRtcTimerCb(void)
{
  // some code here ...
}

static void SampleRtcHalfSencondCb(void)
{
  // some code here ...
}

static void SampleRtcOneSencondCb(void)
{
  // some code here ...
}
```

```
function
{
  stc_rtc_timer_config_t   stcRtcTimer;
  en_result_t                          enResult;
  ...
  // Set the RTC configuration
  stcRtcConfig.bUseFreqCorr          = FALSE;
  stcRtcConfig.bUseDivider = FALSE;
  stcRtcConfig.u16FreqCorrValue      = 0;
  stcRtcConfig.u8DividerRatio        = RtcDivRatio1;
  stcRtcConfig.u8FreqCorrCycle       = 0x13;
  stcRtcConfig.u8CoSignalDiv                  = RtcCoDiv1;
  stcRtcConfig.u8AllInterrupts       = 0xCB;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmYearEnable  = FALSE;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmMonthEnable= FALSE;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmDayEnable   = FALSE;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmHourEnable  = FALSE;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmMinuteEnable= FALSE;
  stcRtcConfig.pfnReadCallback       = SampleRtcReadCb;
  stcRtcConfig.pfnTimeWrtErrCallback= SampleRtcTimeWrtErrCb;
  stcRtcConfig.pfnAlarmCallback      = NULL;
  stcRtcConfig.pfnTimerCallback      = SampleRtcTimerCb;
  stcRtcConfig.pfnHalfSecondCallback= SampleRtcHalfSencondCb;
  stcRtcConfig.pfnOneSecondCallback = SampleRtcOneSencondCb;
  stcRtcConfig.pfnOneMinuteCallback = NULL;
  stcRtcConfig.pfnOneHourCallback    = NULL;

  // Default RTC setting (23:59:00 31th of May 2013)
  stcRtcTime.u8Second      = 0;     // Second          : 00
  stcRtcTime.u8Minute      = 59;    // Minutes         : 59
  stcRtcTime.u8Hour= 23;            // Hour            : 23
  stcRtcTime.u8Day = 31;            // Date            : 31th
  stcRtcTime.u8Month       = Rtc_May;// Month          : May
  stcRtcTime.u8Year= Rtc_Year(2013);// Year            : 2013
  (void)Rtc_SetDayOfWeek(&stcRtcTime);// Set Day of the Week in
stcRtcTime

  // Initialize the RTC
  enResult = Rtc_Init( &stcRtcConfig, &stcRtcTime, NULL, TRUE);
  if (Ok != enResult)
  {
    // some code here ...
    while (1);
  }
  ...
```

```
  ...
  // Timer set (interruption per 10sec)
  // Set the 10sec to RTC timer format ((10 * 2)-1)
  stcRtcTimer.u32TimerValue          = 19;
  stcRtcTimer.bTimerIntervalEnable   = TRUE;
  enResult = Rtc_TimerSet(&stcRtcTimer);
  if (Ok == enResult)
  {
    // Timer start
    enResult = Rtc_TimerStart();
  }

