Spansion® Analog and Microcontroller Products

Colophon
The products described in this document are designed, developed and manufactured as contemplated for general use, including without limitation, ordinary industrial use, general office use, personal use, and household use, but are not designed, developed and manufactured as contemplated (1) for any use that includes fatal risks or dangers that, unless extremely high safety is secured, could have a serious effect to the public, and could lead directly to death, personal injury, severe physical damage or other loss (i.e., nuclear reaction control in nuclear facility, aircraft flight control, air traffic control, mass transport control, medical life support system, missile launch control in weapon system), or (2) for any use where chance of failure is intolerable (i.e., submersible repeater and artificial satellite). Please note that Spansion will not be liable to you and/or any third party for any claims or damages arising in connection with above-mentioned uses of the products. Any semiconductor devices have an inherent chance of failure. You must protect against injury, damage or loss from such failures by incorporating safety design measures into your facility and equipment such as redundancy, fire protection, and prevention of over-current levels and other abnormal operating conditions. If any products described in this document represent goods or technologies subject to certain restrictions on export under the Foreign Exchange and Foreign Trade Law of Japan, the US Export Administration Regulations or the applicable laws of any other country, the prior authorization by the respective government entity will be required for export of those products.

Fujitsu Semiconductor Europe
Application Note

MCU-AN-300412-E-V16

FM3 FAMILY

32-BIT MICROCONTROLLER

MB9A/BFXXX

LOW LEVEL LIBRARY MANUAL

APPLICATION NOTE

Revision History

Date	Issue
2012-10-23	V1.0; MWi; 1st Version
2012-12-14	V1.1; MWi; Known restrictions chapter added, New WDG and NAND Flash functions introduced, notes on System_Init() given, USB application notes reference added, EMAC and ETHPHY note added
2013-02-07	V1.2; MWi; Peripheral activation caution notes added; supported instances table added; Work Flash now determined by L3 itself
2013-02-18	V1.3; MWi; Flash Timing module description added; Dual Operation and ECC Flash now determined by L3 itself: All Flash device types now supported; Ethernet added to project set-up
2013-03-28	V1.4; `L3_ZERO_STRUCT` introduced and explained
2013-07-18	V1.5; Misleading CAN configuration example code insertion added; `stc_extif_area_config_t` corrected, `Dma_Enable()` name corrected; `Exint_ClearChannel()` added; `bTouchPrescaler` in can configuration added; CAN `EOB` bit functions added; Base Timer API expanded
2013-10-21	V1.6; Typos corrected; GPIO chapter corrections; Explanation for Mfs_ReadChar() added; More detailed explanation of priority conversion for ADC added

This document contains 265 pages.

Warranty and Disclaimer

Contents

1 Introduction

The *Low Level Library* (called *LLL, L3* or *L*^{3}) for FM3 MCUs is a driver collection, which covers almost all functions of the internal resources, such as timers, communication interfaces, etc. The style of this library is based on the L3 of the Fujitsu’s CR4 devices.

This application note describes the architecture of the FM3 *Low Level Library* and explains how to use it.

Recently L3 is available in version 1.9.

2 L3 Structure

THIS CHAPTER EXPLAINS THE STRUCTURE OF THE LLL

2.1 File System

The LLL (named *L3* from now on) consists of the following files in its *l3* directory:

• *l3.c*	Common (helper) function collection
• *l3.h*	Common definitions of all resources, which are implemented, and other settings
• *l3_user.h*	User definitions for L3 drivers
• *interrupts.c*	ISR calls implementation
• *interrupts.h*	ISR prototypes (device dependent)
• *ramcode.c*	Functions which are linked and copied to RAM
• *ramcode.h*	Corresponding *ramcode.c* header file.
• *base_types_l3.h*	Global L3 definitions and types (e.g. `TRUE`, `FALSE`, `en_result_t`, etc.)
• *resourceabbreviation1*.c	Driver functions of resource #1
• *resourceabbreviation1*.h	Configuration structures and API function prototypes of drivers of resource #1
• *resourceabbreviation2*.c	*see above*
• *resourceabbreviation2*.h	*see above*
• . . .	

For green files user settings are needed, blue files may be edited by the user, and orange files should *not* be changed by the user.

As can be seen above the library has 8 common used files, *l3.c*, *l3.h*, *l3_user.h*, *interrupts.c*, *interrupts.h*, *ramcode.c*, *ramcode.h*, and *base_types_l3.h*.

For a certain resource's driver collection 2 files are needed, *resourceabbreviation*.c, and *resourceabbreviation*.h, where *resourceabbreviation* is the abbreviation of the resource, such as *`dt`* for Dual Timer, *`mfs`* for Multi Function Serial, etc.

2.2 Characteristics

The L3 functions for the resources have the following characteristics:

- Each function of a resource driver is existing only once in the ROM (Flash) memory regardless of the number of resource instances. It uses a so-called handle to the instance.
- Each function of the driver uses a configuration data structure per instance, which has to be defined in the user application.
- Most API (external) functions have a return value (at least just `en_result_t Ok`).
- Each resource driver provides an Init and a De-Init function
- If the resource driver uses interrupts, the interrupt service routine/s is/are 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 is being a member of the project workspace.

2.3 Graphical Example

The following graphic illustrates a driver for resource A (*Ra*) which has 2 active instances.

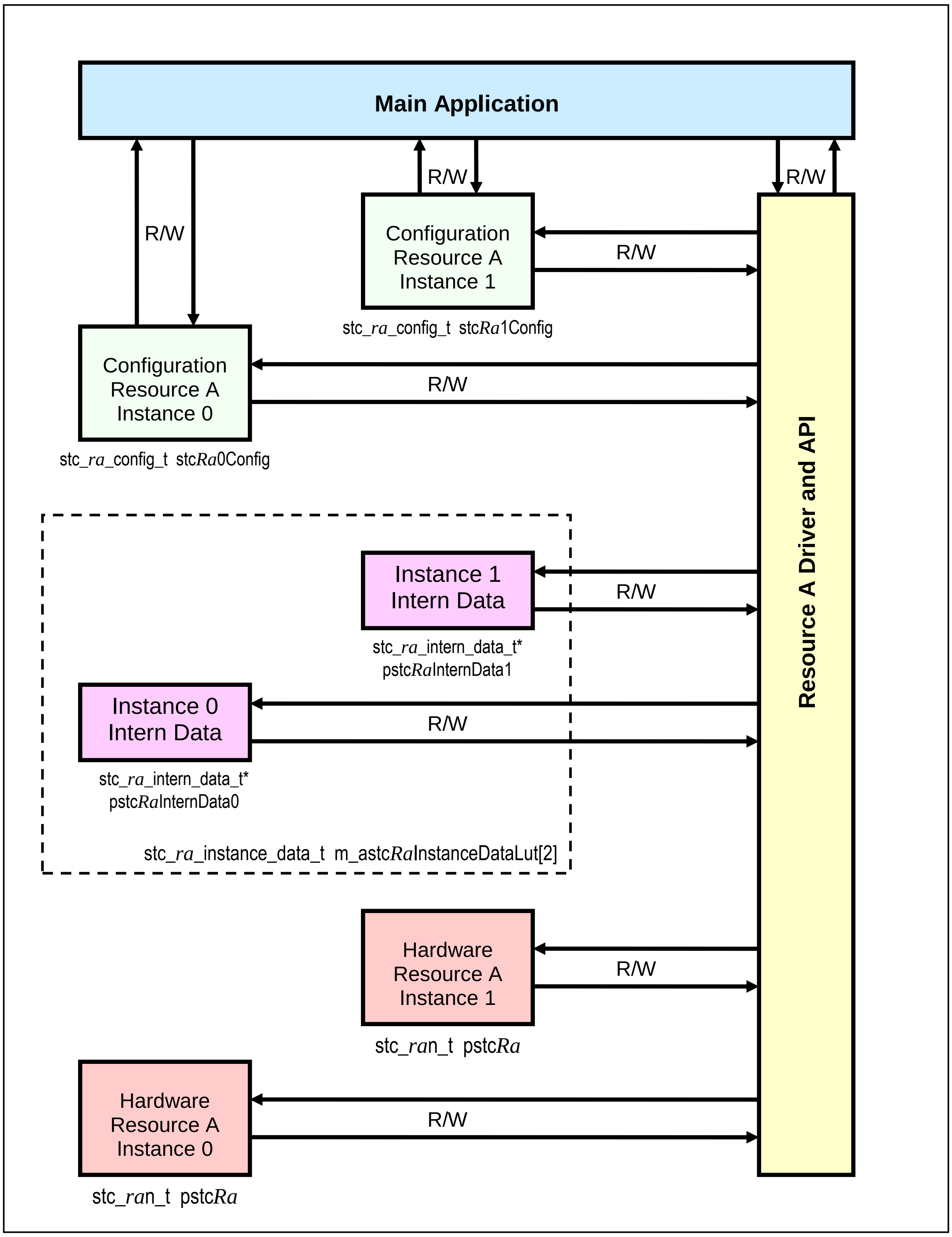

Main Application
R/W
R/W
R/W
Configuration Resource A Instance 1
R/W
stc_ra_config_t stcRa1Config
Configuration Resource A Instance 0
R/W
stc_ra_config_t stcRa0Config
Resource A Driver and API
Instance 1 Intern Data
R/W
stc_ra_intern_data_t* pstcRaInternData1
Instance 0 Intern Data
R/W
stc_ra_intern_data_t* 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

Figure 2-1: Resource Driver Structure Example

2.4 Interrupt Flow

The following flow shows how interrupts are handled in the L3:

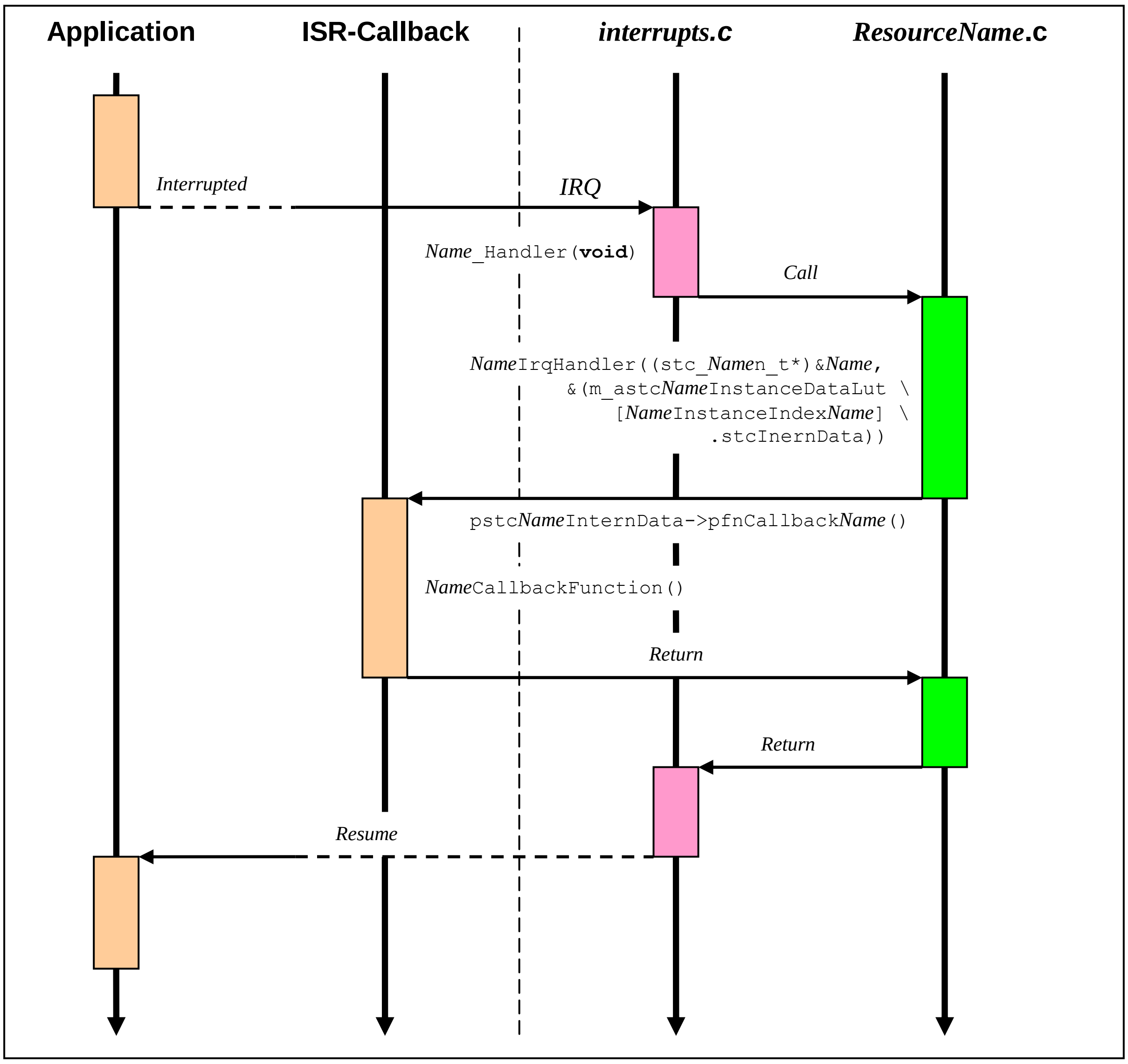


Figure 2-2: Interrupt and Callback Flow Diagram

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

The L3 also controls the shared vector interrupts and always executes the actual interrupt service routines.

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

3 Using L3 in own Application Projects

HOW TO INCLUDE THE L3 TO OWN PROJECTS

3.1 Use Standard Template as Base

The first step is to copy the *l3* directory to the desired FM3 type template's *source* directory.

The next step is to announce this new directory and all the **.c* modules to the workspace of the used tool chain. Therefore a new virtual folder *l3* may be created.

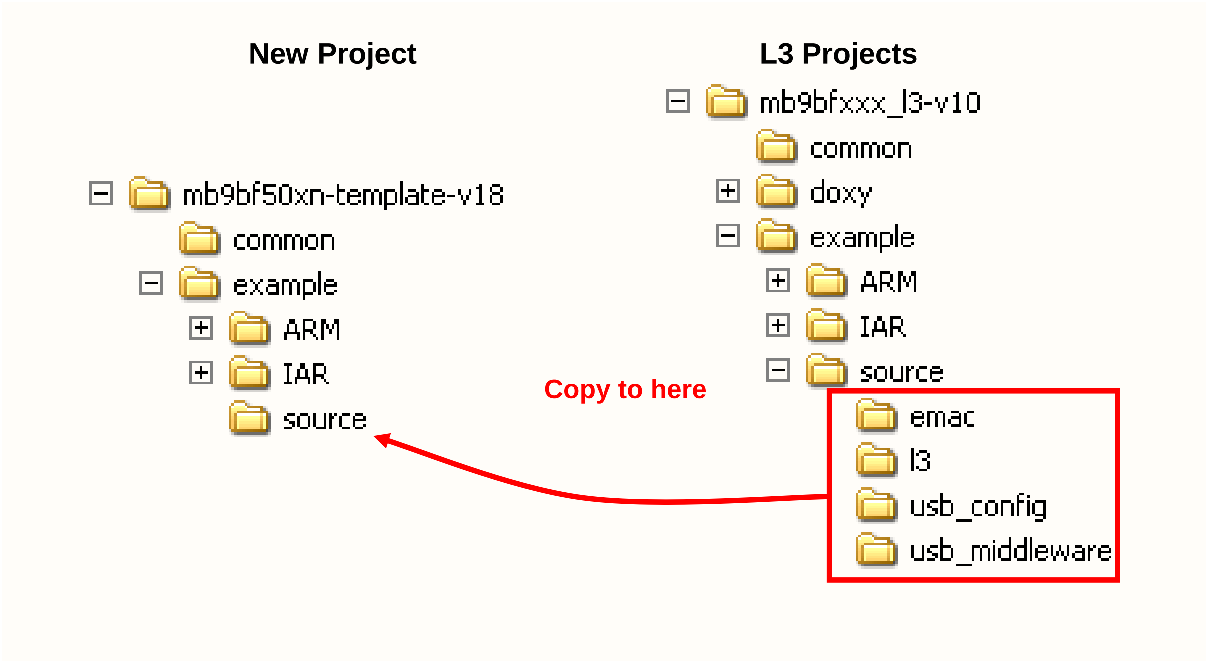


Figure 3-1: Location for L3 modules

For the different parts of the LLL make the following subfolders or subgroups in your preferred IDE:

• l3	Containing all resource drivers and headers
• emac	Containing all Ethernet parts
• usb_config	Containing UsbConfig.c/.h
• usb_middleware	Containing all middleware USB parts

Adjust the preprocessor search paths to:

- <Project start directory>\..\..*common*
- <Project start directory>\..\..*example/source*
- <Project start directory>\..\..*example/source/emac*
- <Project start directory>\..\..*example/source/l3*
- <Project start directory>\..\..*example/source/usb_middleware*
- <Project start directory>\..\..*example/source/usb_config*

Copy also the file *base_types.h* from the *common* directory to the new project's *common* directory.

The additional paths in IAR Workbench notation:

$PROJ_DIR$\..\..\example\source\emac

$PROJ_DIR$\..\..\example\source\l3

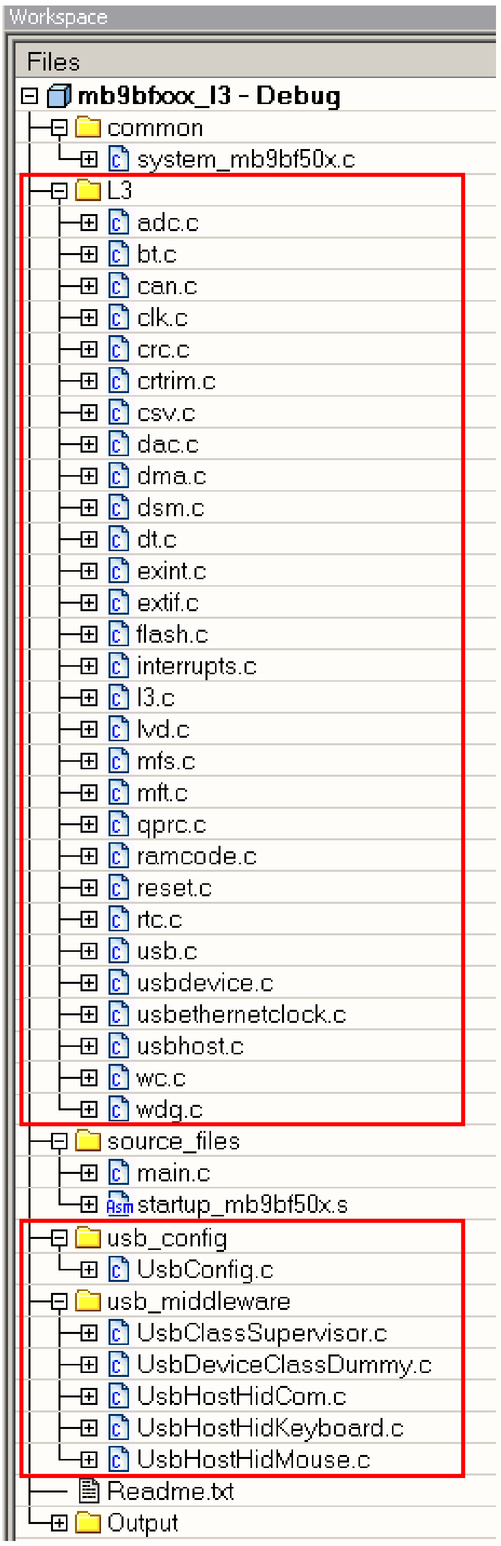
$PROJ_DIR$\..\..\example\source\usb_middleware

$PROJ_DIR$\..\..\example\source\usb_config

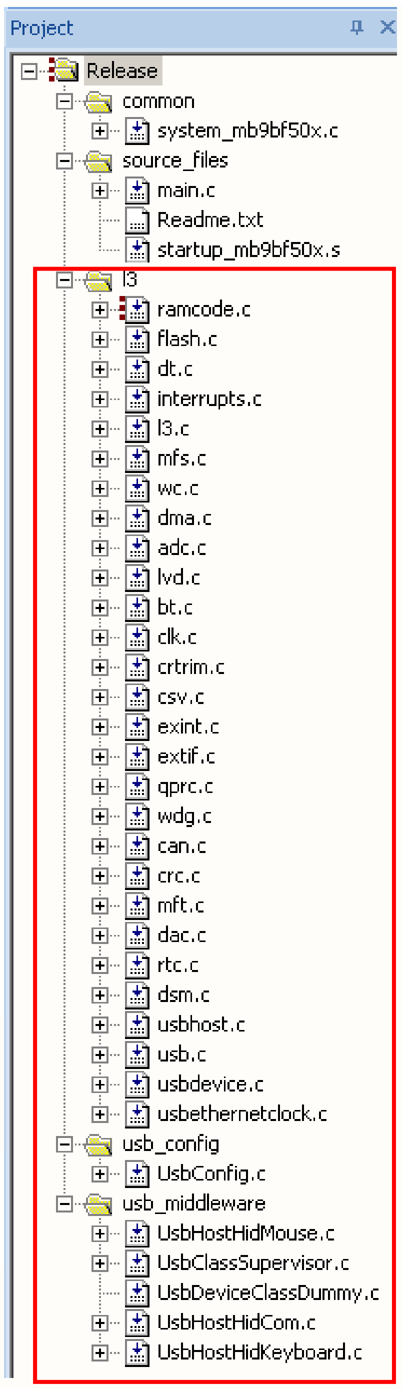
The additional paths in KEIL µVision:

../../example/source/l3;../../example/source/usb_config;../../example/source/usb_middleware;../../example/source/emac

The workspace should get the following groups as shown in the following graphic, which show IAR Workbench and KEIL's µVision screenshots. This must be done by hand by adding all *.c files to the workspace. For easy overview a group named *L3* may be created before.



```
Workspace
Files
⊟ mb9bfxxx_l3 - Debug
  ├⊟ common
  │ └⊞ system_mb9bf50x.c
  ├⊟ L3
  │ ├⊞ adc.c
  │ ├⊞ bt.c
  │ ├⊞ can.c
  │ ├⊞ clk.c
  │ ├⊞ crc.c
  │ ├⊞ crtrim.c
  │ ├⊞ csv.c
  │ ├⊞ dac.c
  │ ├⊞ dma.c
  │ ├⊞ dsm.c
  │ ├⊞ dt.c
  │ ├⊞ exint.c
  │ ├⊞ extif.c
  │ ├⊞ flash.c
  │ ├⊞ interrupts.c
  │ ├⊞ l3.c
  │ ├⊞ lvd.c
  │ ├⊞ mfs.c
  │ ├⊞ mft.c
  │ ├⊞ qprc.c
  │ ├⊞ ramcode.c
  │ ├⊞ reset.c
  │ ├⊞ rtc.c
  │ ├⊞ usb.c
  │ ├⊞ usbdevice.c
  │ ├⊞ usbethernetclock.c
  │ ├⊞ usbhost.c
  │ ├⊞ wc.c
  │ └⊞ wdg.c
  ├⊟ source_files
  │ ├⊞ main.c
  │ └⊞ startup_mb9bf50x.s
  ├⊟ usb_config
  │ └⊞ UsbConfig.c
  ├⊟ usb_middleware
  │ ├⊞ UsbClassSupervisor.c
  │ ├⊞ UsbDeviceClassDummy.c
  │ ├⊞ UsbHostHidCom.c
  │ ├⊞ UsbHostHidKeyboard.c
  │ └⊞ UsbHostHidMouse.c
  ├── Readme.txt
  └⊞ Output
```



```
Project
⊟ Release
  ⊟ common
    ⊞ system_mb9bf50x.c
  ⊟ source_files
    ⊞ main.c
      Readme.txt
      startup_mb9bf50x.s
  ⊟ l3
    ⊞ ramcode.c
    ⊞ flash.c
    ⊞ dt.c
    ⊞ interrupts.c
    ⊞ l3.c
    ⊞ mfs.c
    ⊞ wc.c
    ⊞ dma.c
    ⊞ adc.c
    ⊞ lvd.c
    ⊞ bt.c
    ⊞ clk.c
    ⊞ crtrim.c
    ⊞ csv.c
    ⊞ exint.c
    ⊞ extif.c
    ⊞ qprc.c
    ⊞ wdg.c
    ⊞ can.c
    ⊞ crc.c
    ⊞ mft.c
    ⊞ dac.c
    ⊞ rtc.c
    ⊞ dsm.c
    ⊞ usbhost.c
    ⊞ usb.c
    ⊞ usbdevice.c
    ⊞ usbethernetclock.c
  ⊟ usb_config
    ⊞ UsbConfig.c
  ⊟ usb_middleware
    ⊞ UsbHostHidMouse.c
    ⊞ UsbClassSupervisor.c
      UsbDeviceClassDummy.c
    ⊞ UsbHostHidCom.c
    ⊞ UsbHostHidKeyboard.c
```

Figure 3-2: Example of L3 Directory in Workspaces
(Note that not all files are listed here for page space reasons)

Attention:

In some cases the used IDE loses the device configuration. In this case adjust the used MCU in the projects settings!

3.2 Adjust L3 to used FM3 Device

For announcing the used FM3 derivative an adjustment has to be done.

Set the device name in *l3_user.h* described in chapter 4.1.1. The exact definition name can be found in *l3.h* described in chapter 4.3.

3.2.1 Possible Compile Error

To keep the device list short, not all possible device and their types are defined. If you get a compile error like

```
Device Type not defined!
```

then go to the source code of *l3.h* search for this line, check in the FM3 documentation which device type your device is and put this type manually to the definition. Example of occurrence

```
#if (L3_DEVICE_MB9AF10X == L3_ON)
  #if   (L3_PACKAGE == L3_DEVICE_PACKAGE_N) || \
        (L3_PACKAGE == L3_DEVICE_PACKAGE_R)
    #define L3_DEVICE_TYPE L3_TYPE0
  #else
    #error Device Type not defined!
  #endif
  . . .
```

3.3 Enable Peripherals

For using certain drivers enable the peripherals in *l3_user.h* as described in 4.4.2 and 4.4.4. This adjustment is mandatory, because it controls whether a driver module has to be compiled or not.

Carefully check the activation before usage, whether your device supports a peripheral and especially a certain instance!

3.4 Setting Interrupt Levels

Finally define the interrupt levels as described in 4.4.5.

3.5 Combination of Drivers

Some settings of the resource configuration requires to activate an additional driver module such as external interrupts or DMA usage. In this case the user has to do this manually in *l3_user.h* and has to define the configuration and has to take care of correct initialization of all drivers in his application.

3.5.1 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.

4 L3 Main Files

DESCRIPTIONS OF THE MAIN L3 MODULES

4.1 File: base_types_l3.h

In the *base_types_l3.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.

4.1.1 Type Definitions

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

4.1.2 L3 Return Value Type (en_result_t)

In the *base_types_l3.h* file the L3 global return or result enumeration is done. For all L3 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:

```
/** generic error codes */
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 op. 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 buf. cannot be written 'cuz buf. is full
  ErrorTimeout                = 9,  ///< Time Out error occurred (e.g. I2C arbitration lost,
                                    ///< Flash time-out, etc.)
  ErrorNotReady               = 10, ///< A requested final state is not reached
  OperationInProgress         = 11  ///< Indicator for operation in progress (e.g. ADC
                                    ///< conversion not finished, DMA channel used, etc.)
} en_result_t;
```

4.2 File: l3.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.

```
/**
 ******************************************************************************
 ** \brief Hook function, which is called in polling loops
 **
 ** This functionality has to be mirrored in ramcode.c, if RAM code is used!
 ******************************************************************************/
void L3_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 (see chapter 5.23):

`static void L3_RAMCODE_WAIT_LOOP_HOOK(void).`

4.3 File: l3.h

First the *l3.h* file defines the device type, the FM3 device names, and package code. This file should not be changed by the user.

```
/**
 ******************************************************************************
 ** All definitions needed for l3_user.h are stated here
 ******************************************************************************/

#define L3_ON   1    ///< Switches a feature on.
#define L3_OFF 0     ///< Switches a feature off.

#define L3_TYPE0 0  ///< FM3 device type0
#define L3_TYPE1 1  ///< FM3 device type1
#define L3_TYPE2 2  ///< FM3 device type2
#define L3_TYPE3 3  ///< FM3 device type3
#define L3_TYPE4 4  ///< FM3 device type4
#define L3_TYPE5 5  ///< FM3 device type5
#define L3_TYPE5 6  ///< FM3 device type6
#define L3_TYPE5 7  ///< FM3 device type7

/**
 ******************************************************************************
 ** Global User Device Choice List
 ******************************************************************************/
#define L3_DEVICE_TYPE_MB9AF10X   0
#define L3_DEVICE_TYPE_MB9AF11X  10
#define L3_DEVICE_TYPE_MB9AF13X  20
#define L3_DEVICE_TYPE_MB9AF31X  30
#define L3_DEVICE_TYPE_MB9BF10X  40
#define L3_DEVICE_TYPE_MB9BF20X  50
#define L3_DEVICE_TYPE_MB9BF30X  60

       . . .

#define L3_DEVICE_TYPE_MB9BF31X 130
#define L3_DEVICE_TYPE_MB9BF41X 140
#define L3_DEVICE_TYPE_MB9BF51X 150
#define L3_DEVICE_TYPE_MB9BF61X 160
#define L3_DEVICE_TYPE_MB9BFD1X 170

/**
 ******************************************************************************
 ** Global User Package Choice List
 ******************************************************************************/
#define L3_DEVICE_PACKAGE_K  10
#define L3_DEVICE_PACKAGE_L  20
#define L3_DEVICE_PACKAGE_M  30
#define L3_DEVICE_PACKAGE_N  40
#define L3_DEVICE_PACKAGE_R  50
#define L3_DEVICE_PACKAGE_S  60
#define L3_DEVICE_PACKAGE_T  70
```

After this the user setting *l3_user.h* file is included.

```
/******************************************************************************/
/* User Setting Include file                                                  */
/******************************************************************************/
#include "l3_user.h"  // MUST be included here!
```

Then the user's device type definition sets the device definition itself.

```
/**
 ******************************************************************************
 ** Global device family definition
 ******************************************************************************/
#if (L3_MCU == L3_DEVICE_TYPE_MB9AF10X)
  #define L3_DEVICE_MB9AF10X L3_ON
#else
  #define L3_DEVICE_MB9AF10X L3_OFF
#endif

#if (L3_MCU == L3_DEVICE_TYPE_MB9AF11X)
  #define L3_DEVICE_MB9AF11X L3_ON
#else
  #define L3_DEVICE_MB9AF11X L3_OFF
#endif
              . . .
```

The device type is retrieved from the device definition and the user package definition.

```
/**
 ******************************************************************************
 ** Global device type definition
 ******************************************************************************/
#if (L3_DEVICE_MB9AF10X == L3_ON)
  #if    (L3_PACKAGE == L3_DEVICE_PACKAGE_N) || \
         (L3_PACKAGE == L3_DEVICE_PACKAGE_R)
    #define L3_DEVICE_TYPE L3_TYPE0
  #else
    #error Device Type not defined!
  #endif

#elif (L3_DEVICE_MB9AF11X == L3_ON)
  #if    (L3_PACKAGE == L3_DEVICE_PACKAGE_L) || \
         (L3_PACKAGE == L3_DEVICE_PACKAGE_M) || \
         (L3_PACKAGE == L3_DEVICE_PACKAGE_N)
    #define L3_DEVICE_TYPE L3_TYPE1
  #elif (L3_PACKAGE == L3_DEVICE_PACKAGE_K)
    #define L3_DEVICE_TYPE L3_TYPE5
  #else
    #error Device Type not defined!
  #endif
              . . .
```

Finally the user activated resources are defined for the compilation process.

```
/**
 ******************************************************************************
 ** L3 resource enable check
 **
 ** \note It does not check, if a device has actually any in l3.h enabled
 **       resource and all instances available!
 **
 ******************************************************************************/
// Activate code for dma.c
#if L3_PERIPHERAL_ENABLE_DMA == L3_ON
    #define L3_PERIPHERAL_DMA_ACTIVE
#endif
              . . .
```

4.3.1 L3_ZERO_STRUCT()

Some compilers (e.g GCC) do not accept a structure zero-initialization like:

```
stc_somestructtype_t stcSomeStructure = {0};
```

IAR and KEIL compilers throw a warning, if the first structure element is an enume type, but those compilers automatically include the library function `__eabi_memclr4()` to clear the structure.

For this reason `L3_ZERO_STRUCT()` was introduced in *l3.h* as a macro.

```
#define L3_ZERO_STRUCT(x) l3_memclr((uint8_t*)&(x), (uint32_t)(sizeof(x)))
```

The function `l3_memclr()` is defined in *l3.c*:

```
void l3_memclr(uint8_t* pu32Address, uint32_t u32Count)
{
    while(u32Count--)
    {
        *pu32Address++ = 0;
    }
}
```

The zero-initialization then is done like:

```
L3_ZERO_STRUCT(stcSomeStructure);
```

4.4 File: l3_user.h

This file is the most important file for user settings of the L3. It controls every activation of the resource drivers and its instances.

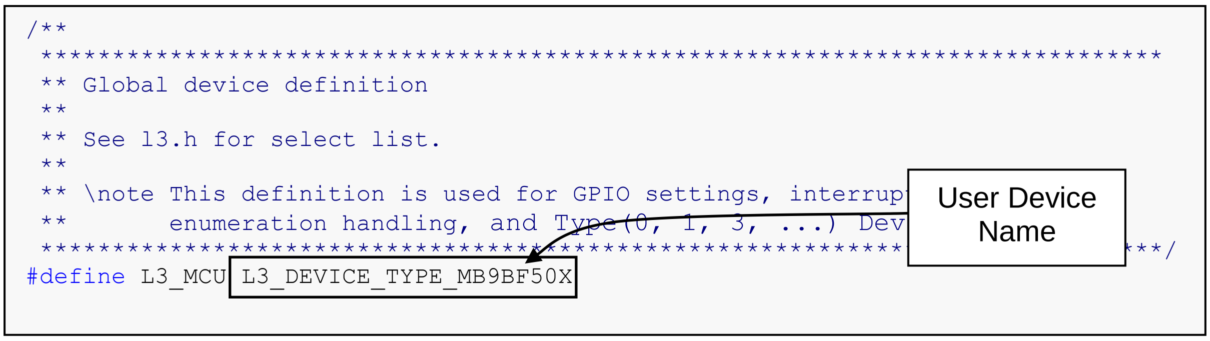
Be aware that these activations control the compile process via the preprocessor. Also the interrupt service routines are dependent on this activations.

Attention:

The adjustments in this chapter *do not* set the used IDE environment settings or options! Make sure, that the used device, the provided tool's Flash programming tool, and the linker settings are properly adjusted to the device, which is defined in *l3_user.h*!

4.4.1 Device Definition

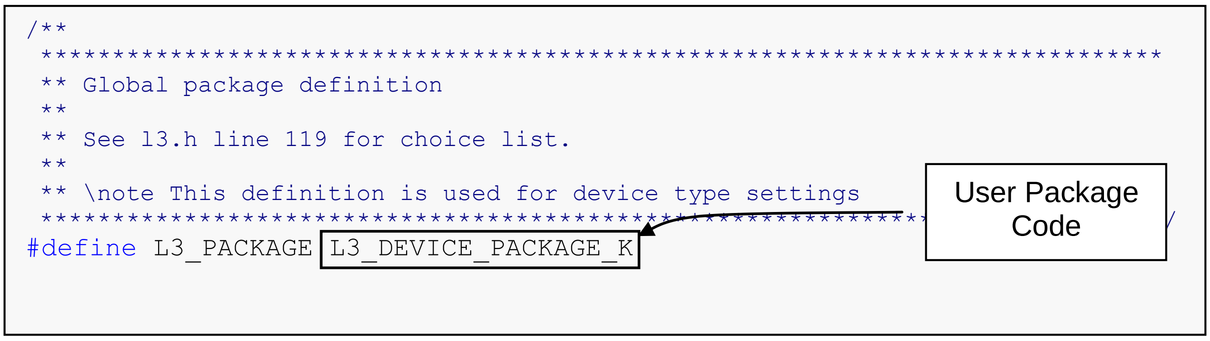
The device definition macro is `L3_MCU` and the device name is checked in *l3.h*.



```
/**
 ******************************************************************************
 ** Global device definition
 **
 ** See l3.h for select list.
 **
 ** \note This definition is used for GPIO settings, interrup
 **       enumeration handling, and Type(0, 1, 3, ...) Dev
 *************************************************************               ***/
#define L3_MCU L3_DEVICE_TYPE_MB9BF50X
```

4.4.2 Package Definition

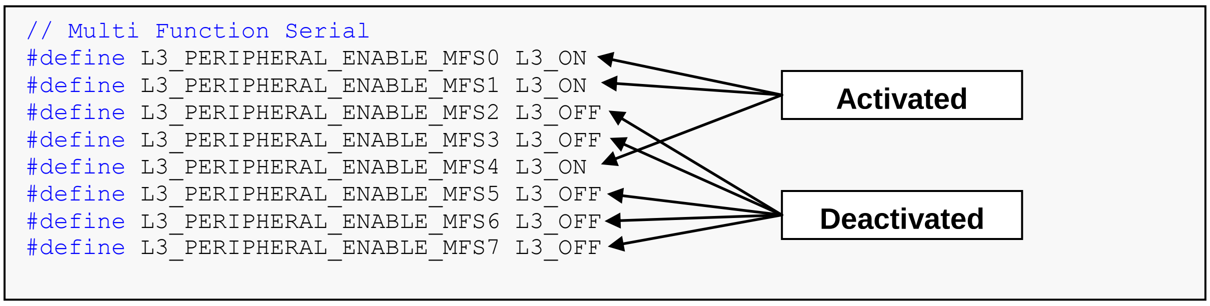
Together with the device name the L3 is able to determine the device type. Therefore the package has also to be defined.



```
/**
 ******************************************************************************
 ** Global package definition
 **
 ** See l3.h line 119 for choice list.
 **
 ** \note This definition is used for device type settings
 ***************************************************************            /
#define L3_PACKAGE L3_DEVICE_PACKAGE_K
```

4.4.3 Multiple Instances Resource Activation

For each instance of a resource there is an individual definition for activation. In the following example, there is shown the activation definitions for the Multi Function Serial Interface:



```
// Multi Function Serial
#define L3_PERIPHERAL_ENABLE_MFS0 L3_ON
#define L3_PERIPHERAL_ENABLE_MFS1 L3_ON
#define L3_PERIPHERAL_ENABLE_MFS2 L3_OFF
#define L3_PERIPHERAL_ENABLE_MFS3 L3_OFF
#define L3_PERIPHERAL_ENABLE_MFS4 L3_ON
#define L3_PERIPHERAL_ENABLE_MFS5 L3_OFF
#define L3_PERIPHERAL_ENABLE_MFS6 L3_OFF
#define L3_PERIPHERAL_ENABLE_MFS7 L3_OFF
```

Always use keywords `L3_ON` or `L3_OFF` for this purpose!

Carefully check the activation before usage, whether your device supports a peripheral and especially a certain instance, because the L3 does *not* check for availability by device name and package!

4.4.4 Single Instances Resource Activation

Any resource, which only has a single instance is defined the same way like multiple-instance resources. The resource gets an index `0` for this purpose.

The MB9BF50x series only has one Dual Timer. Therefore the definition looks like the following:

0-Instace

```
// Dual Timer
#define L3_PERIPHERAL_ENABLE_DT0   L3_ON
```

4.4.5 Interrupt Level Definition

The interrupt level definitions must be set by the user.

It may look like the following for e.g. the Multi Function Serial:

User Levels

```
#define L3_IRQ_LEVEL_MFS0_TX          2
#define L3_IRQ_LEVEL_MFS0_RX          2
#define L3_IRQ_LEVEL_MFS1_TX          7
#define L3_IRQ_LEVEL_MFS1_RX          7
   . . .
```

4.4.6 Numerical Definitions

Some feature amount (e.g. number of ADC inputs) are not handled by the L3 itself because it depends on the device itself and the user demands. Therefore they have to be set by the user.

4.4.6.1 Number of ADC channels

The number of ADC channels is specified by the following definition.

```
/**
 *******************************************************************************
 ** \brief User ADC input configuration
 **
 ** Number of analog input pins (1 to 16)
 *******************************************************************************/
#define L3_ADC_CHANNELS          16
```

Do not specify a number higher than the maximum available analog inputs.

4.4.6.2 ADC FIFO Depth

The number of ADC scan conversion FIFO (*not* priority scan FIFO!) is specified by the following definition.

```
/**
 ******************************************************************************
 ** \brief User ADC FIFO depth configuration
 **
 ** ADC Scan conversion FIFO depth (1 to 16)
 ******************************************************************************/
#define L3_ADC_FIFO_DEPTH        16
```

Do not specify a number higher than the maximum available analog inputs.

4.4.6.3 Number of DMA Channels

The number of available DMA channels has to be specified. If a device does not support DMA, it is not needed to set this to definition to 0, because it is overruled by the device type definition.

```
/**
 ******************************************************************************
 ** \brief User Number of DMA channels. If a device does not support DMA it
 ** is not needed to set this value to 0.
 **
 ** Number of available DMA channels (1 to 8)
 ******************************************************************************/
#define L3_DMA_CHANNELS           8
```

4.4.6.4 Number of external Interrupts

The number of available external interrupts is specified by the following definition.

```
/**
 ******************************************************************************
 ** \brief User Number of external interrupt channels
 **
 ** Number of available EXINT channels (1 to 16)
 ******************************************************************************/
#define L3_EXINT_CHANNELS        16
```

Do not specify a number higher than the maximum available external interrupt inputs.

4.4.7 Special Activators

Some activators and availability of a resource cannot be done by the device and package definition only. For these cases some special activators are provided, and the use has to set them individually.

4.4.7.1 PPG 16+16 Mode

Because the 16+16 mode of the PPG is not used often, it can be “undefined” by setting `L3_PPG_1616_MODE_AVAILABLE` to `L3_OFF`.

```
/**
 ******************************************************************************
 ** \brief User 16+16 Bit activation for PPG configuration
 **
 ** Possible values are:
 ** - L3_ON    PPG driver supports 16+16 bit mode
 ** - L3_OFF   PPG driver does not support 16+16 bit mode
 ******************************************************************************/
#define L3_PPG_1616_MODE_AVAILABLE L3_OFF
```

In this case the extra configuration elements u16Ppg6Low and u16Ppg6High of the PPG's configuration stc_mft_ppg_config_t are not compiled to save memory.

4.4.7.2 Number of ADC Channels

The user can adjust the maximum number of ADC channels by setting L3_ADC_CHANNELS to a certain value.

```
/**
 ******************************************************************************
 ** \brief User ADC input configuration
 **
 ** Number of analog input pins (1 to 24)
 ******************************************************************************/
#define L3_ADC_CHANNELS  16
```

Note, that using smaller numbers than the maximum only skip the left upper channels.

4.4.7.3 ADC FIFO depth

The user can also set the FIFO depth. Normally it should be the same value than L3_ADC_CHANNELS.

```
/**
 ******************************************************************************
 ** \brief User ADC FIFO depth configuration
 **
 ** ADC Scan conversion FIFO depth (1 to 24)
 ******************************************************************************/
#define L3_ADC_FIFO_DEPTH  16
```

4.4.7.4 Number of External Interrupts

With L3_EXINT_CHANNELS the user determines the maximum number of External Interrupt Inputs.

```
/**
 ******************************************************************************
 ** \brief User Number of external interrupt channels
 **
 ** Number of available EXINT channels (1 to 16)
 ******************************************************************************/
#define L3_EXINT_CHANNELS     16
```

4.4.7.5 No Flash RAMCODE Compiler Switch

This definition sets RAMCODE-Flash routines compiling or not. Note, that this is an Opt-Out logic by choosing L3_ON!

```
/**
 ******************************************************************************
 ** \brief User does not need Flash routines in RAMCODE
 **
 ** Possible values are:
 ** - L3_ON   Flash routines in RAMCODE are not compiled
 ** - L3_OFF  Flash routines in RAMCODE active
 ******************************************************************************/
#define L3_NO_FLASH_RAMCODE      L3_OFF
```

4.4.7.6 Flash Routines DMA Halt

This macro defines whether to halt beside interrupts also possible DMA transfers before calling RAM routines.

```
/**
 ******************************************************************************
 ** \brief Global DMA halt during Flash routines
 **
 ** Possible values are:
 ** - L3_ON    DMA is globally halted during Flash routine executions
 ** - L3_OFF   DMA is not touched in Flash routines
 ******************************************************************************/
#define L3_FLASH_DMA_HALT      L3_ON
```

4.4.7.7 RAMCODE NMI Vector Handle

If an NMI occurs during RAM code execution, the NMI vector must be temporarily patched to RAM location. This switch determines whether to do this or not.

```
/**
 ******************************************************************************
 ** \brief Global NMI service during Flash routines. Only used, if RAMCODE is
 **        used and L3_FLASH_DMA_HALT == L3_ON.
 **
 ** \note  Because the NMI callback of the Ext-I/F driver module is used,
 **        also L3_PERIPHERAL_ENABLE_EXINT0_7 has to be set to L3_ON.
 **
 ** Possible values are:
 ** - L3_ON    NMI is pre-serviced in RAM vector code
 ** - L3_OFF   NMI is not handled in RAM routines
 ******************************************************************************/
#define L3_NMI_RAM_VECTOR_HANDLE      L3_ON
```

4.4.7.8 NMI RAMCODE Vector Address

This macro defines the location of the NMI vector in RAM. Take care of correct address choice.