  if (Ok != enResult)
  {
    // some code here ...
    while (1);
  }

  while (1)
  {
    // some code here ...
  }
}
```

7.27 (SD) SD Card Inerface

Type Definition	-
Configuration Types	stc_sd_comdconfig_t stc_sd_config_t
Address Operator	-

Before using SD card, Sd_HostInit() should be called at first time to initializes SD host. Also can call Sd_HostDeInit() to reset and close sd host.

Call Sd_SoftwareReset() to do a soft reset of command line / data line reset.

A SD interrupt can be enabled by the function Sd_EnableInt() and disabled by calling Sd_DisableInt().

Note, interrupt mode is not implemented, so the callback function is not avialable.

Call Sd_CardDetect() to get SD card status(inserted or not existed in a slot).

Call Sd_ClockSupply() to set the sd host clock frequency supply to a SD card.

Call Sd_ClockStop() to disable the output of SD clock.

when detect a Card, call Sd_SendCmd () to send SD command to control the card.

After a SD card is initialized by a series of SD command, call Sd_TxData() to send data or Sd_RxData() to read data.

7.27.1 Configuration Structure

A SD command configure data instance uses the following configuration structure of the type of `stc_sd_comdconfig_t`:

Type	Field	Possible Values	Description
en_sd_response_t	enResponse	NO_RSP R1NORMAL_R5_R6_R7 R1AUTO R1B_NORMAL R1B_AUTO R2 R3_R4 R5B	SD response (see chapter 2.2.6 SD specifications (v2.00) Part A2 for details)
en_sd_dir_t	enReadWrite	SD_WRITE SD_READ	SD transfer direction host to card card to host
en_sd_scmd_type_t	enCmdType	NORMAL SUSPEND RESUME ABORT	SD special command index Non special command Bus Suspend Function Select I/O abort
boolean_t	bDataPresent	TRUE FALSE	Indicate if a command have data to transfer.
uint8_t	u8Index		SD command number.

Type	Field	Possible Values	Description
		0 - 63	(see chapter 2.2.6 SD specifications (v2.00) Part A2 for details)
en_sd_auto_cmd_t	enAutoCmd	AUTO_DISABLE AUTO_CMD12 AUTO_CMD23	SD auto command functions
uint16_t	u16BlockCount	0 - FFFF	Config the number of data blocks. (see chapter 2.2.6 SD specifications (v2.00) Part A2 for details)
uint16_t	u16BlockSize	0 - 0x800	Configiure the number of bytes in a data block. (see chapter 2.2.6 SD specifications (v2.00) Part A2 for details)
Uint32_t	u32Argument1		(see chapter 2.2.6 SD specifications (v2.00) Part A2 for details)
uint32_t	u32SysAddr_Arg2		
en_sd_boundary_t	enbound	BOUND_4K BOUND_8K BOUND_16K BOUND_32K BOUND_64K BOUND_128K BOUND_256K BOUND_512K	(see chapter 2.2.6 SD specifications (v2.00) Part A2 for details)
uint32_t *	pu32Buffer		Pointer to a data buffer.
func_ptr_sd_arg32_t	pfnErrorResponseCallback		Pointer to a error response callback function.

A SD configure data instance uses the following configuration structure of the type of stc_sd_config_t:

Type	Field	Possible Values	Description
en_sd_existing_t	enExist	DEBOUNCING INSERTED REMOVAL	SD card exist status. Debouncing sd card is inserted sd card is removed
en_sd_buswidth_t	enBusWidth	BIT_1 BIT_4	SD bus width 1 bit mode 4 bit mode
boolean_t	bCmdComplete	TRUE FALSE	SD cmd complete flag. (This flag return true when a command is done.)
boolean_t	bSendComplete	TRUE FALSE	SD cmd send complete flag. (This flag return true when a command transfer is done.)
boolean_t	bDMAComplete	TRUE FALSE	
func_ptr_sd_arg32_t	pfnTxCallback	-	Pointer to a tx callback function.
func_ptr_sd_arg32_t	pfnRxCallback	-	Pointer to a rx callback function.
func_ptr_sd_arg32_t	pfnWakeupCallback	-	Pointer to a wakeup callback function.
func_ptr_sd_arg32_t	pfnErrorCallback	-	Pointer to a error callback function.

7.27.2 SD Card API

7.27.2.1 Sd_HostInit ()

This function initializes SD host.

Prototype	
en_result_t Sd_HostInit(void);	
Parameter Name	**Description**
-	
Return Values	**Description**
Ok	SD host init successfully.
Error	No SD card exist. SD I/F peripheral clock enable fail.

7.27.2.2 Sd_HostDeInit ()

This function de-initializes SD host.

Prototype	
en_result_t Sd_HostDeInit(void);	
Parameter Name	**Description**
-	
Return Values	**Description**
Ok	SD host de-init successfully.

7.27.2.3 Sd_SoftwareReset ()

This function does software reset for command and data line.

Prototype	
en_result_t Sd_SoftwareReset(volatile stc_sd_t* pstcSd,uint8_t u8reset);	
Parameter Name	**Description**
[in] pstcSd	Pointer to a SDIF instance.