```
/**
 ******************************************************************************
 ** \brief NMI RAM vector address. Only used, if L3_NMI_RAM_VECTOR_HANDLE ==
 **        L3_ON, L3_FLASH_DMA_HALT == L3_ON, and RAMCODE used.
 **
 ** Used for remapping VTOR to RAM area. Must be defined by user. Note that the
 ** last 6 bits are always treated as '0'. A global varibale
 ** u32L3NmiVectorAddress for the RAM NMI vector is placed at this address + 8.
 **
 ** \attention Be careful by choosing the address! It must not interfere with
 **            the RAM vector table, if 'Debug' build is used. The address must
 **            be less than 0x20000000 and bigger than the bottom of the RAM.
 **            There is no automatic plausibility check!
 ******************************************************************************/
#define L3_RAM_VECTOR_TABLE_ADDRESS   0x1FFFFF00
```

4.5 File: interrupts.c

The *interrupts.c* file contains all first-level service routines (interrupt vectors of start code). 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 should not be changed by the user.

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

```
// DT ISR not shared with QPRC
#if ((L3_PERIPHERAL_ENABLE_QPRC0 != L3_ON)  && \
     (L3_PERIPHERAL_ENABLE_QPRC1 != L3_ON)) && \
     (L3_PERIPHERAL_ENABLE_DT0   == L3_ON)
void DT_Handler(void)
{
  DtIrqHandler0((stc_qprn_t*)&DT0,
                &(m_astcDtInstanceDataLut[DtInstanceIndexDt0].stcInternData));
}

// QPRC ISR not shared with DT
#elif ((L3_PERIPHERAL_ENABLE_QPRC0 == L3_ON)  || \
       (L3_PERIPHERAL_ENABLE_QPRC1 == L3_ON)) && \
       (L3_PERIPHERAL_ENABLE_DT0   != L3_ON)
void DT_Handler(void)
{
  #if (L3_PERIPHERAL_ENABLE_QPRC0 == L3_ON)
    if ((FM3_INTREQ->IRQ06MON & IRQ06MON_QPRC0_BITMASK) != 0)
    {
      QprcIrqHandler((stc_qprcn_t*)&QPRC0,
                &(m_astcQprcInstanceDataLut[QprcInstanceIndexQprc0].stcInternData));
    }
  #endif

  #if (L3_PERIPHERAL_ENABLE_QPRC1 == L3_ON)
    if ((FM3_INTREQ->IRQ06MON & IRQ06MON_QPRC1_BITMASK) != 0)
    {
      QprcIrqHandler((stc_qprcn_t*)&QPRC1,
                &(m_astcQprcInstanceDataLut[QprcInstanceIndexQprc0].stcInternData));
    }
  #endif
}

// DT and QPRC have shared ISR
#elif ((L3_PERIPHERAL_ENABLE_QPRC0 == L3_ON)  || \
       (L3_PERIPHERAL_ENABLE_QPRC1 == L3_ON)) && \
       (L3_PERIPHERAL_ENABLE_DT0   == L3_ON)
void DT_Handler(void)
{
  if ((FM3_DTIM->TIMER1RIS_f.TIMERXRIS == 1) || (FM3_DTIM->TIMER2RIS_f.TIMERXRIS == 1))
  {
    DtIrqHandler0((stc_dtn_t*)&DT0,
                &(m_astcDtInstanceDataLut[DtInstanceIndexDt0].stcInternData));
  }

  #if (L3_PERIPHERAL_ENABLE_QPRC0 == L3_ON)
    if ((FM3_INTREQ->IRQ06MON & IRQ06MON_QPRC0_BITMASK) != 0)
    {
      QprcIrqHandler((stc_qprcn_t*)&QPRC0,
                &(m_astcQprcInstanceDataLut[QprcInstanceIndexQprc0].stcInternData));
    }
  #endif

  #if (L3_PERIPHERAL_ENABLE_QPRC1 == L3_ON)
    if ((FM3_INTREQ->IRQ06MON & IRQ06MON_QPRC1_BITMASK) != 0)
    {
      QprcIrqHandler((stc_qprcn_t*)&QPRC1,
                &(m_astcQprcInstanceDataLut[QprcInstanceIndexQprc0].stcInternData));
    }
  #endif
}
#endif
```

For problems, which may occur at compile time in *interrupts.c* please also read the hints in chapter 6.3.

Also refer to supported interrupt types described in chapter 6.4.

4.6 File: interrupts.h

This header module includes the resource's header files to collect the interrupt service routines prototypes.

Additionally it defines some needed IRQMON bit masks for shared interrupt vectors.

4.7 File: ramcode.c

This module holds Flash programming routines, which are linked for the RAM area. For IAR and KEIL tool chains the needed compiler directives or identifier are defined for the automatically copying during start-up phase.

The directives have the following `#if/#else` structure:

```
#ifdef __ICCARM__
  __ramfunc
#elif  __CC_ARM
  __attribute__ ((section (".ramfunc")))
#else
  #error Please check compiler and linker settings for RAM code
#endif
       . . .
```

Between the `#elif` and `#else` another `#elif` can be inserted for a different compiler by the user:

```
#ifdef __ICCARM__
  __ramfunc
#elif  __CC_ARM
  __attribute__ ((section (".ramfunc")))
#elif  __USER_COMPILER__
  // User RAMCODE attributes
#else
  #error Please check compiler and linker settings for RAM code
#endif
       . . .
```

4.8 File: ramcode.h

This header files contains the Flash sequence addresses and command definitions and the prototypes of the *ramcode.c* functions.

5 L3 Resource Driver API

DETAILED DESCRIPTION OF THE RESOURCE DRIVER API AND EXAMPLE CODE

5.1 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* pstcModule,`
`                      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();

  . . .
}
```

See chapter 5.18.5 for more details.

Carefully check the resource activation in* l3_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 L3 is not able to check for availability.

5.2 (ADC) Analog Digital Converter Module

Type Definition	`stc_adcn_t`
Configuration Type	`stc_adc_config_t`
Address Operator	`ADC`*`n`*

Each ADC channel (max. 16) gets an own buffer, which has to be set-up by the user application. These buffers and their states are defined in the ADC configuration `stc_adc_config_t`.

The buffer pointers are collected in `stc_adc_config_t::pu8ChannelBufferPointer[]`, if 8 Bit conversion method is used or `stc_adc_config_t::pu16ChannelBufferPointer[]`, if 12 Bit conversion method is used.

The recent buffer index is stored in `stc_adc_config_t::u16ChannelBufferCounter`. This value has to be initialized to `0` by the user or to be set to any value less than the buffer size.

The end index has to be stored in `stc_adc_config_t::u16ChannelBufferEndCount`. This value can be the end of the buffer or any value between the recent index and the buffer size.

If no interrupts are used, init the ADC first, start the conversion via `Adc_TriggerStart()`, the status can be polled by `Adc_ConversionStatus()`. The buffer(s) then have to be filled by `Adc_StoreData()`.

If interrupts or DMA is used, init ADC first and start the conversion via `Adc_TriggerStart()`, the callback function referenced in `stc_adc_config_t::pfnCallback` then indicates the complete filled buffer(s).

`Adc_PurgeFifo()` and `Adc_PurgePrioFifo()` can be called, if an error (FIFO overrun) occurred. All remaining data will get lost in this case! Nevertheless the overrun flags are cleared automatically by the ADC interrupt handler before the error callback is used (if defined).

Note:

- The FM3 ADC supports only DMA in scan conversion mode.
- If DMA transfer is selected only full-12-bit conversion mode is supported!
- If DMA transfer is selected only one common buffer can be used for each activated channel (HW restriction)! Recently all 32-bit data (including ADC value and channel number) are stored in an 32-bit buffer.
- Using priority scan mode, interrupts must be enabled and DMA must be ***disabled***. The user has to take care, that the priority scan buffer is big enough (according priority FIFO scan interrupt depth). The priority scan buffer per channel must have the depth of 1 (according 8-Bit or 16-Bit mode).
- Type3 devices do not support DMA. Thus `en_adc_interruptsdma_t::AdcDma` is not available if Device Type3 is selected.
- Since L3 version 1.1 only `Adc_Init()` need a pointer to the user configuration `stc_adc_config_t`.

5.2.1 Interrupt Scan Conversion

If the scan conversion only and interrupt method is enabled, the conversion of several channels are stored in own buffers. These buffers will have all the same size. An end-notification is available via `stc_adc_config_t:: pfnCallback` or in erroneous case: `stc_adc_config_t:: pfnErrorCallback` is called, when the buffers are filled (or an error occurred).

The ADC driver takes care of the FIFO fill level according to the data left to be stored and the number of the channels.

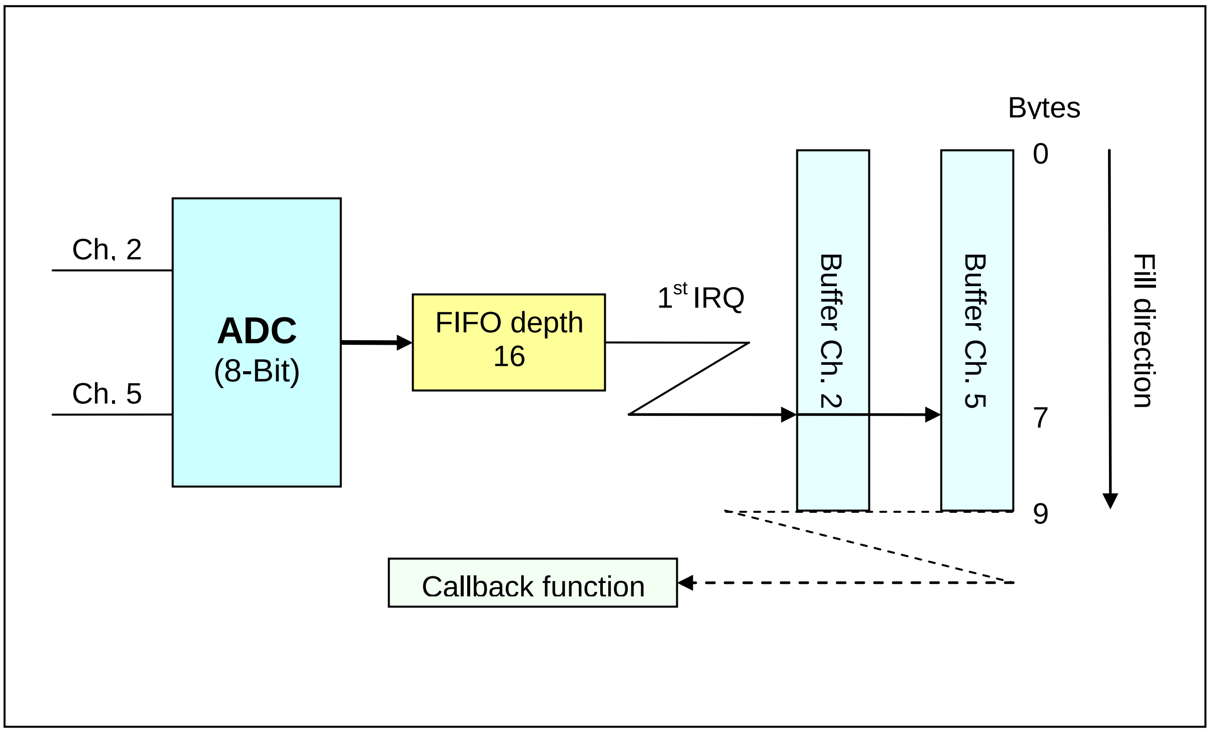


Figure 5-1: Example ADC Interrupt Scan Conversion

Assume channel 2 and 5 are selected for 8-Bit conversion. The buffer sizes may be 10 bytes for each channel, which means 20 conversions overall. In this case the starting FIFO depth is set to maximum, i.e. 16, which means 8 conversions for each of the two channels. After these 16 conversions the ADC driver gets the first interrupt and fills the buffers with 8 bytes each according to the channel information. Then the driver updates the buffer pointer and calculates 2 bytes left for each channel. Then the FIFO depth is set to 4. The next (and last) interrupt then stores the 4 FIFO conversions for the 2 buffers, the interrupt is disabled and the callback function is called for end-notification to the application.

The figure above shows the state, where the 1st interrupt have been occurred and the buffers have been updated with the 8 bytes each in the interrupt service routine of the driver. The callback function call, which will be called later, is shown in dashed lines.

5.2.2 DMA Scan Conversion

If DMA is used, there is no method to fill channel-dedicated buffers. Therefore the user has to know, that the full 32-Bit FIFO data is stored in a single buffer. After the DMA event (and thus the filled buffer) the software has to read-out the channel data to determine, which channel data was stored.

If a single channel conversion is desired the channel bits of the FIFO are obsolete. In this case a single buffer can be filled with the conversion data only by using the upper 16-Bit of the SCFD register as the DMA data source.

Note that the priority scan has no DMA functionality. It also cannot be used with interrupts while the scan conversion uses DMA.

5.2.3 Priority Conversion

The priority conversion can be used optional besides scan conversion. It is recommended to set the priority FIFO depth to the number of expected priority conversions, to get an immediate callback.

5.2.4 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`	`u32Channel…Select`	–	32-Bit ADC channel bitfield. LSB represents enabling channel 0, MSB represents enabling channel 31.
`En_adc_…conversion…mode_t`	`enConver…sionMode`	`AdcModeScan…Conversion` `AdcModePriority…Conversion`	ADC configured to scan conversion ADC configured to scan and priority conversion
`uint8_t`	`u8Sampling…Time`	–	Sampling time. See peripheral manual (analog part) for correct settings.

`En_adc_…sampling…time…multiplier…_t`	`enSampling…Time…Multiplier`	`AdcSamplingTime Multiplier…`
		`…1`
		`…4`
		`…8`
		`…16`
		`…32`
		`…64`
		`…128`
		`…256`
			Sampling time multiplier:
			Sampling time multiplied by 1
			Sampling time multiplied by 4
			Sampling time multiplied by 8
			Sampling time multiplied by 16
			Sampling time multiplied by 32
			Sampling time multiplied by 64
			Sampling time multiplied by 128
			Sampling time multiplied by 256

`uint8_t`	`u8…Comparison…TimeDivider`	–	Frequency division ratio for comparison time
`oolean_t`	`bSingle…Conversion`	`TRUE` `FALSE`	Single conversion Repeated conversion
`oolean_t`	`bReset…Buffer…Counter`	`TRUE` `FALSE`	Reset buffer counter(s) at end of conversion (for repeated conversion) Does not reset buffer counter(s)
`en_adc_…resolution…_t`	`en…Resolution`	`Adc12Bit` `Adc8Bit`	ADC results in full 12 Bit ADC results in 8 Bit

`en_adc_…interrupts…dma_t`	`enUse…Interrupts…Dma`	`AdcPolling`
		`AdcInterrupts`
		`AdcDma`
			Polling mode for result
			Interrupts used
			DMA used (if provided by MCU)

`uint8_t*`	`pu8Channel…Buffer…Pointer…[L3_ADC_…CHANNELS]`	–	Pointer list to channel buffers (8 bit resolution)
`uint16_t*`	`pu16Channel…Buffer…Pointer…[L3_ADC_…CHANNELS]`	–	Pointer list to channel buffers (16 bit resolution)

uint8_t*	pu8Priority… Buffer… Pointer… [L3_ADC_… CHANNELS]	-	Pointer list to priority channel buffers (8 bit resolution)
uint16_t*	pu16Priority… Buffer… Pointer… [L3_ADC_… CHANNELS]	-	Pointer list to priority channel buffers (16 bit resolution)
uint32_t*	pu32Dma… Buffer… Pointer	-	Pointer to DMA (plain) data. Plain means original FIFO data.
Uint16_t	u16Channel… Buffer… Counter	-	Count of channel buffers
uint16_t	u16Channel… Buffer… EndCount	-	Maximum buffer size or end count
boolean_t	bExternal… Trigger… AnalogInput	TRUE FALSE	Set external trigger input selection Do not set these bits
boolean_t	bPriority… Conversion… External… Start	TRUE FALSE	Priority conv. External start enable Priority conv. External start disable
boolean_t	bPriority… Conversion… TimerStart	TRUE FALSE	Priority conversion timer start enable Priority conversion timer start disable
en_adc_… prio_irq_… level_t	enPriority… Conversion… IrqLevel	AdcPrioIrqFifoLevel1 AdcPrioIrqFifoLevel2 AdcPrioIrqFifoLevel3 AdcPrioIrqFifoLevel4	IRQ request, when result in 1st FIFO stage IRQ request, when result in 2nd FIFO stage IRQ request, when result in 3rd FIFO stage IRQ request, when result in 4th FIFO stage
uint8_t	u8PrioLevel1… AnalogInput… Channel	0 … 7	Priority level 1 analog input selection channel
uint8_t	u8PrioLevel2… AnalogInput… Channel	0 … 31	Priority level 2 analog input selection channel
boolean_t	bComparison… Enable	TRUE FALSE	Enables comparison function Disables comparison function
boolean_t	bComparision… GreaterEqual	TRUE FALSE	IRQ, if ADC value ≥ CMPD IRQ, if ADC value < CMPD
boolean_t	bComparison… Channel	TRUE FALSE	Enables comparison channel Disables comparison channel
uint8_t	u8Comparison… Channel	0 … 31	Defines channel of comparison, if enabled
uint16_t	u16Comparison… Value	-	Value of comparison
uint8_t	u8Enable… CycleValue	-	For device types ≠ 0: Enables cycle setting. Use 0xFF as default.
Func_ptr_t	pfnCallback	-	End of conversion callback pointer
func_ptr_t	pfnError… Callback	-	Callback pointer for error during conversion occurred
func_ptr_t	pfnPriority… CallbackAdc	-	Priority scan callback pointer
func_ptr_t	pfnPrio…	-	Priority FIFO overrun error callback

	`Error… CallbackAdc`		pointer

5.2.5 Adc_Init()

This function initializes an ADC instance according the given configuration.

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 successfully done.
`ErrorInvalidParameter`	• `pstcAdc == NULL` • `pstcConfig == NULL` • Other invalid configuration settings

5.2.6 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`	De-Initialization successfully done.
`ErrorInvalidParameter`	`pstcAdc == NULL`

5.2.7 Adc_TriggerStart()

This function triggers an ADC instance for conversion.

Prototype	
`en_result_t Adc_TriggerStart( volatile stc_adcn_t* pstcAdc )`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`Ok`	Trigger done.
`ErrorInvalidParameter`	`pstcAdc == NULL`

5.2.8 Adc_ConversionStatus()

This function returns an ADC instance conversion status.

Prototype	
`en_result_t Adc_ConversionStatus( volatile stc_adcn_t* pstcAdc )`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`Ok`	Conversion finished
`OperationInProgress`	Conversion ongoing
`ErrorInvalidParameter`	`pstcAdc == NULL`

5.2.9 Adc_StoreData()

This function stores a conversion data into the correct channel buffer.

Prototype	
`en_result_t Adc_StoreData( volatile stc_adcn_t* pstcAdc )`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`Ok`	Data stored successfully.
`ErrorBufferFull`	Data buffer full, data not stored.
`ErrorInvalidParameter`	`pstcAdc == NULL`

5.2.10 Adc_PurgeFifo()

This function purges the scan conversion FIFO and rejects all data. It should be called in case of an error.

Prototype	
`en_result_t Adc_PurgeFifo( volatile stc_adcn_t* pstcAdc )`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`Ok`	Scan conversion FIFO purged
`ErrorInvalidParameter`	`pstcAdc == NULL`

5.2.11 Adc_PurgePrioFifo()

This function purges the priority scan conversion FIFO and rejects all data. It should be called in case of an error.

Prototype	
`en_result_t Adc_PurgePrioFifo( volatile stc_adcn_t* pstcAdc )`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`Ok`	Priority scan conversion FIFO purged
`ErrorInvalidParameter`	`pstcAdc == NULL`

5.2.12 Adc_GetStatus()

This function returns the status of an ADC instance. Note that the return type is `en_adc_status_t`.

Prototype	
`en_adc_status_t Adc_GetStatus( volatile stc_adcn_t* pstcAdc )`	
Parameter Name	**Description**
`[in] pstcAdc`	ADC instance pointer
Return Values	**Description**
`AdcErrorUnknownState`	`pstcAdc == NULL` or unknown state (should never happen)
`AdcStandby`	Standby for A/D conversion.
`AdcPrioInProgress`	Priority A/D conversion (priority level 1 or 2) is in progress.
`AdcPrioInProgress… ScanPending`	Priority A/D conversion (priority level 1 or 2) is in progress Scan conversion is pending.
`AdcPrio1InProgress… Prio2Pending`	Priority A/D conversion (priority level 1) is in progress. Priority conversion (priority level 2) is pending.
`AdcPrio1InProgress… Prio2ScanPending`	Priority A/D conversion (priority level 1) is in progress. Scan conversion and priority conversion (priority level 2) are pending.

5.2.13 CallbackAdc()

This function is called, if `stc_adc_config_t::pfnCallbackAdc` is defined and the according interrupt is enabled.

Callback Function
`void` *`CallbackAdc`*`( void )`

5.2.14 ErrorCallbackAdc()

This function is called, if `stc_adc_config_t::pfnErrorCallbackAdc` is defined and the according interrupt is enabled.

Callback Function
`void` *`ErrorCallbackAdc`*`( void )`

5.2.15 PriorityCallbackAdc()

This function is called, if `stc_adc_config_t::pfnPriorityCallbackAdc` is defined and the according interrupt is enabled.

Callback Function
`void PriorityCallbackAdc( void )`

5.2.16 PriorityErrorCallbackAdc()

This function is called, if `stc_adc_config_t::pfnPrioErrorCallbackAdc`is defined and the according interrupt is enabled.

Callback Function
`void PriorityErrorCallbackAdc( void )`

5.2.17 Example Code

The following code excerpts show some applications of using the ADC driver.