[in] u8reset	Options of reset setting,such as command ,data line and all.
Return Values	**Description**
Ok	Software reset done sucessfully.
ErrorInvalidParameter	pstcSd == NULL

7.27.2.4 Sd_CardDetect ()

This function detects the SD card insertion status.

Prototype	
en_sd_existing_t Sd_CardDetect(volatile stc_sd_config_t* pstcCfg);	
Parameter Name	**Description**
[in] pstcCfg	Pointer to command parameters.
Return Values	**Description**
DEBOUNCING	SD card is debouncing. pstcCfg == NULL
INSERTED	SD card is inserted.
REMOVAL	SD card is removed.

7.27.2.5 Sd_ClockSupply ()

This function sets supply clock frequence to a SD card.

Prototype	
`en_result_t Sd_ClockSupply(volatile stc_sd_t* pstcSd,en_sd_clk_t enClk);`	
Parameter Name	**Description**
[in] pstcSd	Pointer to a SDIF instance.
[in] enClk	SD clock frequency.
Return Values	**Description**
Ok	Clock set successfully.
ErrorInvalidParameter	pstcSd == NULL

7.27.2.6 Sd_ClockStop ()

This function stops SD supply clock.

Prototype	
`en_result_t  Sd_ClockStop(volatile stc_sd_t* pstcSd);`	
Parameter Name	**Description**
[in] pstcSd	Pointer to a SDIF instance.
Return Values	**Description**
Ok	Clock stops successfully.
ErrorInvalidParameter	pstcSd == NULL

7.27.2.7 Sd_SendCmd ()

This function transmits commands and checks the corresponding response.

Prototype	
`boolean_t Sd_SendCmd(en_sd_transaction_t enTran,volatile stc_sd_t* pstcSd, stc_sd_comdconfig_t * pstcCmdCfg,uint32_t* pu32Buf);`	
Parameter Name	**Description**
[in]enTran	Transmission type depends on the CMD and expected response.
[in] pstcSd	Pointer to a SDIF instance.
[in] pstcCmdCfg	Pointer to a Command parameters.
[in] pu32Buf	Pointer to data buffer to recieved data.
Return Values	**Description**
TRUE	Data is transmitted successfully.
FAlSE	Error occurs during transmission.

7.27.2.8 Sd_Handler ()

This function is SD interrupt callback function.

Prototype	
`void Sd_Handler(void);`	
Parameter Name	**Description**
-	
Return Values	**Description**
-	.

7.27.2.9 Sd_EnableInt ()

This function enables the interrupt flag of SD.

Prototype	
`en_result_t Sd_EnableInt(volatile stc_sd_t* pstcSd );`	
Parameter Name	**Description**
`[in] pstcSd`	Pointer to a SDIF instance.
Return Values	**Description**
`Ok`	SD interrupt enabled successfully.
`ErrorInvalidParameter`	`pstcSd == NULL`

7.27.2.10 Sd_DisableInt ()

This function disables the interrupt flag of SD.

Prototype	
`en_result_t Sd_DisableInt(volatile stc_sd_t* pstcSd );`	
Parameter Name	**Description**
`[in] pstcSd`	Pointer to a SDIF instance.
Return Values	**Description**
`Ok`	SD interrupts disabled successfully.
`ErrorInvalidParameter`	`pstcSd == NULL`

7.27.2.11 Sd_TxData ()

This function transmits data segment for write commands only.

Prototype	
`boolean_t Sd_TxData(volatile stc_sd_t* pstcSd,stc_sd_comdconfig_t * pstcCmdCfg,uint32_t* pu32Buf);`	
Parameter Name	**Description**
[in] pstcSd	Pointer to a SDIF instance.
[in] pstcCmdCfg	Pointer to command parameters.
[in] pu32Buf	Pointer to TX buffer.
Return Values	**Description**
TRUE	Data is transmitted.
FAlSE	Error occurs during transmission.

7.27.2.12 Sd_RxData ()

This function receives the data of read command.

Prototype	
`boolean_t Sd_RxData(en_sd_transaction_t enTran,volatile stc_sd_t* pstcSd,stc_sd_comdconfig_t * pstcCmdCfg,uint32_t* pu32Buf);`	
Parameter Name	**Description**
[in] enTran	Transmission type depends on the CMD and expected response. (Not support multi block.)
[in] pstcSd	Pointer to a SDIF instance.
[in] pstcCmdCfg	Pointer to a command parameter.
[in] pu32Buf	Pointer to a RX data buffer.
Return Values	**Description**
TRUE	Data is received.
FAlSE	Error occurs during reception.

7.28 (SWWDG) Software Watchdog

Type Definition	-
Configuration Type	stc_swwdg_config_t
Address Operator	-

The SWWG driver library sets interrupt callback functions to be called, in which the user has to feed the Software Watchdog.

`Swwdg_Init()` sets interval time.
`Swwdg_Feed()` resets the Software Watchdog timer block by a function call. `Swwdg_QuickFeed()` does the same, but the code is inline expanded for time-critical polling loop.

The Software Watchdog has a timing window mode if `Swwdg_Init()` sets `bWinWdgEnable` in `stc_swwdg_config_t` to `TRUE`. The timing is set by `stc_swwdg_config_t::bu8TimingWindow`. If `stc_swwdg_config_t::bWinWdgResetEnable` is set to `TRUE`, when the counter is not reloaded within the timing window, or when the counter underflows, the Software Watchdog block generates a reset.

`Swwdg_Init()` initializes the Software Watchdog block.
`Swwdg_DeInit()` disables the Software Watchdog block.
`Swwdg_Start()` starts the timer of the Software Watchdog block.
`Swwdg_Stop()` stops the timer.
`Swwdg_WriteWdgLoad()` writes reload value for the timer.
`Swwdg_ReadWdgValue()` reads out timer.
`Swwdg_Feed()` resets the timer by a function call.
`Swwdg_QuickFeed()` works the same as `Swwdg_Feed()`, but the code is inlined for time-critical operation.

7.28.1 SWWDG Configuration Structure

The Software Watchdog driver library uses a structure of the SWWDG configuration, `stc_swwdg_config_t`:

Type	Field	Possible Values	Description
uint32_t	u32LoadValue	0x00000001 - 0xFFFFFFFF	Interval value
boolean_t	bResetEnable	TRUE FALSE	Enables Software Watchdog reset Disables Software Watchdog reset
boolean_t	bWInWdgEnable	TRUE FALSE	Enables Window Watchdog mode Disables Window Watchdog mode
boolean_t	bWinWdgResetEnable	TRUE FALSE	Enables Software Watchdog reset when reload without timing window was occurred Disables Software watchdog reset when reload without timing window was occurred
uint8_t	u8TimingWindow		Timing window settings(*)

(*)This member is used when `bWinWdgEnable` is `TRUE`.

7.28.1.1 *Timing Window Enumerators*

These enumerators are used for `uint8_t u8TimingWindow`.

Enumerator	Description
`en_swwdg_timing_window_100`	Reload can be executed at less than or equal to `WdogLoad`
`en_swwdg_timing_window_75`	Reload can be executed at less than or equal to 75% of `WdogLoad`
`en_swwdg_timing_window_50`	Reload can be executed at less than or equal to 50% of `WdogLoad`
`en_swwdg_timing_window_25`	Reload can be executed at less than or equal to 25% of `WdogLoad`

7.28.2 API reference

7.28.2.1 *Swwdg_Init()*

Initializes the Software Watchdog block.

Prototype	
`en_result_t Swwdg_Init(stc_swwdg_config_t* pstcConfig)`	
Parameter Name	**Description**
[in] `pstcConfig`	A pointer to a structure of the SWWDG configuration
Return Values	**Description**
`Ok`	Initializing the Software Watchdog block was done with no error
`ErrorInvalidParameter`	`pstcConfig == NULL` Invalid argument(s)

7.28.2.2 *Swwdg_DeInit()*

Deinitializes the Software Watchdog block.

Prototype
`void Swwdg_DeInit(void)`

7.28.2.3 *Swwdg_Start()*

Starts the timer.

Prototype	
`en_result_t Swwdg_Start(func_ptr_t pfnSwwdgCb)`	
Parameter Name	**Description**
[in] `pfnSwwdgCb`	A pointer to a callback function (Can set to NULL)
Return Values	**Description**
`Ok`	Success to start Software Watchdog
`ErrorOperationInProgress`	Software Watchdog is already started

7.28.2.4 Swwdg_Stop()

Stops the timer.

Prototype
`void Swwdg_Stop(void)`

7.28.2.5 Swwdg_WriteWdgLoad()

Writes a load value to the timer.

Prototype	
`void Swwdg_WriteWdgLoad(uint32_t u32LoadValue)`	
Parameter Name	**Description**
[in] `u32LoadValue`	A load value to the timer

7.28.2.6 Swwdg_ReadWdgValue()

Reads out a counter value.

Prototype	
`uint32_t Swwdg_ReadWdgValue(void)`	
Return Values	**Description**
`uint32_t`	A counter value

7.28.2.7 Swwdg_Feed()

Feeds the Software Watchdog block.

Prototype
`void Swwdg_Feed(void)`

7.28.2.8 Swwdg_EnableDbgBrkWdgCtl()

Keeps the counter counting while CPU halts by a debug tool.

Prototype
`void Swwdg_EnableDbgBrkWdgCtl(void)`

7.28.2.9 Swwdg_DisableDbgBrkWdgCtl()

Stops the counter while CPU halts by a debug tool. (default setting)

Prototype
`void Swwdg_DisableDbgBrkWdgCtl(void)`

7.28.2.10 *Static Inline Function*

The Software Watchdog driver library provides a feed function which is defined as static inline in the `swwdg.h` file.

Swwdg_QuickFeed()

Feeds the Software Watchdog block.

Prototype
`static __INLINE void Swwdg_Feed(void)`

7.28.2.11 *Callback function*

The callback function is registered by `Swwdg_Start()` and called in the interrupt handler when the Software Watchdog block generates an interrupt.

Prototype
`void (*func_prt_t)(void)`

7.28.3 Example Code

The example software is in *\example\wdg\swwdg*.

Folder ***\example\wdg\swwdg***	**Summary**
`swwdg_normal`	Normal usage
`swwdg_window_mode`	Window mode

7.28.3.1 SWWDG normal mode

This example software excerpt shows an usage of the Software Watchdog driver library.