5.2.17.1 Polling Mode, 8-Bit

The following code shows how to use the ADC in polling mode using 8-Bit conversion.

```
#include "adc.h"

#define BUFFER8SIZE 16

function
{
  stc_adcn_t*      pstcAdc0 = NULL;
  stc_adc_config_t stcAdcConfig0;
  uint8_t          au8AdcBufferCh0[BUFFER8SIZE];
  uint8_t          u8Counter;

  L3_ZERO_STRUCT(stcAdcConfig0);

  stcAdcConfig0.u32ChannelSelect.AD_CH_0   = 1;         // Enable AN0 for ADC0
  stcAdcConfig0.enConversionMode           = AdcModeScanConversion;
  stcAdcConfig0.enResolution               = Adc8Bit;
  stcAdcConfig0.pu8ChannelBufferPointer[0] = au8AdcBufferCh0;
  stcAdcConfig0.u16ChannelBufferCounter  = 0;             // Init buffer counter
  stcAdcConfig0.u16ChannelBufferEndCount = BUFFER8SIZE;  // Set buffer end
  stcAdcConfig0.enUseInterruptsDma       = AdcPolling;

  // Sampling Time: 9 * 4 + 1 = 37 cycles => 25 ns/cycle * 37 cycles = 925 ns
  stcAdcConfig0.u8SamplingTime = 9;
  stcAdcConfig0.enSamplingTimeMultiplier = AdcSamplingTimeMultiplier4;

  // Comparison cycles: Divider 5 => 25 ns/cycle * 5 = 125 ns =>
  //   125 ns * 14 = 1750 ns overall comparison time (TYPE0 device)
  stcAdcConfig0.u8ComparisonTimeDivider = 5;

  if (Ok == Adc_Init((stc_adcn_t*)&ADC0, &stcAdcConfig0))
  {
    pstcAdc0 = (stc_adcn_t*)&ADC0;

    for (u8Counter = 0; u8Counter < BUFFER8SIZE; u8Counter++)
    {
      Adc_TriggerStart(pstcAdc0);

      while(Ok != Adc_ConversionStatus(pstcAdc0));       // Polling ...
      Adc_StoreData(pstcAdc0);
    }
  }
}
```

5.2.17.2 Interrupt Mode, 8-Bit

The following code shows how to use the ADC in interrupt mode using 8-Bit scan conversion.

```
#include "adc.h"

#define BUFFER8SIZE 16

uint8_t u8ConversionDone = 0;

void Adc_Callback(void)
{
  u8ConversionDone = 1;
}

void Adc_ErrorCallback(void)
{
  // Do something ...
}

function
{
  stc_adcn_t*       pstcAdc0 = NULL;
  stc_adc_config_t stcAdcConfig0;
  uint8_t           au8AdcBufferCh0[BUFFER8SIZE];

  L3_ZERO_STRUCT(stcAdcConfig0);

  stcAdcConfig0.u32CannelSelect.AD_CH_0 = 1;  // Enable AN0 for ADC0
  stcAdcConfig0.u32CannelSelect.AD_CH_1 = 1;  // Enable AN0 for ADC0
  stcAdcConfig0.enConversionMode = AdcModeScanConversion ;
  stcAdcConfig0.enResolution = Adc8Bit;
  stcAdcConfig0.pu8ChannelBufferPointer[0] = au8AdcBufferCh0;
  stcAdcConfig0.pu8ChannelBufferPointer[1] = au8AdcBufferCh1;
  stcAdcConfig0.u16ChannelBufferCounter = 0;            // Init buffer counter
  stcAdcConfig0.u16ChannelBufferEndCount = BUFFER8SIZE; // Set buffer end
  stcAdcConfig0.enUseInterruptsDma = AdcInterrupts;
  stcAdcConfig0.pfnCallback = &Adc_Callback;
  stcAdcConfig0.pfnErrorCallback = &Adc_ErrorCallback;

  // Sampling Time: 9 * 4 + 1 = 37 cycles => 25 ns/cycle * 37 cycles = 925 ns
  stcAdcConfig0.u8SamplingTime = 9;
  stcAdcConfig0.enSamplingTimeMultiplier = AdcSamplingTimeMultiplier4;

  // Comparison cycles: Divider 5 => 25 ns/cycle * 5 = 125 ns =>
  //   125 ns * 14 = 1750 ns overall comparison time (TYPE0 device)
  stcAdcConfig0.u8ComparisonTimeDivider = 5;

  if (Ok == Adc_Init((stc_adcn_t*)&ADC0, &stcAdcConfig0))
  {
    pstcAdc0 = (stc_adcn_t*)&ADC0;

    Adc_TriggerStart(pstcAdc0);

    while(0 == u8ConversionDone);              // Wait for finish ...
  }
}
```

5.2.17.3 DMA Mode, 12-Bit

The following code shows how to use the ADC in DMA mode using 12-Bit scan conversion. Note, that the full 32-Bit FIFO data (inclusive channel number) is transferred via DMA to the buffer.

For DMA usage refer to chapter 5.10.

```
#include "adc.h"
#include "dma.h"

#define DMABUFFERSIZE 16

uint8_t u8ConversionDone = 0;

void Adc_Callback(void)
{
  u8ConversionDone = 1;
}

void Adc_ErrorCallback(void)
{
  // Do something ...
}

void Dma_ErrorCallback(uint8_t u8ErrorCode)
{
  // Do something ...
}

function
{
  stc_dma_config_t stcDmaConfig;
  stc_adcn_t*      pstcAdc0 = NULL;
  stc_adc_config_t stcAdcConfig0;
  uint32_t         au32AdcBuffer[DMABUFFERSIZE];

  L3_ZERO_STRUCT(stcAdcConfig0);

  // Setup ADC
  stcAdcConfig0.u32CannelSelect.AD_CH_0  = 1;
  stcAdcConfig0.enConversionMode         = AdcModeScanConversion ;
  stcAdcConfig0.enResolution             = Adc12Bit;
  stcAdcConfig0.pu32DmaBufferPointer     = au32AdcBuffer;
  stcAdcConfig0.u16ChannelBufferCounter  = 0;
  stcAdcConfig0.u16ChannelBufferEndCount = DMABUFFERSIZE;
  stcAdcConfig0.enUseInterruptsDma       = AdcDma;
  stcAdcConfig0.pfnCallback              = &Adc_Callback;
  stcAdcConfig0.pfnErrorCallback         = &Adc_ErrorCallback;
  stcAdcConfig0.u8SamplingTime           = 9;
  stcAdcConfig0.enSamplingTimeMultiplier = AdcSamplingTimeMultiplier4;
  stcAdcConfig0.u8ComparisonTimeDivider  = 5;

  // Setup DMA
  stcDmaConfig.bEnable                     = 0;
  stcDmaConfig.bPause                      = 0;
  stcDmaConfig.bSoftwareTrigger            = 0;
  stcDmaConfig.enDmaIdrq                   = Adc0;
  stcDmaConfig.u8DmaChannel                = 0;
  stcDmaConfig.u8BlockCount                = 0;
  stcDmaConfig.u16TransferCount            = DMABUFFERSIZE;
  stcDmaConfig.enTransferMode              = DmaDemandTransfer;
  stcDmaConfig.enTransferWidth             = Dma32Bit;
  stcDmaConfig.u32SourceAddress            = (uint32_t) &(FM3_ADC0->SCFD);
  stcDmaConfig.u32DestinationAddress       = (uint32_t) &au32AdcBuffer;
  stcDmaConfig.bFixedSource                = TRUE;
  stcDmaConfig.bFixedDestination           = FALSE;
  stcDmaConfig.bReloadCount                = FALSE;
  stcDmaConfig.bReloadSource               = FALSE;
  stcDmaConfig.bReloadDestination          = FALSE;
  stcDmaConfig.bErrorInterruptEnable       = TRUE;
  stcDmaConfig.bCompletionInterruptEnable  = TRUE;
  stcDmaConfig.bEnableBitMask              = FALSE;
  stcDmaConfig.pfnCallback                 = &Adc_Callback;
```

▼

▲

```
  stcDmaConfig.pfnErrorCallback           = &Dma_ErrorCallback;


  Dma_Init_Channel(&stcDmaConfig);
  Dma_Enable();

  if (Ok == Adc_Init((stc_adcn_t*)&ADC0, &stcAdcConfig0))
  {
    pstcAdc0 = (stc_adcn_t*)&ADC0;

    // "Unlock" DMA
    stcDmaConfig.bEnable = 1;
    Dma_Set_Channel(&stcDmaConfig);

    // Start ADC
    Adc_TriggerStart(pstcAdc0);

    while(0 == u8ConversionDone);

    Dma_DeInit_Channel(&stcDmaConfig);
  }
}
```

5.3 (BT) Base Timer

Type Definition	`stc_btn_t`
Configuration Type (BT)	`stc_bt_config_t`
Configuration Type (BTIOSEL)	`stc_bt_iosel_config_t`
Address Operator	`BT`*n*

To initialize a BT instance, use `Bt_Init()`. After this the BT instance can be enabled by `Bt_Enable()` and disabled by `Bt_Disable()`. To trigger an instance by software use `Bt_Trigger()`. After operation `Bt_DeInit()` can be used to de-initialize a BT instance and disable its interrupts.

For each BT mode callback functions are provided for the following events:

- Callback0: Timer start trigger detection for PWM, PPG, RLT; Completion of measurement for PWC
- Callback1: Underflow detection for PWM, PPG, RLT; Overflow detection for PWC
- Callback2: Duty cycle detection for PWM

Attention:

If an interrupt reason is enabled by the configuration, the corresponding callback function pointer **must** be defined. The BT ISR does not check the callback pointer for definition because of speed reasons!

The BT I/O selector is configured by `Bt_InitIoSel()`. The parameters are the BT instance pair (0/1, 2/3, 4/5, or 6/7) and the mode.

`Bt_TriggerInstances()` trigger bit-pattern-selected BT instances software-based. An I/O selector can be de-initialized (reset to mode 0) by using `Bt_DeInitIoSel()`.

To reset all BT I/O selectors `Bt_ResetIoSel()` can be used.

`Bt_ReadTimer16()` or `Bt_ReadTimer32()` reads out the recent timer count, if RLT mode was set.

`Bt_ReadDataBuffer16()` or `Bt_ReadDataBuffer32()` reads out the result count, if PWC mode was set. Reading the data buffer is **mandatory** for the callback function, because the interrupt request is only cleared by reading the data buffer – not by 'clearing' (read-only) `EDIR` bit!

It's recommended to use `Bt_CheckPwcError()` before to check counter overrun.

If 32-bit is set for RLT or PWC mode, settings are only done in an even numbered BT instance number. The instance number + 1 is then not available for individual settings anymore! Do not use this "upper" BT instance, because the driver does not check this.

If 32-bit is set for RLT mode, the driver takes care of correct 32-bit reload value setting in `Bt_Init()`. Note that only even numbered BT instances can be used for 32-bit mode and the number + 1 is not available for individual settings anymore.

Note:

For speed reasons this module does not provide an instance data look-up table. The callback pointers are held in individual global variables which are only defined, if a BT instances is actually activated.

If several BTs should be combined by certain BTIOSEL I/O modes, each BT instance must be activated individually by `L3_PERIPHERAL_ENABLE_BTn` with `L3_ON` in *l3_user.h*! Otherwise malfunction of the BT driver may result.

5.3.1 Configuration Structures

5.3.1.1 Base Timer

The BT module uses the following configuration structure of the type `stc_bt_config_t`:

Type	Field	Possible Values	Description
en_bt_mode_t	enMode	BtPwm BtPpg BtRlt BtPwc	Pulse Width Modulation Mode Programmable Pulse Gen. Mode Reload Timer Mode Pulse Width Counter Mode
en_bt_… prescaler_t	enPrescaler	BtExtClkRisingEdge BtExtClkFallingEdge BtExtClkBothEdges BtIntClkDiv1 BtIntClkDiv4 BtIntClkDiv16 BtIntClkDiv128 BtIntClkDiv256 BtIntClkDiv512 BtIntClkDiv1024 BtIntClkDiv2048	Ext. clock, BT triggered by rising edge Ext. clock, BT triggered by falling edge Ext. clock, BT triggered by both edges Int. clock: PCLK Int. clock: PCLK / 4 Int. clock: PCLK / 16 Int. clock: PCLK / 128 Int. clock: PCLK / 256 Int. clock: PCLK / 512 Int. clock: PCLK / 1024 Int. clock: PCLK / 2048
oolean_t	bRestart	TRUE FALSE	Restart by software trig. Or trig. Input No restart possible
oolean_t	bOutputLow	TRUE FALSE	PWM, PPG output masked to low PWM, PPG output as is
oolean_t	bOutput… Invert	TRUE FALSE	Output inverted (high after reset) No output inversion
oolean_t	bOneShot	TRUE FALSE	One shot operation Continuous operation
oolean_t	bTimer32	TRUE FALSE	RLT, PWC in 32 bit mode RLT, PWC in 16 bit mode
en_bt_… trigger_… input_t	enExtTrigger… Mode	BtTriggerDisable BtTriggerRisingEdge BtTriggerFallingEdge BtTriggerBothEdges BtMeasureHighPulse BtMeasureLowPulse BtMeasureRisingEdges BtMeasureFallingEdges BtMeasureAllEdges	Trigger input disabled (PWM, PPG, RLT) Trigger by rising edge (PWM, PPG, RLT) Trigger by falling edge (PWM, PPG, RLT) Trigger by both edges (PWM, PPG, RLT) Measurement of high pulse (PWC) Measurement of low pulse (PWC) Measurement between rising edges (PWC) Measurement between falling edges (PWC) Measurement between all edges (PWC)
uint32_t or uint16_t uint16_t	u32Cycle or u16Cycle / u16LowWidth u16Duty / u16HighWidth	- - -	32-Bit RLT reload value or PWM, RLT cycle / PPG low width PWM, RLT duty / PPG high width
oolean_t	bUnderflow… IrqEnable bOverflow… IrqEnable	TRUE FALSE TRUE FALSE	PWM, PPG, RLT: underflow IRQ and callback PWM, PPG, RLT: no underflow IRQ and callb. PWC: overflow IRQ and callback enable PWC: no overflow IRQ and callback
oolean_t	bTrigger… IrqEnable bCompletion… IrqEnable	TRUE FALSE TRUE FALSE	PWM, PPG, RLT: trigger IRQ and callback PWM, PPG, RLT: no trigger IRQ and callb. PWC: completion IRQ and callback enable PWC: no completion IRQ and callback
oolean_t	bDutyIrq…	TRUE	PWM: Enable duty match IRQ and callb.

	Enable	FALSE	PWM: No duty match IRQ and callb.
Func_ptr_t	pfn… Callback0	-	IRQ1 callback: • PWM: Timer start trigger detection • PPG: Timer start trigger detection • RLT: Timer start trigger detection • PWC: Completion of measurement detection
func_ptr_t	pfn… Callback1	-	IRQ0 callback: • PWM: Underflow detection • PPG: Underflow detection • RLT: Underflow detection • PWC: Overflow detection
func_ptr_t	pfn… Callback2	-	IRQ0 callback: • PWM: Match in duty detection

5.3.1.2 Base Timer I/O Selectors

The Base Timer I/O Selectors have the following configuration structure of the type `bt_iosel_config_t`:

Type	Field	Possible Values	Description
en_bt_iosel_… instance_t	enBtInstance	BtSel01 BtSel23 BtSel45 BtSel67	Selector for BT0 and BT1 Selector for BT2 and BT3 Selector for BT4 and BT5 Selector for BT6 and BT7
en_bt_iosel_… mode_t	enMode	BtMode0 BtMode1 BtMode2 BtMode3 BtMode4 BtMode5 BtMode6 BtMode7 BtMode8	Standard 16-bit timer mode Timer full mode Shared external trigger mode Shared channel signal trigger mode Timer start/stop mode SW-based simultaneous startup mode SW-based startup and timer start/stop mode Timer start mode Shared ch. Signal timer and timer start/stop

5.3.2 Bt_Init ()

Initializes a BT instance according to its configuration. It does not enable the operation. Use `Bt_Enable()` for enabling a timer after `Bt_Init()`.

Prototype	

`en_result_t Bt_Init( stc_btn_t* pstcBt,`
`stc_bt_config_t* pstcConfig )`	

Parameter Name	**Description**
[in] pstcBt	BT instance pointer
[in] pstcConfig	Pointer to BT instance configuration
Return Values	**Description**
Ok	Initialization successfully done.
ErrorInvalidParameter	• pstcConfig == NULL • 32 Bit mode set to odd BT instance number • Other invalid configuration settings

5.3.3 Bt_DeInit()

This function de-inits a BT instance. Because all Base Timers share one IRQ this interrupt can be disabled by parameter. This parameter should be set to `TRUE`, when the last BT instance is de-initialized.

With `bDisableBtInterrupts == TRUE` all shared BT interrupts can be disabled.

Prototype	
`en_result_t Bt_DeInit( stc_btn_t* pstcBt, boolean_t bDisableBtInterrupts )`	
Parameter Name	**Description**
`[in] pstcBt`	BT instance pointer
`[in] bDisableBt… Interrupts`	`TRUE`: All shared BT interrupts are disabled `FALSE`: The interrupt enable is not touched
Return Values	**Description**
`Ok`	De-Initialization successfully done.

5.3.4 Bt_Enable()

This function enables a BT instance. `Bt_Init()` has to be called before.

Prototype	
`en_result_t Bt_Enable( stc_btn_t* pstcBt )`	
Parameter Name	**Description**
`[in] pstcBt`	BT instance pointer
Return Values	**Description**
`Ok`	BT instance successfully enabled.

5.3.5 Bt_Disable()

This function disables a BT instance.

Prototype	
`en_result_t Bt_Disable( stc_btn_t* pstcBt )`	
Parameter Name	**Description**
`[in] pstcBt`	BT instance pointer
Return Values	**Description**
`Ok`	BT instance successfully disabled.

5.3.6 Bt_Trigger()

This function triggers a BT instance by software. This function has no effect, if the instance is configured as PWC.

Prototype	
`en_result_t Bt_Trigger(stc_btn_t* pstcBt)`	
Parameter Name	**Description**
`[in] pstcBt`	BT instance pointer
Return Values	**Description**
`Ok`	BT instance successfully triggered. (PWC mode is not checked!)

5.3.7 Bt_ResetIoSel()

This function resets all Base Timer's I/O Selectors.

Prototype	
`en_result_t Bt_ResetIoSel(void)`	
Return Values	**Description**
`Ok`	All I/O Selectors successfully reset to initial values ('`0`')

5.3.8 Bt_InitIoSel()

This function initializes the BT I/O Selectors by their configuration.

Prototype	
`en_result_t Bt_InitIoSel(stc_bt_iosel_config_t* pstcConfig)`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to BT IOSEL configuration structure
Return Values	**Description**
`Ok`	Initialization successfully done
`ErrorInvalidParameter`	• `pstcConfig == NULL` • Other invalid configuration settings

5.3.9 Bt_DeInitIoSel()

This function de-initializes one or more BT I/O Selectors by their configuration.

Prototype	
`en_result_t Bt_DeInitIoSel(stc_bt_iosel_config_t* pstcConfig)`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to BT IOSEL configuration structure
Return Values	**Description**
`Ok`	De-Initialization successfully done
`ErrorInvalidParameter`	• `pstcConfig == NULL` • Other invalid configuration settings

5.3.10 Bt_TriggerInstances()

This function triggers a BT instance by the software-based simultaneous startup register.

Note:

Do not use this function unless the selected BT instances are set to either of the following modes:

- I/O mode 5 (Software-based simultaneous startup mode)
- I/O mode 6 (Software-based startup and timer start/stop mode) (Even channels only)

For the BT instance started up by using this method, select the rising edge as a trigger input edge using `en_bt_trigger_input_t::BtTriggerRisingEdge` configuration.

Prototype	
`en_result_t Bt_TriggerInstances(uint16_t u16TriggerBtInstance)`	
Parameter Name	**Description**
`[in] u16TriggerBtInstance`	Bit pattern of `BTSSSR_BT`*n* definition in *bt.h*
Return Values	**Description**
`Ok`	Trigger done. (Correct IOSEL mode is not checked!)

5.3.11 Bt_ReadTimer16()

This function returns the recent 16-Bit timer value (`TMR`).

Prototype	
`uint16_t Bt_ReadTimer16(stc_btn_t* pstcBt)`	
Parameter Name	**Description**
`[in] pstcBt`	BT instance pointer
Return Values	**Description**
`uint16_t`	Recent 16-Bit timer value (`TMR`)

5.3.12 Bt_CheckPwcError()

This function reads the `ERR` bit of the `STC` register. It assumes that the BT is set to PWC mode. Otherwise the return value is regardless.

This bit indicates that the next measurement has been completed in continuous measurement mode before the measurement result is read from the `DTBF` register. In this case the result of the previous measurement in the `DTBF` register is replaced by that of the next measurement. The measurement is continued regardless of the `ERR` bit value.

The `ERR` bit is cleared by reading the measurement result (reading `DTBF` or by calling `Bt_ReadDataBuffer16()`).

Prototype	
`en_result_t Bt_CheckPwcError(stc_btn_t* pstcBt)`	
Parameter Name	**Description**
`[in] pstcBt`	BT instance pointer
Return Values	**Description**
`Ok`	`ERR` bit is not set
`Error`	`ERR` bit is set

5.3.13 Bt_ReadDataBuffer16()

This function reads out the 16-bit PWC result counter (if set to 16-Bit mode) and clear a possible `ERR` bit.

Note:

This function must be called in the callback function to clear the interrupt request!

Prototype	
`uint16_t Bt_ReadDataBuffer16(stc_btn_t* pstcBt)`	
Parameter Name	**Description**
`[in] pstcBt`	BT instance pointer
Return Values	**Description**
`Ok`	Recent PWC counter value (result).

5.3.14 Bt_ReadDataBuffer32()

This function reads out the 32-bit PWC result counter (if set to 32-Bit mode) and clear a possible `ERR` bit.

Note:

This function must be called in the callback function to clear the interrupt request!

Prototype	
`uint32_t Bt_ReadDataBuffer16(stc_btn_t* pstcBt)`	
Parameter Name	**Description**
`[in] pstcBt`	BT instance pointer
Return Values	**Description**
`Ok`	Recent PWC counter value (result).

5.3.15 Bt_PwmSetCycle()

This function sets the cycle count for the PWM mode of the Base Timer.

Note:

This function does not check, whether the Base Timer is actual in PWM mode for speed reasons.

Prototype	
`en_result_t Bt_PwmSetCycle(stc_btn_t* pstcBt, uint16_t u16Cycle)`	
Parameter Name	**Description**
`[in] pstcBt`	BT instance pointer
`[in] u16Cycle`	New cycle count value
Return Values	**Description**
`Ok`	New cycle count value set.

5.3.16 Bt_PwmSetDuty()

This function sets the duty cycle count for the PWM mode of the Base Timer.

Note:

This function does not check, whether the Base Timer is actual in PWM mode for speed reasons.

Prototype	
`en_result_t Bt_PwmSetDuty(stc_btn_t* pstcBt, uint16_t u16Duty)`	
Parameter Name	**Description**
`[in] pstcBt`	BT instance pointer
`[in] u16Duty`	New duty cycle count value
Return Values	**Description**
`Ok`	New duty cycle count value set.

5.3.17 Bt_PpgSetLowWidth()

This function sets the low width (16 Bit) for the PPG mode of the Base Timer.

Note:

This function does not check, whether the Base Timer is actual in PPG mode for speed reasons.

Prototype	
`en_result_t Bt_PpgSetLowWidth(stc_btn_t* pstcBt, uint16_t u16LowWdith)`	
Parameter Name	**Description**
`[in] pstcBt`	BT instance pointer
`[in] u16LowWidth`	New low width value
Return Values	**Description**
`uint32_t`	New low width value set.

5.3.18 Bt_PwmSetDuty()

This function sets the high width (16 Bit) for the PPG mode of the Base Timer.

Note:

This function does not check, whether the Base Timer is actual in PPG mode for speed reasons.

Prototype	
`en_result__t Bt_PpgHighWidth(stc_btn_t* pstcBt, uint16_t u16HighWidth)`	
Parameter Name	**Description**
`[in] pstcBt`	BT instance pointer
`[in] u16Duty`	New duty cycle count value
Return Values	**Description**
`uint32_t`	New duty cycle count value set.