```
#include "wdg/swwdg.h"

...
static const stc_swwdg_config_t stcSwwdgConfig = {
  20000000,                    // Timer interval
  TRUE,                        // Enables Software watchdog reset
  FALSE,                       // Disables Window watchdog mode
  FALSE,                       // Disables reset when reload
                               //   without timing window was occurred
  en_swwdg_timing_window_100 // Timing window setting (Not effective)
};
...

static void WdgSwCallback(void)
{
  Swwdg_Feed();   // Only for example! Do not use this in your
application!
  // some code here ...
}

function
{
  ...
  // Initialize Software watchdog
  if (Ok != Swwdg_Init((stc_swwdg_config_t *)&stcSwwdgConfig))
  {
    // some code here ...
  }
  else
  {
    // Start Software watchdog
    Swwdg_Start(WdgSwCallback);
  }

  // wait for interrupts
  while (1);
}
```

7.28.3.2 SWWDG window mode

This example software excerpt shows an usage of the Software Watchdog driver library in window mode. The window mode detects whether or not the timer is reloaded within window period. This software example reloads the timer outside or inside of the window period.

```
#include "wdg/swwdg.h"

...
static const stc_swwdg_config_t stcSwwdgConfig = {
  20000000,                   // Timer interval
  TRUE,                       // Enables Software watchdog reset
  TRUE,                       // Enables Window watchdog mode
  FALSE,                      // Disables reset when reload
                              // without timing window was occurred
  en_swwdg_timing_window_50  // Reload can be executed at less than
                              // or equal to 50% of WdogLoad
};
...
```

```
static void WdgSwCallback(void)
{
  Swwdg_Stop();           // Only for example! Do not use this in your
                             application!
  Swwdg_Feed();           // Only for example! Do not use this in your
                             application!
  u8SwwdtActive = FALSE; // Stop Software watchdog
}

function
{
  ...
  // Initialize Software watchdog
  if (Ok != Swwdg_Init((stc_swwdg_config_t *)&stcSwwdgConfig))
  {
    // some code here ...
    while (1);
  }
  ...
  // Only for example below! Do not use this in your application!
  while (10 > u8Index)
  {
    // Restart software watchdog
    Swwdg_Start(WdgSwCallback);
    u8SwwdtActive = TRUE;
    // Adjust the timing for reloading watchdog counter
    u32Count = au32CountValue[u8Index];
    while (0 != (u32Count--))
    {
      continue;
    }
    __disable_irq();
    // Clear interrupt and reload watchdog counter
    Swwdg_Feed();
    __enable_irq();
    // Insert cycle for interrupt
    PDL_WAIT_LOOP_HOOK();
    // If watchdog is in-active, error is occured.
    if (FALSE == u8SwwdtActive)
    {
      // Error status set;
      au8ErrorStatus[u8Index] = 1;
    }
    u8Index++;
    __disable_irq();
    // If watchdog is active...
    if (TRUE == u8SwwdtActive)
    {
      // Stop software watchdog and clear interrupt
      WdgSwCallback();
    }
    __enable_irq();
  }

  while (1);
}
```

7.29 (UID) Unique ID

Type Definition	–
Configuration Type	–
Address Operator	–

This module provides access to the Unique ID register. This driver does not need any configuration.

7.29.1 UID API

The following API functions are used for handling the Unique ID contents.

7.29.1.1 Uid_ReadUniqueId()

This function reads out UIDR0 and UIDR1 as is without any shift to a pointered structure of the type `stc_unique_id_t`. Reserved bits are masked to '`0`'.

Prototype	
`en_result_t Uid_ReadUniqueId( stc_unique_id_t* pstcUniqueId );`	
Parameter Name	**Description**
`[out] pstcUniqueId`	Pointer to Unique ID structure
Return Values	**Description**
`Ok`	Successfully stored
`ErrorInvalidParameter`	`pstcUniqueId == NULL`

Structure of the type of `stc_unique_id_t`.

Element Type	Element Name
`uint32_t`	`u32Uidr0`
`uint32_t`	`u32Uidr1`

7.29.1.2 Uid_ReadUniqueId0 ()

This function reads out UIDR0 and right-shifts the content by 4 (LSB alignment).

Prototype	
`uint32_t Uid_ReadUniqueId0( void );`	
Return Values	**Description**
`uint32_t`	`UIDR0 >> 4`

7.29.1.3 *Uid_ReadUniqueId1 ()*

This function reads out UIDR0 and masks the upper 19 bits to '0'.

Prototype	
`uint32_t Uid_ReadUniqueId1( void );`	
Return Values	**Description**
`uint32_t`	`UIDR1 & 0x00001FFF`

7.29.1.4 *Uid_ReadUniqueId64()*

This function reads Unique ID registers 0 and 1 and merge it LSB aligned to a 64-bit value.

Prototype	
`uint64_t Uid_ReadUniqueId64( void );`	
Return Values	**Description**
`Uint64_t`	`UIDR1` and `UIDR0` aligned to LSB.

7.29.2 UID Example

The PDL example folder contains a UID usage example:

- uid_read All API methods to read out the Unique ID.

7.30 (WC) Watch Counter

Type Definition	-

Configuration Types	`stc_wc_pres_clk_t`
	`stc_wc_config_t`
	`stc_wc_intern_data_t`
	`stc_wc_instance_data_t`

Address Operator	-

Before using WC, WC prescaler must be configured first. Use Wc_Pres_SelClk()to select input clock of prescaler. Following clocks can be selected:

- Sub clock
- Main clock
- High-speed CR
- CLKLC (divided by low speed CR)

Wc_Pres_EnableDiv() is used to enable watch counter prescaler.

Wc_Pres_DisableDiv() is used to disable watch counter prescaler.

Wc_Init() must be used for configuration of watch counter with a structure of the type stc_wc_config_t.

A WC interrupt can be enabled by the function Wc_EnableInt().This function can set callback function for each channel too.

With Wc_WriteReloadVal() the WC reloader value is set to the value given in the parameter Wc_WriteReloadVal#u8Val.

After above setting, calling Wc_EnableCount() will start WC.

With Wc_ReadCurCnt() the current WC count can be read when WC is counting. with Wc_GetOperationFlag() the current WC operaiton status can be read.

With interrupt mode, when the interrupt occurs, the interrupt flag will be cleared and run into user interrupt callback function.

With polling mode, user can use Wc_GetIntFlag() to check if the interrupt occurs, and clear the interrupt flag by Wc_ClrIntFlag().

When stopping the WC, use Wc_DisableCount() to disable WC and Wc_DisableInt() to disable WC interrupt.

7.30.1 Configuration Structure

A counter prescaler instance uses the following configuration structure of the type of `stc_wc_pres_clk_t`:

Type	Field	Possible Values	Description

`en_input_clk_t`	`enInputClk`	`WcPresInClkSubOsc`
		`WcPresInClkMainOsc`
		`WcPresInClkHighCr`
		`WcPresInClkLowCr`
			Select the input source clock of watch counter precaler:
			sub oscillator
			main oscillator
			high-speed CR
			low-speed CR