5.3.19 Callback0()

This function is called, if `stc_bt_config_t::pfnCallback0` is defined and the according interrupt is enabled. It is called by IRQ1.

Callback Function
`void Callback0( void )`

5.3.20 Callback1()

This function is called, if `stc_bt_config_t::pfnCallback1` is defined and the according interrupt is enabled. It is called by IRQ0.

Callback Function
`void Callback1( void )`

5.3.21 Callback2()

This function is called, if `stc_bt_config_t::pfnCallback2` is defined and the according interrupt is enabled. It is called by IRQ0, Match in duty in PWM function.

Callback Function
`void Callback2( void )`

5.3.22 Example Code

The following code excerpts shows some applications of using the BT driver.

5.3.22.1 BT in PWM Mode

The following code excerpt shows, how to use a BT instance as a Pulse Width Modulator.

```
#include "bt.h"

void Bt0CallbackTg(void) // trigger callback of PWM
{
  // Do something ...
}

void Bt0CallbackUf(void) // underflow detection of PWM
{
  // Do something ...
}

void Bt0CallbackDc(void) // duty cycle of PWM
{
  // Do something ...
}

function
{
  stc_bt_config_t stcBtConfig0;

  L3_ZERO_STRUCT(stcBtConfig0);

  stcBtConfig0.enMode               = BtPwm;
  stcBtConfig0.enPrescaler          = BtIntClkDiv16;
  stcBtConfig0.bRestart             = TRUE;
  stcBtConfig0.bOutputLow           = FALSE;
  stcBtConfig0.bOutputInvert        = FALSE;
  stcBtConfig0.bTimer32             = FALSE;   // PWM has no 32 bit mode!
  stcBtConfig0.bOneShot             = FALSE;
  stcBtConfig0.enExtTriggerMode     = BtTriggerDisable;
  stcBtConfig0.u16Cycle             = 0xC000;  // Just a random value ...
  stcBtConfig0.u16Duty              = 0x8000;  // Just a random value ...
  stcBtConfig0.bTriggerIrqEnable    = TRUE;
  stcBtConfig0.bUnderflowIrqEnable = TRUE;
  stcBtConfig0.bDutyIrqEnable       = TRUE;
  stcBtConfig0.pfnCallback0 = &Bt0CallbackTg; // trigger irq callback
  stcBtConfig0.pfnCallback1 = &Bt0CallbackUf; // underflow irq callback
  stcBtConfig0.pfnCallback2 = &Bt0CallbackDc; // duty irq callback

  Bt_ResetIoSel();

  if (Ok == Bt_Init((stc_btn_t*)&BT0, &stcBtConfig0))  // Init BT instance
  {
    Bt_Enable((stc_btn_t*)&BT0);     // Enable BT instance
    Bt_Trigger((stc_btn_t*)&BT0);    //   ... and start it ...

    while(1);
  }
}
```

5.3.22.2 BT in RLT 32-Bit Mode

The following code excerpt shows, how to use a BT instance as a Reload Timer in 32-Bit mode.

```
#include "bt.h"

void Bt0CallbackUf(void) // underflow detection of RLT
{
  // Do something ...
}

function
{
  stc_bt_config_t stcBtConfig0;

  L3_ZERO_STRUCT(stcBtConfig0);

  stcBtConfig0.enMode               = BtRlt;
  stcBtConfig0.enPrescaler          = BtIntClkDiv1;
  stcBtConfig0.bRestart             = TRUE;
  stcBtConfig0.bOutputLow           = FALSE;
  stcBtConfig0.bOutputInvert        = FALSE;
  stcBtConfig0.bTimer32             = TRUE;        // 32-bit mode
  stcBtConfig0.bOneShot             = FALSE;
  stcBtConfig0.enExtTriggerMode     = BtTriggerDisable;
  stcBtConfig0.u32Cycle             = 0x00024000;  // Just a random value ...
  stcBtConfig0.bTriggerIrqEnable    = FALSE;
  stcBtConfig0.bUnderflowIrqEnable = TRUE;
  stcBtConfig0.pfnCallback1 = &Bt0CallbackUf;      // underflow irq callback

  Bt_ResetIoSel();

  if (Ok == Bt_Init((stc_btn_t*)&BT0, &stcBtConfig0))
  {
    Bt_Enable((stc_btn_t*)&BT0);     // Enable BT instance
    Bt_Trigger((stc_btn_t*)&BT0);    //   ... and start it ...

    while(1);
  }
}
```

5.3.22.3 BT in PPG Mode with simultaneous start of two PPGs

The following code excerpt shows, how to use a BT instances acting in PWC 32-Bit mode and started by simultaneous software trigger.

```
#include "bt.h"

void Bt0CallbackUf(void) // underflow detection PPG0
{
  // Do something ...
}

void Bt1CallbackUf(void) // underflow detection PPG1
{
  // Do something ...
}

function
{
  stc_bt_config_t stcBtConfig0;
  stc_bt_config_t stcBtConfig1;

  L3_ZERO_STRUCT(stcBtConfig0);
  L3_ZERO_STRUCT(stcBtConfig1);

  stcBtConfig0.enMode                = BtPpg;
  stcBtConfig0.enPrescaler           = BtIntClkDiv4;
  stcBtConfig0.bRestart              = TRUE;
  stcBtConfig0.bOutputLow            = FALSE;
  stcBtConfig0.bOutputInvert         = FALSE;
  stcBtConfig0.bTimer32              = FALSE;    // PPG has no 32 bit mode!
  stcBtConfig0.bOneShot              = FALSE;
  stcBtConfig0.enExtTriggerMode      = BtTriggerRisingEdge;
  stcBtConfig0.u16LowWidth           = 0x0100;   // Just a random value ...
  stcBtConfig0.u16HighWidth          = 0x0200;   // Just a random value ...
  stcBtConfig0.bTriggerIrqEnable     = FALSE;
  stcBtConfig0.bUnderflowIrqEnable = TRUE;
  stcBtConfig0.bDutyIrqEnable        = TRUE;
  stcBtConfig0.pfnCallback1 = &Bt0CallbackUf;   // underflow irq callback

  stcBtConfig1.enMode                = BtPpg;
  stcBtConfig1.enPrescaler           = BtIntClkDiv4;
  stcBtConfig1.bRestart              = TRUE;
  stcBtConfig1.bOutputLow            = FALSE;
  stcBtConfig1.bOutputInvert         = FALSE;
  stcBtConfig1.bTimer32              = FALSE;    // PPG has no 32 bit mode!
  stcBtConfig1.bOneShot              = FALSE;
  stcBtConfig1.enExtTriggerMode      = BtTriggerRisingEdge;
  stcBtConfig1.u16LowWidth           = 0x0200;   // Just a random value ...
  stcBtConfig1.u16HighWidth          = 0x0100;   // Just a random value ...
  stcBtConfig1.bTriggerIrqEnable     = FALSE;
  stcBtConfig1.bUnderflowIrqEnable = TRUE;
  stcBtConfig1.bDutyIrqEnable        = TRUE;
  stcBtConfig1.pfnCallback1 = &Bt1CallbackUf;   // underflow irq callback

  stcBtIoselConfig.enBtInstance = BtSel01; // Combine BT0 and BT1
  stcBtIoselConfig.enMode       = BtMode5; // sw-based simultaneous startup

  Bt_ResetIoSel();

  if (Ok == Bt_Init((stc_btn_t*)&BT0, &stcBtConfig0))
  {
    if (Ok == Bt_Init((stc_btn_t*)&BT1, &stcBtConfig1))
    {
      Bt_Enable((stc_btn_t*)&BT0);
      Bt_Enable((stc_btn_t*)&BT1);

      // Set I/O mode 5 (sw-based simultaneous startup) for BT0 and BT1
      Bt_InitIoSel(&stcBtIoselConfig);
```

▼

```
▲

            Bt_TriggerInstances(BTSSSR_BT0_BIT | BTSSSR_BT1_BIT);  // SW trigger!

            while(1);
        }
    }
}
```

5.3.22.4 BT in PWC 32-Bit Mode

The following code excerpt shows, how to use a BT instances acting in PWC 32-Bit mode. It measures the pulse width produced by another BT in PWM mode. The input of BT2 has to be connected physically to BT1 output in this case.

```
#include "bt.h"

uint32_t u32Count;
uint8_t  u8PwcComplete = 0;

void Bt2CallbackComplete(void) // completion callback of PWC2
{
  // Only reading the data buffer clears the interrupt request!
  // Expected value: stcBtConfig1.u16Duty + 1 (0x00008013) }
  u32Count = Bt_ReadDataBuffer32((stc_btn_t*)&BT2);

  u8PwcComplete = 1; // end notification ...
}

function
{
  stc_bt_config_t stcBtConfig1;
  stc_bt_config_t stcBtConfig2;

  L3_ZERO_STRUCT(stcBtConfig1);
  L3_ZERO_STRUCT(stcBtConfig2);

  stcBtConfig1.enMode                 = BtPwm;
  stcBtConfig1.enPrescaler            = BtIntClkDiv4;
  stcBtConfig1.bRestart               = FALSE;      // No restart: one-shot mode
  stcBtConfig1.bOutputLow             = FALSE;
  stcBtConfig1.bOutputInvert          = FALSE;
  stcBtConfig1.bTimer32               = FALSE;      // PWM has no 32 bit mode!
  stcBtConfig1.bOneShot               = TRUE;
  stcBtConfig1.enExtTriggerMode       = BtTriggerDisable;
  stcBtConfig1.u16Cycle               = 0xC000;     // low pulse start phase
  stcBtConfig1.u16Duty                = 0x8012;     // high pulse to be measured
  stcBtConfig1.bTriggerIrqEnable      = FALSE;
  stcBtConfig1.bUnderflowIrqEnable    = FALSE;
  stcBtConfig1.bDutyIrqEnable         = FALSE;

  stcBtConfig2.enMode                 = BtPwc;
  stcBtConfig2.enPrescaler            = BtIntClkDiv4;
  stcBtConfig2.bRestart               = TRUE;
  stcBtConfig2.bOutputLow             = FALSE;
  stcBtConfig2.bOutputInvert          = FALSE;
  stcBtConfig2.bTimer32               = TRUE;
  stcBtConfig2.bOneShot               = TRUE;       // only one measurement
  stcBtConfig2.enExtTriggerMode       = BtMeasureHighPulse;
  stcBtConfig2.bCompletionIrqEnable = TRUE;
  stcBtConfig2.pfnCallback0 = &Bt2CallbackComplete; // completion callback

  if (Ok == Bt_Init((stc_btn_t*)&BT1, &stcBtConfig1))
  {
    if (Ok == Bt_Init((stc_btn_t*)&BT2, &stcBtConfig2))
    {
      u8PwcComplete = 0;

      Bt_Enable((stc_btn_t*)&BT2);    // Enable PWC

      Bt_Enable((stc_btn_t*)&BT1);    // Start single-shot PWM pulse
      Bt_Trigger((stc_btn_t*)&BT1);

      while(u8PwcComplete == 0);      // wait for PWC pulse measurement complete

      while(1);
    }
  }
}
```

5.4 (CAN) Controller Area Network

Type Definition	`stc_cann_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!

5.4.1 Configuration Structures

5.4.1.1 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 5.4.1.1.1	See 5.4.1.1.1

5.4.1.1.1 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`

			(see define section in *can.h*)
`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.

5.4.2 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`	• `pstcConfig == 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

5.4.3 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`	• `pstcConfig == NULL` • `pstcCanInternData == NULL` Invalid or deactivated CAN uint (`L3_PREIPHERAL_ENABLE_CAN`*n*)

5.4.4 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 (`L3_PREIPHERAL_ENABLE_CAN`*n*) • `pstcMsgId == NULL` • `u8MsgBuf` out of range
`ErrorOperationInProgress`	A transmission is pending (either wait or call `Can_ResetMsgBuffer()`).

5.4.5 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 (`L3_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()`).

5.4.6 Can_UpdateAndTransmitFifoMsgBuffer()

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

5.4.7 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* pstcCan, 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 (`L3_PREIPHERAL_ENABLE_CAN`*n*) • `pstcMsgBuffer == NULL` • `u8MsgBuf` out of range

5.4.8 Can_SetReceiveFifoMsgBuffer()

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

5.4.9 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 (`L3_PREIPHERAL_ENABLE_CAN`*n*) • `u8MsgBuf` out of range

5.4.10 CanTxCallback().

Callback Function
`void CanTxCallback( uint8_t u8MsgBuf )`

5.4.11 CanRxCallback()

Callback Function
`void CanRxCallback( uint8_t u8MsgBuf, stc_can_msg_t* pstcRxMsg )`

5.4.12 CanStatusCallback()

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

Callback Function
`void CanStatusCallback( 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.

5.4.13 CanErrorCallback()

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

Callback Function
`void CanErrorInterruptCallback( 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.

5.4.14 Example Code

The following code excerpts shows some applications of using the CAN driver.

5.4.14.1 CAN Configuration and Settings

Assume the following configuration and Settings also for the further example code. Note that it is assumed that the callback functions `CanStatusCb()` and `CanErrorCb()` exist.

```
// Example configuration for following sample code on next pages

    stc_can_config_t    stcCanConfig;     // CAN configuratiion
    stc_can_msg_data_t  stcMsgData;       // CAN message data
    stc_can_msg_id_t    stcMsgId;         // CAN message ID
    stc_can_msg_t       stcMsgBuffer1;    // CAN message buffer 1
    stc_can_msg_t       stcMsgBuffer2;    // CAN message buffer 2

    stc_cann_t*         pstcCan        = NULL; // CAN instance pointer
    stc_can_msg_id_t*   pstcMsgId      = NULL; // Message ID pointer
    stc_can_msg_data_t* pstcMsgData    = NULL; // Message Data pointer
    stc_can_msg_t*      pstcMsgBuffer1 = NULL; // Message Buffer 1 pointer
    stc_can_msg_t*      pstcMsgBuffer2 = NULL; // Message Buffer 2 pointer

    L3_ZERO_STRUCT(stcCanConfig);
    L3_ZERO_STRUCT(stcMsgData);
    L3_ZERO_STRUCT(stcMsgId);
    L3_ZERO_STRUCT(stcMsgBuffer1);
    L3_ZERO_STRUCT(stcMsgBuffer2);

    uint32_t u32MsgIdMask1;            // ID Mask for message 1
    uint32_t u32MsgIdMask2;            // ID Mask for message 2

    pstcCan = (stc_cann_t*)(&CAN0);    // Set CAN instance 0
    pstcMsgId      = &stcMsgId;        // Set Message ID pointer
    pstcMsgData    = &stcMsgData;      // Set Message Data pointer
    pstcMsgBuffer1 = &stcMsgBuffer1;   // Set Message Buffer 1 pointer
    pstcMsgBuffer2 = &stcMsgBuffer2;   // Set Message Buffer 2 pointer

    stcCanConfig.pfnStatusCallback = (can_status_chg_func_ptr_t)CanStatusCb;
    stcCanConfig.pfnErrorCallback = (can_error_func_ptr_t)CanErrorCb;
    stcCanConfig.bDisableAutomaticRetransmission = TRUE;

    // 100k Bit/s
    stcCanConfig.stcBitrate.u8TimeSegment1  = 15;
    stcCanConfig.stcBitrate.u8TimeSegment2  = 7;
    stcCanConfig.stcBitrate.u8SyncJumpWidth = 4;
    stcCanConfig.stcBitrate.u16Prescaler    = 7;
    stcCanConfig.stcBitrate.enCanPrescaler = CanPreDiv15; // 80 MHz / 5 = 16 Mhz

    // TX buffer settings
    stcMsgId.u32Identifier      = 0x12;
    stcMsgId.bExtended          = FALSE ;
    stcMsgData.u8DataLengthCode = 8;

    // Message data (no special values, just random ...)
    stcMsgData.au8Data[0] = 0x70;
    stcMsgData.au8Data[1] = 0x61;
    stcMsgData.au8Data[2] = 0x52;
    stcMsgData.au8Data[3] = 0x43;
    stcMsgData.au8Data[4] = 0x34;
    stcMsgData.au8Data[5] = 0x25;
    stcMsgData.au8Data[6] = 0x16;
    stcMsgData.au8Data[7] = 0x07;

    // RX buffer 1 settings (assumed auto-response from other CAN node)
    stcMsgBuffer1.stcIdentifier.u32Identifier = 0x0A;
    stcMsgBuffer1.stcIdentifier.bExtended = FALSE;
    u32MsgIdMask1 = 0x00000000;

    // RX buffer 2 settings (assumed response from other CAN node)
    stcMsgBuffer2.stcIdentifier.u32Identifier = 0x01;
    stcMsgBuffer2.stcIdentifier.bExtended = FALSE;
    u32MsgIdMask2 = 0x00000000;
```

5.4.14.2 Can Initialization and Communication

The following code shows an example of CAN initialization and communication with another CAN node, using the configuration from above. Also here the following callback functions are assumed to be existing: `CanTxCompleteCb()`, `CanStatusCb()`, and `CanErrorCb()`.

This code assumes a configured and connected external CAN node, which is capable to send an auto-respond message and a dedicated message by a condition 1 and 2.

11-Bit CAN-IDs are used.

After `Can_Init()` to 100k Bit/s the following message is sent by the MCU:

- Tx: D:012h L:8h 70h 61h 52h 43h 34h 25h 16h 70h C:2669h N EF
- Rx: D:012h L:8h 70h 61h 52h 43h 34h 25h 16h 70h C:2669h A EF

Auto-Response to ID 012h from external CAN node:

- Rx: ID:00Ah L:1h 71h C:3FB5h A EF

Message from external CAN node condition 1:

- Rx: ID:001h L:2h 42h 43h C:03Eah A EF

Message from external CAN node condition 2:

- Rx: ID:001h L:1h 43h C:04DBh A EF

```
#include "can.h"

uint8_t u8ReceiveCounter = 0;

void CanTxCompleteCb(uint8_t u8MsgBuf){}

void CanRxReceivedCb(uint8_t u8MsgBuf, stc_can_msg_t* pstcRxMsg)
{
  u8ReceiveCounter++;
}

function
{
  // Place example configuration from above here

  if (Ok == Can_Init(pstcCan, &stcCanConfig))
  {
    Can_SetTransmitMsgBuffer(pstcCan,
                             1,
                             pstcMsgId,
                             (can_tx_msg_func_ptr_t)CanTxCompleteCb);

    Can_SetReceiveMsgBuffer(pstcCan,
                            2,
                            pstcMsgBuffer1,
                            u32MsgIdMask1,
                            (can_rx_msg_func_ptr_t)CanRxReceivedCb);

    Can_SetReceiveMsgBuffer(pstcCan,
                            3,
                            pstcMsgBuffer2,
                            u32MsgIdMask2,
                            (can_rx_msg_func_ptr_t)CanRxReceivedCb);

    Can_UpdateAndTransmitMsgBuffer(pstcCan,
                                   1,
                                   pstcMsgData,
                                   CanImmediateTransmit);

    // Wait for receive callback
    // Rx: ID:00Ah L:1h 71h C:77ABh A EF IM (ID from settings above)
    while(u8ReceiveCounter < 2);

    if ((stcMsgBuffer2.stcData.au8Data[0] == 0x42) &&
        (stcMsgBuffer2.stcData.au8Data[1] == 0x43) &&
        (stcMsgBuffer2.stcData.u8DataLengthCode == 2)
       )
    {
      // Code for CAN node reception condition 1
      // Rx: ID:001h L:2h 42h 43h C:03EAh A EF (ID from settings above)
    }
    else if ((stcMsgBuffer2.stcData.au8Data[0] == 0x43) &&
             (stcMsgBuffer2.stcData.u8DataLengthCode == 1)
            )
    {
      // Code for CAN node reception condition 2
      // Rx: ID:001h L:1h 43h C:04DBh A EF (ID from settings above)
    }
    else
    {
      // Unknown message was received ...
    }
  }
}
```

5.5 (CLK) Clock Module

Type Definition	–
Configuration Type	`stc_clk_config_t`
Address Operator	–

This driver manages the FM3 clock settings for:

- Clock Source (Main, Sub, HS-CR, LS-CR, PLL)
- Clock Mode (Run, Sleep, Timer, Stop)
- Bus dividers (HCLK, PCLK0, PCLK1, PCLK2)
- Stabilization Wait Times (Main, Sub, PLL)

Also polling routines for the oscillation stabilization wait are provided, e.g. `Clk_WaitForMainOscillator()`. If this waiting/polling is not wanted in the user application, there are also function, which return the current state of the stability of a clock, e.g. `Clk_MainOscillatorReady()`.

For Clock Mode transition a hook function is called (if specified) for making individual power saving settings between setting `SLEEPDEEP` and the final `WFI` instruction (Timer-, Sleep-, Stop modes).

Each wait function has at lest a parameter for time out counting. The value of this parameter depends on the user's wait time settings. Additionally these functions call the library's `L3_WAIT_LOOP_HOOK()` function.

Notes:

- After each Clock Source change `Clk_WaitForClockSourceReady()` with the clock source enumerator `en_clk_source_t` and a time out count value as parameters has to be called, if a previous waiting for stabilization was not performed.
- If the used device supports RTC and the RTC low power modes should be used, switch `L3_RTC_AVAILABLE` to `L3_ON` in *l3_user.h*.
- The configuration `en_clk_pll_src_t::PllSrcClkHc` and the function `Clk_SetPllSource()` can only be used, if the used device supports HS-RC-Clock as PLL input clock source! This driver does not check for availability!
- Because the Clock interrupts are shared with other interrupts (e.g. Watch Counter), `Clk_SetIrq()` has an argument, whether to touch the NVIC registers or not to prevent interference with other driver part.
- The CLK driver functions may replace the `System_Init()` clock settings in *system_mb9[ab]xyz.c*. Set the definition for `CLOCK_SETUP` to `0` in *system_mb9[ab] xyz.h* in this case.
- The availability of PLL connected to HS-CR clock (by `PINC` bit) is not checked by the L3. Please refer to device's datasheet.

5.5.1 Configuration Structure

The CLK module uses the following configuration structure of the type `stc_clk_config_t`:

Type	Field	Possible Values	Description

`en_clk_source_t`	`enSource`	`ClkMain`
		`ClkSub`
		`ClkHsCr`
		`ClkLsCr`
			Main Clock Oscillator
			Sub Clock Oscillator
			High-Speed CR Clock Oscillator
			Low-Speed CR Clock Oscillator

		`ClkPll`	PLL Clock
`oolean_t`	`bEnablePll`	`TRUE` `FALSE`	Enable PLL Do not enable PLL
`oolean_t`	`bEnableMainClock`	`TRUE` `FALSE`	Enable Main Clock Do not enable Main Clock
`oolean_t`	`bEnableSubClock`	`TRUE` `FALSE`	Enable Sub Clock Do not enable Sub Clock

`en_clk_mode_t`	`enMode`	`ClkRun`
		`ClkSleep`
		`ClkTimer`
		`ClkStop`
		`ClkDeepStop`
			Run modes
			Sleep modes
			Timer modes
			Stop mode
			Deep standby mode

		If RTC available on device:	
		`ClkRtc` `ClkDeepRtc`	RTC mode Deep standby RTC mode
`oolean_t`	`bLpmPortHiZState`	`TRUE` `FALSE`	High-z pin state in timer and stop mode Pins not touched in timer and stop mode

`en_clk_apb0div_t`	`enAPB0Div`	`Apb0Div1`
		`Apb0Div2`
		`Apb0Div4`
		`Apb0Div8`
			PCLK0 Division 1/1
			PCLK0 Division ½
			PCLK0 Division ¼
			PCLK0 Division 1/8

`en_clk_apb1div_t`	`enAPB1Div`	`Apb1Div1`
		`Apb1Div2`
		`Apb1Div4`
		`Apb1Div8`
			PCLK1 Division 1/1
			PCLK1 Division ½
			PCLK1 Division ¼
			PCLK1 Division 1/8

`en_clk_apb2div_t`	`enAPB2Div`	`Apb2Div1`
		`Apb2Div2`
		`Apb2Div4`
		`Apb2Div8`
			PCLK2 Division 1/1
			PCLK2 Division ½
			PCLK2 Division ¼
			PCLK2 Division 1/8

`boolean_t`	`bAPB1Disable`	`TRUE` `FALSE`	Disables APB1 regardless of settings APB1 enabled
`boolean_t`	`bAPB2Disable`	`TRUE` `FALSE`	Disables APB2 regardless of settings APB2 enabled

`en_clk_scowaittime_t`	`enSCOWaitTime`[1]	`ScoWaitExp10`
		`ScoWaitExp11`
		. . .
		`ScoWaitExp17`
			Sub Clock wait time $2^{10}/f_{CLK} \rightarrow$ ~10.3 ms
			Sub Clock wait time $2^{11}/f_{CLK} \rightarrow$ ~20.5 ms
			. . .
			Sub Clock wait time $2^{17}/f_{CLK} \rightarrow$ ~1.31 s

`en_clk_mcowaittime_t`	`enMCOWaitTime`[1]	`McoWaitExp11`
		`McoWaitExp15`
		. . .
		`McoWaitExp123`
			Main Clock wait time $2^{1}/f_{CLK} \rightarrow$ ~500 ns
			Main Clock wait time $2^{5}/f_{CLK} \rightarrow$ ~8 µs
			. . .
			Main Clock wait time $2^{23}/f_{CLK} \rightarrow$ ~2.0 s

`en_clk_pllowaittime_t`	`enPLLOWaitTime`[1]	`PlloWaitExp19`
		`PlloWaitExp110`
		. . .
		`McoWaitExp116`
			PLL Clock wait time $2^{9}f_{CLK} \rightarrow$ ~128 µs
			PLL Clock wait time $2^{10}/f_{CLK} \rightarrow$ ~256 µs
			. . .
			PLL Clock wait time $2^{16}/f_{CLK} \rightarrow$ ~26.4 ms

`uint8_t`	`u8PllK`[2]	`0 ... `K_{max}	PLL K value
`uint8_t`	`u8PllM`[2]	`0 ... `M_{max}	PLL M value
`uint8_t`	`u8PllN`[2]	`0 ... `N_{max}	PLL N value
`en_clk_pll_src_t`	`enPllSource`[3]	`PllSrcClkMo` `PllSrcClkHc`	PLL clock from Main oscillator PLL clock from HC-RC oscillator
`boolean_t`	`bPllIrq`	`TRUE` `FALSE`	Interrupt after PLL stabilization wait time No interrupt
`boolean_t`	`bScoIrq`	`TRUE` `FALSE`	Interrupt after sub clock stab. Wait time No interrupt
`oolean_t`	`bMcoIrq`	`TRUE` `FALSE`	Interrupt after main clock stab. Wait time No interrupt
`func_ptr_t`	`pfnPllStabCb`	-	Pointer to PLL stabilization callback

			function
func_ptr_t	pfnScoStabCb	-	Pointer to sub clock stabilization callback function
func_ptr_t	pfnMcoStabCb	-	Pointer to main clock stabilization callback function
func_ptr_t	pfnHook	-	Hook call for low power modes (called between SLEEPDEEP = 1 and WFI)

[1] See peripheral manual for exact possible wait time values
[2] See peripheral manual and data sheet for exact possible PLL K, M, N values
[3] Check whether the device supports HS-RC clock source for PLL (PINC bit of PSW_TMR)

5.5.2 Clk_GetParameters ()

This function sets the *current* elements of the configuration according the 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 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_GetParameters(stc_clk_config_t* pstcConfig)	
Parameter Name	**Description**
[out] pstcConfig	Clock configuration parameters
Return Values	**Description**
Ok	Clock configuration has been set into pstcConfig
ErrorInvalidParameter	pstcConfig == NULL
ErrorInvalidMode	Illegal clock mode has been detected

5.5.3 Clk_SetDividers ()

This function sets the clock dividers.

Note: 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.).

Also 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_config_t* pstcConfig)	
Parameter Name	**Description**
[in] pstcConfig	Clock configuration parameters
Return Values	**Description**
Ok	Dividers have been set
ErrorInvalidParameter	• pstcDt == NULL • Illegal divider setting

5.5.4 Clk_SetStabilizationWaitTimes ()

This function sets the stabilization wait times.

Prototype	
`en_result_t Clk_GetParameters( stc_clk_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcConfig`	Clock configuration parameters
Return Values	**Description**
`Ok`	Stabilization wait times have been set
`ErrorInvalidParameter`	`pstcConfig == NULL`
`ErrorInvalidMode`	• `pstcDt == NULL` • Illegal timing setting

5.5.5 Clk_WaitForMainOscillator ()

This function waits for the Main Oscillator stabilization via polling. `L3_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
`ErrorTimeout`	Main Clock not stabilized after timeout count

5.5.6 Clk_WaitForSubOscillator ()

This function waits for the Sub Oscillator stabilization via polling. `L3_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`	Sub Clock stabilized
`ErrorTimeout`	Sub Clock not stabilized after timeout count

5.5.7 Clk_WaitForPllOscillator ()

This function waits for the PLL Oscillator stabilization via polling. `L3_WAIT_LOOP_HOOK()` is called during polling. It should be called, if the system needs a stable PLL clock (i.e. for communication).

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
`ErrorTimeout`	PLL Clock not stabilized after timeout count

5.5.8 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. `L3_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`	PLL Clock stabilized
`ErrorTimeout`	Clock Source not ready within time out count
`ErrorInvalidParameter`	Not a valid Clock Mode to be checked

5.5.9 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

5.5.10 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

5.5.11 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

5.5.12 Clk_SetSource ()

This function sets the clock source and performs transition, if wanted. Illegal transitions (e.g. changing PLL settings or switch to PLL from other modes than Main Clock) are checked and corresponding error values are returned.

Precondition: `Clk_SetStabilizationWaitTimes()` has to be called before.

Prototype	
`en_result_t Clk_SetSource( stc_clk_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcConfig`	Clock configuration parameters
Return Values	**Description**
`Ok`	Clock source set
`ErrorInvalidParameter`	`pstcConfig == NULL`
`ErrorInvalidMode`	Clock setting not possible

5.5.13 Clk_SetPllSource ()

This function sets the PLL input clock source, if the device supports HS-RC-Clock input for PLL! This function only works, if PLL is disabled.

Attention: The L3 itself does not check the `PINC` bit availability!

Prototype	
`en_result_t Clk_SetPllSource( stc_clk_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcConfig`	Clock configuration parameters
Return Values	**Description**
`Ok`	PLL Clock source set
`ErrorInvalidParameter`	`pstcConfig == NULL` or Illegal mode
`ErrorInvalidMode`	Clock setting not possible (PLL not disabled)

5.5.14 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_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcConfig`	Clock configuration parameters
Return Values	**Description**
`Ok`	PLL Clock source set
`ErrorInvalidParameter`	`pstcConfig == NULL` or Illegal mode

5.5.15 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
`ErrorInvalidMode`	Sub Clock not enabled (e.g. not existing)

5.5.16 Clk_DisableSubClock ()

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

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

5.5.17 Clk_EnableMainClock ()

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

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

5.5.18 Clk_DisableMainClock ()

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

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

5.5.19 Clk_SetIrq ()

This function enables or disables the clock stabilization interrupts according configuration.

Note: Because of shared vectors for clock interrupts, the parameter `bTouchNvic` defines, if the NVIC registers shall be touched or not.

Prototype	
`en_result_t Clk_SetIrq( stc_clk_config_t* pstcConfig,` `oolean_t bTouchNvic )`	
Parameter Name	**Description**
`[in] pstcConfig`	Clock configuration parameters
`[in] bTouchNvic`	`TRUE`: Touch NVIC registers at function call `FALSE`: Do not touch NVIC registers at function call
Return Values	**Description**
`Ok`	Interrupts have been set
`ErrorInvalidParameter`	`pstcConfig == NULL`

5.5.20 PllStabilizedCallback()

This function is called, if `stc_clk_config_t::pfnPllStabCb` is defined and the according interrupt is enabled.

Callback Function
`void PllStabilizedCallback( void )`

5.5.21 SubClockStabilizedCallback()

This function is called, if `stc_clk_config_t::pfnScoStabCb` is defined and the according interrupt is enabled.

Callback Function
`void SubClockStabilizedCallback( void )`

5.5.22 MainClockStabilizedCallback()

This function is called, if `stc_clk_config_t::pfnMcoStabCb`is defined and the according interrupt is enabled.

Callback Function
`void MainClockStabilizedCallback( void )`

5.5.23 HookFunction()

This function is called between `SLEEPDEEP = 1` and `WFI` , if `stc_clk_config_t::pfnHook` defined.

Hook Function
`void HookFunction( void )`

5.5.24 Example Code

The following code excerpts shows some applications of using the clock driver.

5.5.24.1 Clock Configuration

Assume the following example configuration for the examples in the following sub chapters. The PLL values were chosen for a MB9BF50x device.

```
#include "clk.h"

void ClkHookCallback(void)
{
  // Some action ...
}

function
{
  stc_clk_config_t  stcClkConfig;

  L3_ZERO_STRUCT(stcClkConfig);

  stcClkConfig.enSource          = ClkPll;
  stcClkConfig.bEnablePll        = TRUE;
  stcClkConfig.bEnableMainClock = TRUE;
  stcClkConfig.bEnableSubClock  = FALSE;
  stcClkConfig.enMode            = ClkRun;
  stcClkConfig.bLpmPortHiZState = FALSE;
  stcClkConfig.enBaseClkDiv     = BaseClkDiv1;
  stcClkConfig.enAPB0Div        = Apb0Div2;
  stcClkConfig.enAPB1Div        = Apb1Div2;
  stcClkConfig.enAPB2Div        = Apb2Div2;
  stcClkConfig.bAPB1Disable     = FALSE;
  stcClkConfig.bAPB2Disable     = FALSE;
  stcClkConfig.enSCOWaitTime    = ScoWaitExp15;
  stcClkConfig.enMCOWaitTime    = McoWaitExp117;
  stcClkConfig.enPLLOWaitTime   = PlloWaitExp19;
  stcClkConfig.u8PllK           = 1;
  stcClkConfig.u8PllM           = 1;
  stcClkConfig.u8PllN           = 20;
  stcClkConfig.bPllIrq          = FALSE;
  stcClkConfig.bScoIrq          = FALSE;
  stcClkConfig.bMcoIrq          = FALSE;
  stcClkConfig.pfnHook          = &ClkHookCallback;

  // CLK functions ...
       . . .
}
```

5.5.24.2 Setting the Clock

Assume a switched off Main oscillator and PLL. The system runs at HC-RC oscillator. Then it can be switched to PLL clock with the configuration above like:

```
#include "clk.h"

function
{
  stc_clk_config_t  stcClkConfig;

  L3_ZERO_STRUCT(stcClkConfig);

      . . .

  if (Ok == Clk_SetDividers(&stcClkConfig))         // PCLK0/1/2 => HCLK / 2
  {
    if (Ok == Clk_SetStabilizationWaitTimes(&stcClkConfig)) // see above
    {
      if (Ok == Clk_SetSource(&stcClkConfig))       // PLL clock
      {
        if (Ok == Clk_SetMode(&stcClkConfig))       // Clock Run Mode
        {
          // If pc arrives here, function calls above work properly
      . . .
}
```

5.5.24.3 Transition to Sub Clock

Assume a transition form PLL clock to Sub Clock run mode.

```
#include "clk.h"

#define SUB_CLOCK_STABILIZATION_TIMEOUT 200000000 // Time out count for Sub
                                                  //   clock stabilization
#define CLK_SOURCE_TRANSITION_TIMEOUT   2         // Time out for clock source
                                                  //   transition (only 2
                                                  //   trials, if clock is
                                                  //   stabilized!)
function
{
  stc_clk_config_t  stcClkConfig;

  L3_ZERO_STRUCT(stcClkConfig);

      . . .

  Clk_EnableSubClock();

  if (Ok != Clk_WaitForSubOscillator(SUB_CLOCK_STABILIZATION_TIMEOUT))
  {
    // Error handling here ...
  }

  stcClkConfig.enSource         = ClkSub;
  stcClkConfig.bEnableSubClock  = TRUE;
  stcClkConfig.bEnableMainClock = FALSE;
  stcClkConfig.bEnablePll       = FALSE;

  if (Ok == Clk_SetSource(&stcClkConfig))
  {
    if (Ok != Clk_WaitForClockSourceReady( ClkSub,
                                           CLK_SOURCE_TRANSITION_TIMEOUT ))
    {
      // Error handling here ...
    }
      . . .
}
```

5.5.24.4 Transitions and wake-up to and from Low Power Modes

The following example shows how an external interrupt is used to wake-up an MCU from timer, sleep, or stop mode. The configuration and API of the external interrupts are explained in detail in chapter 5.13.

```
#include "clk.h"
#include "exint.h"

/******************************************************************************/
/* Eternal interrupt Ch#0 callback function                                   */
/******************************************************************************/
void EICallback(void)
{
  // Do something after wake-up
}

function
{
  stc_clk_config_t   stcClkConfig;
  stc_exint_config_t stcEIConfig;

  L3_ZERO_STRUCT(stcClkConfig);
  L3_ZERO_STRUCT(stcEIConfig);

       . . .

  // Setup External Interrupt Ch#0 for system wake-up
  stcEIConfig.abEnable[0]           = TRUE;
  stcEIConfig.aenLevel[0]           = ExIntFallingEdge;
  stcEIConfig.apfnExintCallback[0] = &EICallback;
  Exint_Init(&stcEIConfig);

       . . .

  // Transition to Sleep Mode
  stcClkConfig.enMode = ClkSleep;
  Clk_SetMode(&stcClkConfig);
  // Wakeup with external interrupt #0 here
                                        // Transition to Timer Mode
  stcClkConfig.enMode = ClkTimer;
  Clk_SetMode(&stcClkConfig);
  // Wakeup with external interrupt #0 here

  // Transition to Stop Mode
  stcClkConfig.enMode = ClkStop;
  Clk_SetMode(&stcClkConfig);
  // Wakeup with external interrupt #0 here

       . . .
}
```

5.5.24.5 Using Clk_GetParameters() for configuration set

Sometime it is useful, only to change some clock configuration parameters, but the exact clock configuration state may be unknown during application run time. In this case `Clk_GetParameters()` can be used.

The following code shows how to do this:

```
#include "clk.h"

function
{
  stc_clk_config_t  stcClkConfig;

  L3_ZERO_STRUCT(stcClkConfig);

       . . .

  if (Ok == Clk_GetParameters(&stcClkConfig))
  {
    // Now all recent clock parameters are stored in stcClkConfig.

    // The parameters can be changed. Assume a transition to HS-RC
    //   clock run mode.
    stcClkConfig.enSource         = ClkHsCr;
    stcClkConfig.bEnableMainClock = FALSE;
    Clk_SetSource(&stcClkConfig);

       . . .
  }
}
```

5.6 (CRC) Cyclic Redundancy Check

Type Definition	`stc_crcn_t`
Configuration Type	`stc_crc_config_t`
Address Operator	`CRC0`

Initialize the CRC with `Crc_Init()`. After this there are to ways to calculate the check result:

- Providing data by CPU via `Crc_Push8()`, `Crc_Push16()`, or `Crc_Push32()`
- Using DMA

The result can be read by calling `Crc_ReadResult()`. The user has to take care of the endianess, which is used.

5.6.1 Configuration Structure

The CRC module uses the following configuration structure of the type `stc_crc_config_t`:

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

5.6.2 Crc_Init()

This function initializes the CRC macro. It has the following format:

Prototype	

`en_result_t Crc_Init( stc_crcn_t* pstcCrc,`
`stc_crc_config_t* pstcConfig )`	

Parameter Name	**Description**
`[in] pstcCrc`	CRC instance pointer
`[in] pstcConfig`	CRC module configuration (see above)
Return Values	**Description**
`Ok`	Initialization of CRC module successfully done
`ErrorInvalidParameter`	• `pstcDt == NULL` • Parameter out of range

5.6.3 Crc_DeInit()

This function de-initializes the CRC macro. It has the following format:

Prototype	
`en_result_t Crc_DeInit( stc_crcn_t*   pstcCrc )`	
Parameter Name	**Description**
`[in] pstcCrc`	CRC instance pointer
Return Values	**Description**
`Ok`	De-Initialization of CRC module successfully done
`ErrorInvalidParameter`	`pstcDt == NULL`

5.6.4 Crc_Push8()

Push 8-bit integer data to a CRC module with if no DMA is used.:

Prototype	

`en_result_t Crc_Push8( stc_crcn_t* pstcCrc,`
`                        uint8_t     u8DataToPush )`	

Parameter Name	**Description**
`[in] pstcCrc`	CRC instance pointer
`[in] u8DataToPush`	8-Bit data to be pushed to CRC
Return Values	**Description**
`Ok`	De-Initialization of CRC module successfully done
`ErrorInvalidParameter`	`pstcDt == NULL`

5.6.5 Crc_Push16()

Push 16-bit integer data to a CRC module with if no DMA is used.:

Prototype	

`en_result_t Crc_Push16( stc_crcn_t* pstcCrc,`
`                         uint16_t    u16DataToPush )`	

Parameter Name	**Description**
`[in] pstcCrc`	CRC instance pointer
`[in] u16DataToPush`	16-Bit data to be pushed to CRC
Return Values	**Description**
`Ok`	De-Initialization of CRC module successfully done
`ErrorInvalidParameter`	`pstcDt == NULL`

5.6.6 Crc_Push32()

Push 16-bit integer data to a CRC module with if no DMA is used.:

Prototype	
`en_result_t Crc_Push32( stc_crcn_t* pstcCrc, uint32_t u32DataToPush )`	
Parameter Name	**Description**
`[in] pstcCrc`	CRC instance pointer
`[in] u32DataToPush`	32-Bit data to be pushed to CRC
Return Values	**Description**
`Ok`	De-Initialization of CRC module successfully done
`ErrorInvalidParameter`	`pstcDt == NULL`

5.6.7 Crc_ReadResult()

This function returns a 32-bit value regardless of a valid pointer to the CRC instance anyhow.

Prototype	
`uint32_t Crc_ReadResult( stc_crcn_t* pstcCrc )`	
Parameter Name	**Description**
`[in] pstcCrc`	CRC instance pointer
Return Values	**Description**
`uint32_t`	CRC result value

5.6.8 Example Code

The following code excerpt examples shows how to handle the CRC functions.

5.6.8.1 Configuration

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

```
#include "crc.h"

function
{
  stcCrcConfig.enMode                 = Crc32;      // CRC32 used
  stcCrcConfig.bUseDma                = …;          // see below
  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
}
```

The following code excerpt shows example data, which is used for CRC here.

```
const uint32_t u32ConstData[64] = {
    0x6393B370, 0xF2BB4FC0, 0x6D793D2C, 0x508B2092,
       . . .
    0xD663962A, 0x67CB9FA2, 0xCD318688, 0x393DAA84
  };
```

5.6.8.2 CRC without DMA

The CRC can be fed without DMA usage as shown in the next code excerpt. Except for `bUseDma` the configuration 5.6.8.1 is used.

```
#include "crc.h"

function
{
  uint8_t          u8Count;
  stc_crc_config_t stcCrcConfig;

  L3_ZERO_STRUCT(stcCrcConfig);

       . . .

  stcCrcConfig.bUseDma = FALSE;      // No DMA used

       . . .

  // Test 32-bit pushing
  if (Ok == Crc_Init((stc_crcn_t*)&CRC0, &stcCrcConfig))
  {
    for (u8Count = 0; u8Count < 64; u8Count++)
    {
      Crc_Push32((stc_crcn_t*)&CRC0, cu32ConstData[u8Count]);
    }

    Crc_DeInit((stc_crcn_t*)&CRC0); // Return value ignored, because of if case
                                    //   above and same instance pointer used.
  }
}
```

5.6.8.3 CRC with DMA

For usage of the DMA please refer to chapter 5.10. The following code uses CRC and DMA together. Except for `bUseDma` the configuration 5.6.8.1 is used.

```
#include "crc.h"
#include "dma.h"

uint32_t u32CrcResult ;
uint8_t  u8DmaFinished = 0;

/******************************************************************************/
/* DMA callback functions                                                     */
/******************************************************************************/
void Crc_DmaError(uint8_t u8DmaErrorCode)
{
  … // Some error handling here ...
}

void Crc_DmaFinished(void)
{
  u8DmaFinished = 1;
}

function
{
  stc_crc_config_t stcCrcConfig;
  stc_dma_config_t stcCrcDmaConfig;

  L3_ZERO_STRUCT(stcCrcConfig);
  L3_ZERO_STRUCT(stcCrcDmaConfig);

       . . .

  stcCrcConfig.bUseDma = TRUE;      // DMA used

       . . .

  if (Ok == Crc_Init((stc_crcn_t*)&CRC0, &stcCrcConfig))
  {
    // Setup DMA
    stcCrcDmaConfig.bEnable                 = 0;        // Just init DMA first
    stcCrcDmaConfig.bPause                  = 0;
    stcCrcDmaConfig.bSoftwareTrigger        = FALSE;    // Just init DMA first
    stcCrcDmaConfig.enDmaIdrq               = Software; // Software DMA trig.
    stcCrcDmaConfig.u8DmaChannel            = 0;
    stcCrcDmaConfig.u8BlockCount            = 0;
    stcCrcDmaConfig.u16TransferCount        = 63;       // u32ConstData size - 1
    stcCrcDmaConfig.enTransferMode          = DmaBurstTransfer;
    stcCrcDmaConfig.enTransferWidth         = Dma32Bit;
    stcCrcDmaConfig.u32SourceAddress        = (uint32_t) &cu32ConstData;
    stcCrcDmaConfig.u32DestinationAddress   = (uint32_t) &(FM3_CRC->CRCIN);
    stcCrcDmaConfig.bFixedSource            = FALSE;
    stcCrcDmaConfig.bFixedDestination       = TRUE;
    stcCrcDmaConfig.bReloadCount            = FALSE;
    stcCrcDmaConfig.bReloadSource           = FALSE;
    stcCrcDmaConfig.bReloadDestination      = FALSE;
    stcCrcDmaConfig.bErrorInterruptEnable   = TRUE;
    stcCrcDmaConfig.bCompletionInterruptEnable = TRUE;
    stcCrcDmaConfig.bEnableBitMask          = FALSE;
    stcCrcDmaConfig.pfnCallback             = &Crc_DmaFinished;
    stcCrcDmaConfig.pfnErrorCallback        = &Crc_DmaError;

    Dma_Init_Channel(&stcCrcDmaConfig);
    Dma_Enable();

    stcCrcDmaConfig.bEnable           = 1;     // Now enable DMA
    stcCrcDmaConfig.bPause            = 0;
    stcCrcDmaConfig.bSoftwareTrigger = TRUE; // Prepare software trigger
```

▼

▲

```
    Dma_Set_Channel(&stcCrcDmaConfig);        // Trigger DMA

    while (0 == u8DmaFinished);               // Wait for DMA finished

    u32CrcResult = Crc_ReadResult((stc_crcn_t*)&CRC0) ;

    // Do something with result value
  }
```

5.7 (CRTRIM) CR Trimming

Type Definition	–
Configuration Type	–
Address Operator	–

The trimming function `Crtrim_SetTrimmingFromFlash()` copies the Flash HS-CR trimming data to the trimming register of the HS-CR Clock. If all bits of this data are equal to 1 (erased CR Trimming Sector) `result_t::ErrorInvalidParameter` returned and the trimming register remains unchanged.