`en_output_clk_t`	`enOutputClk`	`WcPresOutClkArray0`
		`WcPresOutClkArray1`
		`WcPresOutClkArray2`
		`WcPresOutClkArray3`
		`WcPresOutClkArray4`
		`WcPresOutClkArray5`
		`WcPresOutClkArray6`
			Select Watch counter prescaler output setting:
			(see define section in *wc.h*, en_output_clk_t for detail value settings)

A watch counter configure instance uses the following configuration structure of the type of stc_wc_config_t:

Type	Field	Possible Values	Description

`en_wc_cnt_clk_t`	`enCntClk`	`WcCntClkWcck0`
		`WcCntClkWcck1`
		`WcCntClkWcck2`
		`WcCntClkWcck3`
			Select the clock source of watch counter from watch counter prescaler:
			WCCK0
			WCCK1
			WCCK2
			WCCK3

A watch counter internal data instance uses the following configuration structure of the type of stc_wc_intern_data_t:

Type	Field	Possible Values	Description
`func_ptr_t`	`pfnIntCallback`	-	Pointer to a watch counter interrupt callback function.

A watch counter instance data instance uses the following configuration structure of the type of stc_wc_instance_data_t.

Type	Field	Possible Values	Description
stc_wcn_t*	pstcInstance	-	pointer to registers of an instance.
stc_wc_intern_data_t	stcInternData	-	See definition above.

7.30.2 WC API

7.30.2.1 Wc_Pres_SelClk ()

This function selects the input clock and set the division clock to be output.

Prototype	
en_result_t Wc_Pres_SelClk(volatile stc_wcn_t* pstcWc, stc_wc_pres_clk_t* pstcWcPresClk);	
Parameter Name	**Description**
[in] pstcWc	Pointer to WC instance.
[in] pstcWcPresClk	Pointer to WC prescaler clock configuration.
Return Values	**Description**
Ok	Set WC prescaler successfully.
ErrorInvalidParameter	pstcBt == NULL

7.30.2.2 Wc_Pres_EnableDiv ()

This function enables oscilation of the divison clock.

Prototype	
en_result_t Wc_Pres_EnableDiv(volatile stc_wcn_t* pstcWc);	
Parameter Name	**Description**
[in] pstcWc	Pointer to WC instance.
Return Values	**Description**
Ok	Enable oscilation of the divison clock successfully.
ErrorInvalidParameter	pstcBt == NULL

7.30.2.3 Wc_Pres_DisableDiv ()

This function disables oscilation of the divison clock.

Prototype	
`en_result_t Wc_Pres_DisableDiv(volatile stc_wcn_t* pstcWc);`	
Parameter Name	**Description**
`[in] pstcWc`	Pointer to WC instance.
Return Values	**Description**
`Ok`	Disable oscilation of the divison clock successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.30.2.4 Wc_GetDivStat ()

This function gets the operation status of the division counter.

Prototype	
`en_stat_flag_t Wc_GetDivStat(volatile stc_wcn_t* pstcWc);`	
Parameter Name	**Description**
`[in] pstcWc`	Pointer to WC instance.
Return Values	**Description**
`PdlClr`	Oscilation of the division clock is not performed.
`PdlSet`	Oscilation of the division clock is performed.

7.30.2.5 Wc_Init ()

This function initializes a WC by select the input clock and set the division clock to be output.

Prototype	
`en_result_t Wc_Init(volatile stc_wcn_t* pstcWc, stc_wc_config_t* pstcWcConfig);`	
Parameter Name	**Description**
`[in] pstcWc`	Pointer to WC instance.
`[in] pstcWcConfig`	WC prescaler clock configuration.
Return Values	**Description**
`Ok`	Init WC done.

`ErrorInvalidParameter`	`pstcBt == NULL`
	`pstcWcConfig == NULL`
	`cant get internal data`

7.30.2.6 Wc_EnableCount ()

This function enables operation of WC.

Prototype	
`en_result_t Wc_EnableCount(volatile stc_wcn_t* pstcWc);`	
Parameter Name	**Description**
`[in] pstcWc`	Pointer to WC instance.
Return Values	**Description**
`Ok`	Enable WC operation done.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.30.2.7 Wc_DisableCount ()

This function disables operation of WC.

Prototype	
`en_result_t Wc_DisableCount(volatile stc_wcn_t* pstcWc);`	
Parameter Name	**Description**
`[in] pstcWc`	Pointer to WC instance.
Return Values	**Description**
`Ok`	Disable WC operation done.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.30.2.8 Wc_EnableInt ()

This function enables WC underflow interrupt.

Prototype	
`en_result_t Wc_EnableInt(volatile stc_wcn_t* pstcWc, func_ptr_t  pfnIntCallback);`	
Parameter Name	**Description**
`[in] pstcWc`	Pointer to WC instance.
`[in] pfnIntCallback`	Pointer to WC interrupt callback functions.
Return Values	**Description**
`Ok`	WC interrupt enabled successfully.
`ErrorInvalidParameter`	`pstcBt == NULL` can not get internal data.

7.30.2.9 Wc_DisableInt ()

This function disables WC underflow interruption.

Prototype	
`en_result_t Wc_DisableInt(volatile stc_wcn_t* pstcWc);`	
Parameter Name	**Description**
`[in] pstcWc`	Pointer to WC instance.
Return Values	**Description**
`Ok`	WC interruption disabled successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.30.2.10 Wc_WriteReloadVal ()

This function sets the reload value of WC for the 6-bit down counter.

Prototype	
`en_result_t Wc_WriteReloadVal(volatile stc_wcn_t* pstcWc, uint8_t u8Val);`	
Parameter Name	**Description**
`[in] pstcWc`	Pointer to WC instance.
`[in] u8Val`	Reload value.
Return Values	**Description**
`Ok`	Write data successfully done.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.30.2.11 Wc_ReadCurCnt ()

This function reads the value in the 6-bit down counter.

Prototype	
`uint8_t Wc_ReadCurCnt(volatile stc_wcn_t* pstcWc);`	
Parameter Name	**Description**
`[in] pstcWc`	Pointer to WC instance.
Return Values	**Description**
`0xFF`	`pstcWc == NULL`
`value`	The value of WC 6-bit down counter.

7.30.2.12 Wc_ClearIntFlag ()

This function clears WC underflow flag.

Prototype	
`en_result_t Wc_ClearIntFlag(volatile stc_wcn_t* pstcWc);`	
Parameter Name	**Description**
`[in] pstcWc`	Pointer to WC instance.
Return Values	**Description**
`Ok`	Clear WC underflow flag successfully.
`ErrorInvalidParameter`	`pstcBt == NULL`

7.30.2.13 Wc_GetIntFlag ()

This function gets WC underflow flag status.

Prototype	
`en_int_flag_t Wc_GetIntFlag(volatile stc_wcn_t* pstcWc);`	
Parameter Name	**Description**
`[in] pstcWc`	Pointer to WC instance.
Return Values	**Description**
`PdlClr`	WC underflow does not occur.
`PdlSet`	WC underflow occurs.

7.30.2.14 Wc_GetOperationFlag ()

This function gets WC operation state.

Prototype	
`en_stat_flag_t Wc_GetOperationFlag(volatile stc_wcn_t* pstcWc);`	
Parameter Name	**Description**
`[in] pstcWc`	Pointer to WC instance.
Return Values	**Description**
`PdlClr`	WC is stopped.
`PdlSet`	WC is active.

7.30.2.15 Wc_IrqHandler ()

This function is WC interrupt callback function.

Prototype	
`void Wc_IrqHandler( volatile stc_wcn_t* pstcWc,stc_wc_intern_data_t* pstcWcInternData);`	
Parameter Name	**Description**
[in] pstcWc	Pointer to WC instance.
[in] pstcWcInternData	Pointer to WC callback functions.
Return Values	**Description**
-	

8. Revision History

Document Revision History

Document Title: FM4 Peripheral Driver Library, User Guide Document Number: 002-04460			
Revision	**Issue Date**	**Origin of Change**	**Description of Change**
**	09/11/2015	YUIS	Rev 1.0 Initial Release Rev 1.1 Company name and logo change Chapter number correction Typo correction
*A	01/05/2016	YUIS	Migrated Spansion Guide from FM4_AN709-00003-1v1-E to Cypress format