`Crtrim_SetTrimming()` sets the function's argument to the HS-CR trimming register.

Note:

- If `L3_PERIPHERAL_ENABLE_CRTRIM_FLASH` is set `to L3_ON` in *l3_user.h* also the function `Crtrim_UpdateFlash()` is compiled. `L3_NO_FLASH_RAMCODE` must be set to `L3_OFF` and `L3_PERIPHERAL_ENABLE_FLASH` to `L3_ON` in this case!
- The CR-Trimming process like `Crtrim_SetTrimmingFromFlash()` is also done in `System_init()` in the *system_mb9[ab]xyz.c* file. It can be switched off by defining `CR_TRIM_SETUP` to `0` in *system_mb9[ab]xyz.h*.

5.7.1 Crtrim_SetTrimmingFromFlash()

This function copies the data of the Flash CR trimming contents to the HS-CR clock trimming register.

Prototype	
`en_result_t Crtrim_SetTrimmingFromFlash(void)`	
Return Values	**Description**
`Ok`	HS-CR trimming has been setup
`ErrorInvalidParameter`	All trimming data bits are '`1`': Erased sector assumed

5.7.2 Crtrim_SetTrimming()

This function copies the function's argument to the HS-CR clock trimming register. The maximum value (`CR_TRIMM_MAX_VALUE`) is checked Device Type-dependent.

Prototype	
`en_result_t Crtrim_SetTrimming(uint16_t u16TrimmData)`	
Parameter Name	**Description**
`[in] uint16_t`	Trimming data
Return Values	**Description**
`Ok`	HS-CR trimming has been setup
`ErrorInvalidParameter`	Trimming data exceeds `CR_TRIMM_MAX_VALUE`

5.7.3 Crtrim_UpdateFlash()

This function erases the CR trimming data sector and writes new data. RAMCODE and FLASH code (also, if Work Flash is supported) *must* be set active!

Prototype	
`en_result_t Crtrim_UpdateFlash(uint16_t u16TrimmData)`	
Parameter Name	**Description**
`[in] uint16_t`	Trimming data
Return Values	**Description**
`Ok`	New HS-CR trimming data has been written to Flash
`ErrorInvalidParameter`	Trimming data exceeds `CR_TRIMM_MAX_VALUE` or Flash corrupted (still `0`-Bits after erase).
`ErrorTimeout`	Flash erase or write timed-out
`ErrorOperationInProgress`	Flash erase or write access not available yet

5.7.4 Crtrim_SetHsRcClkDiv()

This function sets device type-dependent the HS-CR clock divider with an argument of type of `en_clk_hs_rc_div_t` as described below.

Prototype	
`en_result_t Crtrim_SetHsRcClkDiv(en_clk_hs_rc_div_t enHsRcClkDiv)`	
Parameter Name	**Description**
`[in] en_clk_hs_rc_div_t`	Clock divider, see table below.
Return Values	**Description**
`Ok`	Clock divider has be set.
`ErrorInvalidParameter`	Wrong or unknown enumerator used

The type `en_clk_hs_rc_div_t` has the following enumerators:

Device Type other than 3	**Device Type 3**	**Description**
`HsClkDiv4`	`HsClkDiv4`	HS-CR Clock is divided by 4
`HsClkDiv8`	`HsClkDiv8`	HS-CR Clock is divided by 8
`HsClkDiv16`	`HsClkDiv16`	HS-CR Clock is divided by 16
`HsClkDiv32`	`HsClkDiv32`	HS-CR Clock is divided by 32
-	`HsClkDiv64`	HS-CR Clock is divided by 64
-	`HsClkDiv128`	HS-CR Clock is divided by 128
-	`HsClkDiv256`	HS-CR Clock is divided by 256
-	`HsClkDiv512`	HS-CR Clock is divided by 512

5.7.5 Example Code

The following example shows how to use the CRTRIM driver module's functions.

```
#include "crtrim.h"

function
{
  uint16_t u16TrimmData;

  Crtrim_SetTrimmingFromFlash();

  u16TrimmData = FM3_FLASH_IF->CRTRMM;

  Crtrim_SetTrimming(u16TrimmData);

  Crtrim_UpdateFlash(u16TrimmData); // Needs L3_PERIPHERAL_ENABLE_FLASH == L3_ON
}
```

5.8 (CSV) Clock Super Visor

Type Definition	–
Configuration Type	`stc_csv_config_t`
Address Operator	–

The function `Csv_Init()` initializes the Main- and Sub Clock Supervisor (CSV) if enabled and the (Anomalous) Frequency Count (Detection) Supervisor (FCS) if enabled.

The CSV is only able to perform a reset on failure detection. Therefore after reset (in the initialization part of the application) the function `Csv_ReadStatus()` should be used to determine a possible CSV failure reset cause.

The FCS first generates an interrupt on anomalous frequency detection. In this case a callback function is called with an argument of the content of the Frequency Detection Counter Register (FCSWD_CTL). If `stc_csv_config::bFcsIrqAutoClear` is `TRUE` the interrupt request is cleared within the ISR. If this configuration is not set, the user may set `stc_csv_config::bFcsIrqEnable` to `FALSE` in the callback function, call Csv_EnableDisable() to disable the interrupt and thus let the FCS force a reset on next FCS failure.

If `Csv_EnableDisableParam()` is used instead of `Csv_EnableDisable()` the configuration `stc_csv_config_t` can be purged on runtime after initialization.

Notes:

- The clock(s) to be supervised must be enabled and stable!
- For anomalous frequency detection the High Speed CR must be trimmed!

5.8.1 Configuration Structure

The configuration structure of type of `stc_csv_config_t` consists of:

Type	Field	Possible Values	Description
`oolean_t`	`bMainCsvEnable`	`TRUE` `FALSE`	Main Clock Supervisor enabled Main Clock Supervisor disabled
`oolean_t`	`bSubCsvEnable`	`TRUE` `FALSE`	Sub Clock Supervisor enabled Sub Clock Supervisor disabled
`oolean_t`	`bFcsEnable`	`TRUE` `FALSE`	Frequency Count Supervisor enabled Frequency Count Supervisor disabled
`oolean_t`	`bFcsResetEnable`	`TRUE` `FALSE`	FCS Reset enabled FCS Reset disabled
`oolean_t`	`bFcsIrqEnable`	`TRUE` `FALSE`	FCS Interrupt enabled FCS Interrupt disabled
`oolean_t`	`bFcsIrqAutoClear`	`TRUE` `FALSE`	FCS IRQ is cleared in driver ISR FCS IRQ is not cleared in driver ISR
`en_csv_fcs_…` `count_cycle_t`	`enFcsCountCycle`	`CsvFcsFreq…` `…Div256`	Count cycle setting: 1/256 frequency of HS-CR clock

		…Div512 …Div1024	1/512 frequency of HS-CR clock 1/1024 frequency of HS-CR clock
uint16_t	u16Freqency… WindowLower	–	Frequency detection window lower setting
uint16_t	u16Freqency… WindowUpper	–	Frequency detection window upper setting
csv_func_ptr_t	pfnCallback… (uint16_t)*	–	FCS interrupt callback function with uint16_t argument*

*The argument for the CSV Callback function is the value of the FCSWB_CTL register.

5.8.2 Csv_Init()

This function initializes and enables the Clock Supervisor according the configuration.

Prototype	
en_result_t Csv_Init(stc_csv_config_t* pstcConfig)	
Parameter Name	**Description**
[in] pstcConfig	Pointer to CSV configuration structure
Return Values	**Description**
Ok	CSV has been initialized successfully
ErrorInvalidParameter	• pstcConfig == NULL • Invalid FCS count cycle setting

5.8.3 Csv_DeInit()

This function de-initializes and disabled the Clock Supervisor completely.

Prototype	
en_result_t Csv_DeInit(void)	
Return Values	**Description**
Ok	CSV has been de-initialized successfully

5.8.4 Csv_EnableDisable()

This function en- or disables the FCS, Main CSV, and Sub CSV according to the configuration. Also the FCS interrupt and its auto-clear option can be en- or disabled with this function. Csv_Init() must be called before!

Prototype	
en_result_t Csv_EnableDisable(stc_csv_config_t* pstcConfig)	
Parameter Name	**Description**
[in] pstcConfig	Pointer to CSV configuration structure
Return Values	**Description**
Ok	CSV settings have been updated successfully
ErrorInvalidParameter	pstcConfig == NULL

5.8.5 Csv_EnableDisableParam()

This function en- or disables the FCS, Main CSV, and Sub CSV according to the arguments. Also the FCS interrupt and its auto-clear option can be en- or disabled with this function. This function can be used instead of `Csv_EnableDisable()`, if the configuration structure is not saved on runtime. `Csv_Init()` must be called before!

Prototype	
`en_result_t Csv_EnableDisableParam( oolean_t bMainCsvEnable, oolean_t bSubCsvEnable, oolean_t bFcsEnable, oolean_t bFcsIrqEnable, oolean_t bFcsIrqAutoClear )`	
Parameter Name	**Description**
`[in] bMainCsvEnable`	`TRUE`: Enable Main CSV `FALSE`: Disable Main CSV
`[in] bSubCsvEnable`	`TRUE`: Enable Sub CSV `FALSE`: Disable Sub CSV
`[in] bFcsEnable`	`TRUE`: Enable FCS `FALSE`: Disable FCS
`[in] bFcsIrqEnable`	`TRUE`: Enable FCS Interrupt `FALSE`: Disable FCS Interrupt
`[in] bFcsIrqAutoClear`	`TRUE`: Enable FCS Interrupt auto-clear in ISR `FALSE`: Disable FCS Interrupt auto-clear in ISR
Return Values	**Description**
`Ok`	CSV settings have been updated successfully

5.8.6 Csv_ReadStatus()

This function return the status of the Main- and Subclock Supervisor. Because the `CSV_STR` register is cleared during read, the return value is 'coded' in `en_csv_status_t`, which is described below.

Prototype	
`en_csv_status_t Csv_ReadStatus(void)`	
Return Values	**Description**
`CsvNoDetection`	No failure has been detected.
`CsvMainFailure`	A Main clock failure has been detected
`CsvSubFailure`	A Sub clock failure has been detected
`CsvMainSubFailure`	Main and Sub clock failures have been detected

5.8.7 CvsCallback()

This function is called, if `stc_csv_config_t::pfnCallback` is defined and the according interrupt is enabled. `U16FcswbCtl` is the value of the `FCSWB_CTL` register.

Callback Function
`void *CvsCallback*( uint16_t u16FcswbCtl )`

5.8.8 Example Code

The following code excerpt shows how to set up the CSV.

```
#include "csv.h"

// Adjust these window values for own system
#define CSV_FCS_LOWER 1100
#define CSV_FCS_UPPER 1105

uint16_t u16FreqErrorVar;

void CsvCallback(uint16_t u16FreqError)
{
  u16FreqErrorVar = u16FreqError;

  // Error handle ...
}

function
{
  stc_csv_config_t stcCsvConfig;

  L3_ZERO_STRUCT(stcCsvConfig);

  stcCsvConfig.bMainCsvEnable          = TRUE;
  stcCsvConfig.bSubCsvEnable           = FALSE;
  stcCsvConfig.bFcsEnable              = TRUE;
  stcCsvConfig.bFcsResetEnable         = FALSE;
  stcCsvConfig.bFcsIrqEnable           = TRUE;
  stcCsvConfig.bFcsIrqAutoClear        = TRUE;
  stcCsvConfig.enFcsCountCycle         = CsvFcsFreqDiv512;
  stcCsvConfig.u16FreqencyWindowLower = CSV_FCS_LOWER;
  stcCsvConfig.u16FreqencyWindowUpper = CSV_FCS_UPPER;
  stcCsvConfig.pfnCallback             = &CsvCallback;

  Csv_Init(&stcCsvConfig);

  . . .
}
```

5.9 (DAC) Digital Analog Converter

Type Definition	`stc_dacn_t`
Configuration Type	–
Address Operator	`DAC`*n*

Because the DAC does not need any configuration, no configuration structure and no Init or De-Init functions are needed. The output can be easily enabled and disabled by `Dac_Enable0()`, `Dac_Enable1()`, `Dac_Disable0()`, and `Dac_Disable1()`. The DAC value can be set by `Dac_SetValue0()` and `Dac_SetValue1()`.

The indexes `0` and `1` correspond to the channel 0 and 1 of a DAC instance.

5.9.1 Dac_SetValue0(), Dac_SetValue1()

These functions set the 10-Bit DAC value to channel 0 or 1.

Prototype

```
en_result_t Dac_SetValuen( stc_dacn_t* pstcDac,
                           uint16_t    u16DacValue )
```

n = 0, 1

Parameter Name	**Description**
`[in] pstcDac`	DAC instance pointer
`[in] u16DacValue`	10-Bit DAC value
Return Values	**Description**
`Ok`	DAC value set

5.9.2 Dac_Enable0(), Dac_Enable1()

These functions enable a DAC channel. Note that no other peripheral shall block the pin.

Prototype

```
en_result_t Dac_Enablen( stc_dacn_t* pstcDac )
```

n = 0, 1

Parameter Name	**Description**
`[in] pstcDac`	DAC instance pointer
Return Values	**Description**
`Ok`	DAC channel enabled

5.9.3 Dac_Disable0(), Dac_Disable1()

These functions disable a DAC channel.

Prototype	
`en_result_t Dac_Disablen( stc_dacn_t* pstcDac )` *n* = 0, 1	
Parameter Name	**Description**
`[in] pstcDac`	DAC instance pointer
Return Values	**Description**
`Ok`	DAC channel disabled

5.9.4 Example Code

The following code excerpt shows how to use the DAC by output of two sine wave tones.

```
#include "dac.h"
#include <math.h>

#define DAC_MAX_DATA 1024   // Number of 'samples'

uint16_t au16Sinus0[DAC_MAX_DATA];
uint16_t au16Sinus1[DAC_MAX_DATA];

function
{
  float64_t f64Sinus;
  float64_t f64Phi;
  float64_t f64Pi = 3.141592636;

  // Frequency 0 sample step
  float64_t f64Fract0 = (f64Pi * (float64_t)2) / ((float64_t) DAC_MAX_DATA);

  // Frequency 1 sample step
  float64_t f64Fract1 = (f64Pi * (float64_t)4) / ((float64_t) DAC_MAX_DATA);

  uint16_t u16Index;
  uint32_t u32Delay;

  // Create sine wave tones
  for (u16Index = 0; u16Index < DAC_MAX_DATA; u16Index++)
  {

    f64Phi = (float64_t)u16Index * f64Fract0;
    f64Sinus = sin(f64Phi);
    au16Sinus0[u16Index] = (uint16_t)((float64_t)510 * f64Sinus +
                           (float64_t)512);

    f64Phi = (float64_t)u16Index * f64Fract1;
    f64Sinus = sin(f64Phi);
    au16Sinus1[u16Index] = (uint16_t)((float64_t)510 * f64Sinus +
                           (float64_t)512);
  }

  Dac_Enable0((stc_dacn_t*)DAC0);
  Dac_Enable1((stc_dacn_t*)DAC0);

  // Play sine wave tones
  while(1)
  {
    for (u16Index = 0; u16Index < DAC_MAX_DATA; u16Index++)
    {
      Dac_SetValue0((stc_dacn_t*)DAC0, au16Sinus0[u16Index]);
      Dac_SetValue1((stc_dacn_t*)DAC0, au16Sinus1[u16Index]);

      u32Delay = 10;
      while(--u32Delay);
    }
  }
}
```

5.10 (DMA) Direct Memory Access

Type Definition	–
Configuration Type	`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”!

5.10.1 Configuration

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

Type	Field	Possible Values	Description
`oolean_t`	`bEnable`	`TRUE` `FALSE`	Enable DMA channel Disable DMA channel
`oolean_t`	`bPause`	`TRUE` `FALSE`	Pause of a DMA channel Normal function / resume
`uint8_t`	`bSoftware…` `Trigger`	–	Trigger bit for software transfer (only set by `Dma_Enable()`)
`uint8_t`	`u8DmaChannel`	–	DMA channel number to be configured
`en_dma_idreq_t`	`enDmaIdrq`	*see below*	*see below*
`uint8_t`	`u8BlockCount`	–	Block counter
`uint16_t`	`u16Transfer…` `Count`	–	Transfer count (Number of transfers – 1)

`en_dma_…`
`transfermode_t`
	`enTransfer…`
	`Mode`
		`DmaBlockTransfer`
		`DmaBurstTransfer`
		`DmaDemandTransfer`
			Block transfer
			Burst transfer
			Demand transfer

`en_dma_…`
`transferwidth_t`
	`enTransfer…`
	`Width`
		`Dma8Bit`
		`Dma16Bit`
		`Dma32Bit`
			8 Bit transfer
			16 Bit transfer
			32 Bit transfer

`uint32_t`	`u32Source…` `Address`	–	DMA source address

uint32_t	u32… Destination… Address	-	DMA destination address
oolean_t	bFixedSource	TRUE FALSE	Source address fixed Source address increase
oolean_t	bFixed… Destination	TRUE FALSE	Destination address fixed Destin. Address increase
oolean_t	bReloadCount	TRUE FALSE	Reload Count at end Stop DMA at transfer end
oolean_t	bReload… Source	TRUE FALSE	Reload source address Do not reload source
oolean_t	bReload… Destination	TRUE FALSE	Reload destination address Do not reload destination
oolean_t	bError… Interrupt… Enable	TRUE FALSE	Interrupt at error of DMA No interrupt
oolean_t	bCompletion… Interrupt… Enable	TRUE FALSE	Interrupt at end of DMA No interrupt
oolean_t	bEnableBit… Mask	TRUE FALSE	Clear EB at end of DMA Do not clear EB at end
func_ptr_t	pfnCallback	-	Completion callback function pointer
func_ptr_arg1_t	pfnError… Callback	-	Error callback function pointer
uint8_t	u8ErrorStop… Status	-	Error code from stop status

5.10.2 Dma_Init_Channel()

Sets up an DMA channel without starting immediate DMA transfer. Dma_Enable () is used for starting DMA transfer.

Prototype	
en_result_t Dma_Init_Channel(volatile stc_dma_config_t* pstcConfig)	
Parameter Name	**Description**
[in] pstcConfig	Pointer to DMA configuration structure
Return Values	**Description**
Ok	Init successfully done
ErrorInvalidParameter	• pstcAdc == NULL • Invalid configuration
OperationInProgress	DMA channel already in use

5.10.3 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 structure
Return Values	**Description**
`Ok`	Setting successfully done

5.10.4 Dma_Enable()

Enables DMA globally.

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

5.10.5 Dma_Disable()

Disables DMA globally.

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

5.10.6 Dma_DeInit_Channel()

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 structure
Return Values	**Description**
`Ok`	DMA channel successfully cleared
`ErrorInvalidParameter`	• `pstcAdc == NULL` • Invalid configuration

5.10.7 Dma_Set_ChannelParam()

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_ChannelParam( uint8_t   u8DmaChannel,`
`                                    oolean_t bEnable,`
`                                    oolean_t bPause,`
`                                    oolean_t bSoftwareTrigger )`	

Parameter Name	**Description**
`[in] pstcConfig`	Pointer to DMA configuration structure
`[in] bEnable`	`TRUE` : Enables channel `FALSE` : Disables channel
`[in] bPause`	`TRUE` : Pauses a channel `FALSE` : Starts/resumes a channel
`[in] bSoftwareTrigger`	`TRUE`: Triggers a channel `FALSE`: Does not trigger a channel
Return Values	**Description**
`Ok`	Setting finished successfully

5.10.8 Dma_DeInit_ChannelParam()

This function clears a DMA transfer by channel parameter.

Prototype	
`en_result_t Dma_DeInit_ChannelParam( uint8_t u8DmaChannel )`	
Parameter Name	**Description**
`[in] u8DmaChannel`	DMA channel to be cleared
Return Values	**Description**
`Ok`	DMA channel cleared successfully

5.10.9 DmaCallback()

This function is called, if `stc_dma_config_t::pfnCallback` is defined and the according interrupt is enabled.

Callback Function
`void` *`DmaCallback`*`( void )`

5.10.10 DmaErrorCallback()

This function is called, if `stc_dma_config_t::pfnErrorCallback` is defined and the according interrupt is enabled. The argument `u8StopStatus` contains the stop status error.

Callback Function
`void` *`DmaErrorCallback`*`( uint8_t u8StopStatus )`

5.10.11 Example Code

Please refer to ADC example in chapter 5.2.17.3 and CRC example in chapter 5.6.8.3.

5.11 (DSM) Deep Standby Modes

Type Definition	–
Configuration Type	`stc_dsm_config_t`
Address Operator	–

`Dsm_SetMode()` sets the deep standby mode according to the configuration defined in a structure type of `stc_dsm_config_t`. The return cause can be retrieved by calling Dsm_GetReturnCause(). The return cause is stored in a pointered structure.

For 8-, 16-, and 32-Bit access to the Backup registers, special pointer definitions are defined.

Notes:

- The DSM driver does not handle wake-up pins, which may shared with analog input pins. The user has to take care of correct port settings (`ADE` to '`0`', if wake-up input functionality shall be used).
- Always call `Dsm_GetReturnCause()` very early in your application to clear possible return cause bits. Entering deep standby modes will not work correctly, if cause bits are remaining.

Caution:

Consider that `WKUP0` pin is always active (low level) for DS modes, so that it has to be pulled up by internal or external resistor! Refer to the pin assignment of your device for setting corresponding `PCR` pull-up register.

5.11.1 Configuration Structure

The DSM module uses the following channel configuration structure of the type `stc_dsm_config_t`:

Type	Field	Possible Values	Description
`en_dsm_mode_t`	`enMode`	`DeepStop` `DeepRtc`	Deep Standby Stop mode Deep Standby RTC mode
`oolean_t`	`bLpmPortHiZState`	`TRUE` `FALSE`	Hi-Z on all port during DS/RTC Preserve pin state
`oolean_t`	`bSubClkHdmiCec`	`TRUE` `FALSE`	HDMI-CEC clocked by Sub Clock HDMI-CEC not clocked by Sub Clk
`oolean_t`	`bSubClkRtc`	`TRUE` `FALSE`	RTC clocked by Sub Clock RTC not clocked by Sub Clock
`oolean_t`	`bInitxResetReturn`	`TRUE` `FALSE`	Return from DSM by `INITX` No return from DSM by `INITX`
`oolean_t`	`bLvdResetReturn`	`TRUE` `FALSE`	Return from DSM by LVD Reset No return by LVD Reset
`oolean_t`	`bLvdInterrupt`	`TRUE` `FALSE`	Return from DSM by LVD IRQ No return by LVD IRQ
`oolean_t`	`bRtcInterrupt`	`TRUE` `FALSE`	Return from DSM by RTC IRQ No return by RTC IRQ
`uint8_t`	`HdmiChannelEnable`	see below	Defines return channel of HDMI-

`stc_hdmi_channel_t`	`stcHdmiChannelEnable`		CEC/remote control
`uint8_t` `stc_wkpin_channel_t`	`WkPinChannelEnable` `stcWkPinChannelEnable`	see below	Defines Wake-up pin active
`uint8_t` `stc_wkpin_channel_t`	`WkPinChannelLevel` `stcWkPinChannelLevel`	see below	Defines Wake-up pin level
`oolean_t`	`bSramRetention…` `Enable`	`TRUE` `FALSE`	Retain SRAM contents in DSM No SRAM data retention
`func_ptr_t`	`pfnHook`	-	Hook call for low power modes. Called between `SLEEPDEEP = 1` and `WFI` instructions.

5.11.1.1 Structure of stc_hdmi_channel_t

The 8-Bit bitfield structure of `stc_hdmi_channel_t` is as follows:

Bitfield Name	Size
`Channel0`	1
`Channel1`	1
`reserved`	6

5.11.1.2 Structure of stc_wkpin_channel_t

The 8-Bit bitfield structure of `stc_wkpin_channel_t` is as follows:

Bitfield Name	Size
`Channel0`	1
`Channel1`	1
`Channel2`	1
`Channel3`	1
`Channel4`	1
`Channel5`	1
`reserved`	2

5.11.2 Return cause structure stc_dsm_return_cause_t

Because it may happen that several return causes occur a 16-Bit bitfield structure for these causes is provided:

Bitfield Name	Size	Description
`InitxReset`	1	Return cause: INITX (Reset)
`LvdReset`	1	Return cause: LVD Reset
`LvdInterrupt`	1	Return cause: LVD Interrupt
`RtcInterrupt`	1	Return cause: RTC Interrupt
`Hdmi0Interrupt`	1	Return cause: HDMI0 Interrupt
`Hdmi1Interrupt`	1	Return cause: HDMI1 Interrupt
`WakUpPin0`	1	Return cause: Wake-up Pin0 Interrupt
`WakUpPin1`	1	Return cause: Wake-up Pin1 Interrupt
`WakUpPin2`	1	Return cause: Wake-up Pin2 Interrupt
`WakUpPin3`	1	Return cause: Wake-up Pin3 Interrupt
`WakUpPin4`	1	Return cause: Wake-up Pin4 Interrupt
`WakUpPin5`	1	Return cause: Wake-up Pin5 Interrupt
`reserved`	4	-

5.11.3 Definition for Back-up Registers

For easy access, the following definitions were done for the Back-up Registers:

```
// Backup Register pointer types for 8-, 16-, and 32-bit access
// 8-Bit
#define u8DSM_BUR01 (*(volatile uint8_t*) 0x40035900UL)
#define u8DSM_BUR02 (*(volatile uint8_t*) 0x40035901UL)
#define u8DSM_BUR03 (*(volatile uint8_t*) 0x40035902UL)
#define u8DSM_BUR04 (*(volatile uint8_t*) 0x40035903UL)
#define u8DSM_BUR05 (*(volatile uint8_t*) 0x40035904UL)
#define u8DSM_BUR06 (*(volatile uint8_t*) 0x40035905UL)
#define u8DSM_BUR07 (*(volatile uint8_t*) 0x40035906UL)
#define u8DSM_BUR08 (*(volatile uint8_t*) 0x40035907UL)
#define u8DSM_BUR09 (*(volatile uint8_t*) 0x40035908UL)
#define u8DSM_BUR10 (*(volatile uint8_t*) 0x40035909UL)
#define u8DSM_BUR11 (*(volatile uint8_t*) 0x4003590AUL)
#define u8DSM_BUR12 (*(volatile uint8_t*) 0x4003590BUL)
#define u8DSM_BUR13 (*(volatile uint8_t*) 0x4003590CUL)
#define u8DSM_BUR14 (*(volatile uint8_t*) 0x4003590DUL)
#define u8DSM_BUR15 (*(volatile uint8_t*) 0x4003590EUL)
#define u8DSM_BUR16 (*(volatile uint8_t*) 0x4003590FUL)

// 16-Bit
#define u16DSM_BUR0102 (*(volatile uint16_t*) 0x40035900UL)
#define u16DSM_BUR0304 (*(volatile uint16_t*) 0x40035902UL)
#define u16DSM_BUR0506 (*(volatile uint16_t*) 0x40035904UL)
#define u16DSM_BUR0708 (*(volatile uint16_t*) 0x40035906UL)
#define u16DSM_BUR0910 (*(volatile uint16_t*) 0x40035908UL)
#define u16DSM_BUR1112 (*(volatile uint16_t*) 0x4003590AUL)
#define u16DSM_BUR1314 (*(volatile uint16_t*) 0x4003590CUL)
#define u16DSM_BUR1516 (*(volatile uint16_t*) 0x4003590EUL)

// 32-Bit
#define u32DSM_BUR01020304 (*(volatile uint32_t*) 0x40035900UL)
#define u32DSM_BUR05060708 (*(volatile uint32_t*) 0x40035904UL)
#define u32DSM_BUR09101112 (*(volatile uint32_t*) 0x40035908UL)
#define u32DSM_BUR13141516 (*(volatile uint32_t*) 0x4003590CUL)
```

5.11.4 Dsm_SetMode()

This function sets the Deep Standby STOP or RTC mode. The pin state defined by `stc_dsm_config_t::bLpmPortHiZState` is also set.

Prototype	
`en_result_t Dsm_SetMode(stc_dsm_config_t* pstcConfig)`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to DSM configuration structure
Return Values	**Description**
`Ok`	Successfully returned
`ErrorInvalidParameter`	• `pstcConfig == NULL` • Illegal Mode

Note, if the hook function pointer is given (unequal `NULL`), this user function is called between `SLEEPDEEP = 1` and `WFI` instructions. This allows to make some additional application dependent preparations to the deep standby mode.

5.11.5 Dsm_GetRetrunCause()

This function sets the return cause within a pointered structure of type `stc_dsm_return_t` explained in 5.11.2. If the return cause could not be determined `Error` is returned, otherwise `Ok`.

This function should be called early in the user's application to clear the return cause bits. Entering deep standby modes will not work correctly, if one or more of these bits are remaining.

Caution:

The user has to take care that before calling this function, all bits of the structure have to be '`0`'!

Prototype	
`en_result_t Dsm_GetReturnCause(stc_dsm_return_cause_t* pstcReturnCause)`	
Parameter Name	**Description**
`[out] pstcReturnCause`	DSM return cause structure
Return Values	**Description**
`Ok`	At least one return cause could be retrieved
`Error`	No return cause could be retrieved

5.11.6 HookFunction()

This function is called between `SLEEPDEEP = 1` and `WFI` , if `stc_dsm_config_t::pfnHook` is defined.

Hook Function
`void` *`HookFunction`*`( void )`

5.11.7 Example Code

The following code excerpt shows how to use the DSM functions. Assume, that `INITX`, LVD Reset, and Wake-up Pin #1 should be used for returning from deep stop mode. Also the SRAM data retention should be set.

```
#include "dsm.h"

function0
{
  stc_dsm_config_t stcDsmConfig;

  L3_ZERO_STRUCT(stcDsmConfig);

  . . .

  stcDsmConfig.enMode                             = DeepStop;
  stcDsmConfig.bLpmPortHiZState                   = FALSE;
  stcDsmConfig.bSubClkHdmiCec                     = FALSE;
  stcDsmConfig.bSubClkRtc                         = FALSE;
  stcDsmConfig.bInitxResetReturn                  = TRUE;
  stcDsmConfig.bLvdResetReturn                    = FALSE;
  stcDsmConfig.bLvdInterrupt                      = TRUE;
  stcDsmConfig.bRtcInterrupt                      = FALSE;
  stcDsmConfig.HdmiChannelEnable                  = 0;
  stcDsmConfig.stcWkPinChannelEnable.Channel0 = 0;
  stcDsmConfig.stcWkPinChannelEnable.Channel1 = 1;   // WKUP1
  stcDsmConfig.stcWkPinChannelEnable.Channel2 = 0;
  stcDsmConfig.stcWkPinChannelEnable.Channel3 = 0;
  stcDsmConfig.stcWkPinChannelEnable.Channel4 = 0;
  stcDsmConfig.stcWkPinChannelEnable.Channel5 = 0;
  stcDsmConfig.WkPinChannelLevel                  = 0;
  stcDsmConfig.bSramRetentionEnable               = TRUE;
  stcDsmConfig.pfnHook                            = NULL;

  // Write to Back-up Registerrs
  u8DSM_BUR01        = 0x12;
  u16DSM_BUR0304     = 0x1234;
  u32DSM_BUR09101112 = 0x12345678;

  FM3_CRG->SCM_CTL = 0;          // Select HS-CR clock (or use CLK driver)

  Dsm_SetMode(&stcDsmConfig);

  . . .

  if (0x12345678 != u32DSM_BUR09101112)
  {
    . . .
  }
  . . .
}

function1
{
  union
  {
    uint16_t          u16ReturnCause;
    stc_dsm_return_t  stcReturnCause;
  };

  . . .

  u16ReturnCause = 0;
  Dsm_SetMode(&u16ReturnCause);

  if (1 == stcReturnCause.InitxReset)
  {
    . . .
  }
  . . .
}
```

5.12 (DT) Dual Timer

Type Definition	`stc_dtn_t`
Configuration Type	`stc_dt_config_t`
Address Operator	`DT0`

`Dt_Init()` must be used for activation of the internal data structures for using the Dual Timer (DT). Here the interrupt callback functions for each of the 2 channels are set-up. Because the DT has no output pins, its interrupt service is always used.

For configuration of a DT channel a structure with the type `stc_dt_channel_config_t` has to be used. After configuration the DT channel can be set-up by `Dt_SetChannelConfig()`.

A DT Channel can be enabled by the function `Dt_EnableChannel()`. Depending on the used mode, it is started in:

- Free-run mode
- Periodic mode
- Single-shot mode

With `Dt_WriteLoadChannel()` the recent DT counter is set to the value given in the configuration `stc_dt_channel_config_t::u32DtLoad`. This works in each of the three operation modes.

With `Dt_WriteBgLoadChannel()` the background reload value can be set, which is then set to the load value after the DT counter reaches the next 0.

Before deinitialization of the DT by `Dt_DeInit()`, it is recommended to disable all channels via `Dt_DisableChannel()` before, to avoid a possible, unwanted interrupt.

Note, the Dual Timer shares its interrupt vector with the QPRC. Therefore use `bTouchNVIC` parameter in `Dt_Init() and Dt_DeInit()` whether to touch the QPRC-NVIC registers or not.

5.12.1 Configuration Structure

The DT module uses the following channel configuration structure of the type `stc_dt_channel_config_t`:

Type	Field	Possible Values	Description

`en_dt_mode_t`	`enMode`	`DtFreeRun`
		`DtPeriodic`
		`DtOneShot`
			Free-running Mode
			Periodic Mode
			One-shot Mode

`en_dt_prescaler_t`	`enPrescalerDiv`	`DtPrescalerDiv1`
		`DtRtescalerDiv16`
		`DtPrescalerDiv256`
			Prescaler Divisor 1
			Prescaler Divisor 16
			Prescaler Divisor 256

`en_dt_countersize_t`	`enCounterSize`	`DtConterSize16` `DtConterSize32`	16 Bit counter size 32 Bit counter size
`oolean_t`	`bEnable`	`TRUE` `FALSE`	DT Channel enable DT Channel disable
`uint32_t`	`u32DtLoad`	`0 ... 0xFFFFFFFF`	Ticks of Load Register
`uint32_t`	`u32DtBakgroundLoad`	`0 ... 0xFFFFFFFF`	Ticks of Background Load Register
`dt_cb_func_ptr_t`	`pfnCallback`	`-`	Function pointer to Callback

5.12.2 Dt_Init()

This function initializes a Dual Timer instance. It has the following format:

Prototype	

`en_result_t Dt_Init ( volatile stc_dtn_t* pstcDt,`
`                       boolean_t          bTouchNVIC )`	

Parameter Name	**Description**
`[in] pstcDt`	Dual Timer instance pointer
`[in] bTouchNVIC`	`TRUE:` NVIC is initialized `FALSE:` NVIC registers are not touched
Return Values	**Description**
`Ok`	Internal data has been setup
`ErrorInvalidParameter`	`pstcDt == NULL`

Note:

This function does not initialize the DT channels itself. `Dt_SetChannelConfig()` has to be called for each channel you wish to use after `Dt_Init()`.

5.12.3 Dt_DeInit()

This function de-initializes the DT module. Differently to the `Dt_init()` function, `Dt-DeInit()` also accesses the DT hardware register. They are set to `0` (default reset values). The configuration is not changed, but the callback pointers are reset to `NULL`.

Prototype	

`en_result_t Dt_DeInit ( volatile stc_dtn_t* pstcDt,`
`                         boolean_t          bTouchNVIC )`	

Parameter Name	**Description**
`[in] pstcDt`	Dual Timer instance pointer
`[in] bTouchNVIC`	`TRUE:` NVIC is reset `FALSE:` NVIC registers are not touched
Return Values	**Description**
`Ok`	Registers and internal data have been cleared
`ErrorInvalidParameter`	`pstcDt == NULL`

5.12.4 Dt_SetChannelConfig()

This function initializes the given DT channel. The parameters are checked for plausibility and the given callback is stored internally.

The timer enable `pstcConfig->bEnable` can be set to start the timer via this function. Alternatively the enable is set to `0` here and the timer can be enabled via `Dt_EnableChannel()`.

Prototype	
`en_result_t Dt_SetChannelConfig ( volatile stc_dtn_t* pstcDt, stc_dt_channel_config_t* pstcConfig, uint8_t u8Channel )`	
Parameter Name	**Description**
`[in] pstcDt`	Dual Timer instance pointer
`[in] pstcConfig`	Dt configuration parameters
`[in] u8Channel`	`0` or `1`
Return Values	**Description**
`Ok`	Channel configuration has been set
`ErrorInvalidParameter`	• `pstcDt == NULL` • `pstcConfig == NULL` • `u8Channel >= DT_CHANNEL_COUNT`

5.12.5 Dt_EnableChannel()

This function enables the given DT channel by `u8Channel`.

Prototype	
`en_result_t Dt_EnableChanel ( volatile stc_dtn_t* pstcDt, uint8_t u8Channel )`	
Parameter Name	**Description**
`[in] pstcDt`	Dual Timer instance pointer
`[in] u8Channel`	`0` or `1`
Return Values	**Description**
`Ok`	Channel enabled
`ErrorInvalidParameter`	• `pstcDt == NULL` • `u8Channel >= DT_CHANNEL_COUNT`

5.12.6 Dt_DisableChannel()

This function disables the given DT channel by `u8Channel`.

Prototype	
`en_result_t Dt_DisableChannel ( volatile stc_dtn_t* pstcDt, uint8_t u8Channel )`	
Parameter Name	**Description**
`[in] pstcDt`	Dual Timer instance pointer
`[in] u8Channel`	`0` or `1`
Return Values	**Description**
`Ok`	Channel disabled
`ErrorInvalidParameter`	• `pstcDt == NULL` • `u8Channel >= DT_CHANNEL_COUNT`

5.12.7 Dt_WriteLoadChannel()

This function writes to the load register of the given DT channel via configuration `pstcConfig` by `u8Channel`.

Prototype	
`en_result_t Dt_WriteLoadChannel ( volatile stc_dtn_t* pstcDt, stc_dt_channel_config_t* pstcConfig, uint8_t u8Channel )`	
Parameter Name	**Description**
`[in] pstcDt`	Dual Timer instance pointer
`[in] pstcConfig`	DT configuration parameters
`[in] u8Channel`	`0` or `1`
Return Values	**Description**
`Ok`	Channel disabled
`ErrorInvalidParameter`	• `pstcDt == NULL` • `u8Channel >= DT_CHANNEL_COUNT`

5.12.8 Dt_WriteBgLoadChannel()

This function writes to the background load register of the given DT channel via configuration `pstcConfig` by `u8Channel`.

Prototype	
`en_result_t Dt_WriteBgLoadChannel( volatile stc_dtn_t* pstcDt, stc_dt_channel_config_t* pstcConfig, uint8_t u8Channel )`	
Parameter Name	**Description**
`[in] pstcDt`	Dual Timer instance pointer
`[in] pstcConfig`	DT configuration parameters
`[in] u8Channel`	`0` or `1`
Return Values	**Description**
`Ok`	Channel disabled
`ErrorInvalidParameter`	• `pstcDt == NULL` • `u8Channel >= DT_CHANNEL_COUNT`

5.12.9 Dt_DirectWriteLoadChannel()

This function writes directly without configuration `pstcConfig` to the load register of the given DT channel by `u8Channel`.

Prototype	
`en_result_t Dt_DirectWriteLoadChannel ( volatile stc_dtn_t* pstcDt, uint32_t u32DtLoad, uint8_t u8Channel )`	
Parameter Name	**Description**
`[in] pstcDt`	Dual Timer instance pointer
`[in] u32DtLoad`	DT Load register value
`[in] u8Channel`	`0` or `1`
Return Values	**Description**
`Ok`	Channel disabled
`ErrorInvalidParameter`	• `pstcDt == NULL` • `u8Channel >= DT_CHANNEL_COUNT`

5.12.10 Dt_DirectWriteBgLoadChannel()

This function writes to the background load register of the given DT channel via configuration `pstcConfig` by `u8Channel`.

Prototype	
`en_result_t Dt_DirectWriteBgLoadChannel( volatile stc_dtn_t* pstcDt, uint32_t u32DtLoad, uint8_t u8Channel )`	
Parameter Name	**Description**
`[in] pstcDt`	Dual Timer instance pointer
`[in] u32DtLoad`	DT Background-Load register value
`[in] u8Channel`	`0` or `1`
Return Values	**Description**
`Ok`	Channel disabled
`ErrorInvalidParameter`	• `pstcDt == NULL` • `u8Channel >= DT_CHANNEL_COUNT`

5.12.11 Dt_ReadValueChannel()

This function returns the recent value of the DT count of Channel `u8Channel`. In 16-Bit mode, the upper 16 bits are set to zero automatically.

Prototype	
`uint32_t Dt_ReadValueChannel ( volatile FM3_DTIM_TypeDef * pstcDt, stc_dt_channel_config_t * pstcConfig, uint8_t u8Channel )`	
Parameter Name	**Description**
`[in] pstcDt`	Dual Timer instance pointer
`[in] pstcConfig`	DT configuration parameters
`[in] u8Channel`	`0` or `1`
Return Type	**Description**
`uint32_t`	Recent value of selected channel

5.12.12 DtCallback()

This function is called, if `stc_dt_config_t::pfnCallback` is defined.

Callback Function
`void DtCallback( void )`

5.12.13 Example Code

The following code excerpt examples shows how to handle the Dual Timer functions.

Note that because each instance of the Dual Timer has 2 channels, also 2 individual callback functions have to be defined, if both channels are used. Although the same interrupt vector is used the driver takes care of calling the correct callback function. The priority in case of simultaneous interrupt occurrence is channel 0 is prior to channel 1.

5.12.13.1 Configuration Initialization

```
#include "dt.h"

void Dt0Callback0(void)
{
  // some code here ...
}

void Dt0Callback1(void)
{
  // some code here ...
}

function
{
  stc_dt_channel_config_t stcDtConfig0;
  stc_dt_channel_config_t stcDtConfig1;

  L3_ZERO_STRUCT(stcDtConfig0);
  L3_ZERO_STRUCT(stcDtConfig1);

  // Configure Dual Timer Channel 0
  stcDtConfig0.enMode                = DtPeriodic;
  stcDtConfig0.enPrescalerDiv        = DtPrescalerDiv256;
  stcDtConfig0.enCounterSize         = DtConterSize32;
  stcDtConfig0.bEnable               = FALSE;
  stcDtConfig0.u32DtLoad             = 0xF2345670;  // just a value ...
  stcDtConfig0.u32DtBakgroundLoad = 0xF4567890;  // just a value...
  stcDtConfig0.pfnCallback           = &Dt0Callback0;

  // Configure Dual Timer Channel 1
  stcDtConfig1.enMode                = DtOneShot;
  stcDtConfig1.enPrescalerDiv        = DtPrescalerDiv256;
  stcDtConfig1.enCounterSize         = DtConterSize32;
  stcDtConfig1.bEnable               = FALSE;
  stcDtConfig1.u32DtLoad             = 0xE5667780;  // just a value...
  stcDtConfig1.u32DtBakgroundLoad = 0xE1223340;  // just a value...
  stcDtConfig1.pfnCallback           = &Dt0Callback1;

  // DT functions ...

       . . .
}
```

It is recommended for a certain start time, not to enable a channel from beginning, so let `bEnable = FALSE` in the configuration. This configuration is independent from `Dt_EnableChannel`() or `Dt_DisableChannel`().

5.12.13.2 Initialization, Parameter Change, and De-Initialization

```
uint32_t   Main_u32ReadValue ;

void Dt0Callback0(void); // Same as above
void Dt0Callback1(void); // Same as above

function
{
  stc_dtn_t* pstcDt = NULL ;
  stc_dt_channel_config_t stcDtConfig0;
  stc_dt_channel_config_t stcDtConfig1;

  L3_ZERO_STRUCT(stcDtConfig0);
  L3_ZERO_STRUCT(stcDtConfig1);

       // Same code as above ...

  // Init Dual Timer
  if (Ok == Dt_Init((stc_dtn_t*)&DT0, TRUE))
  {
     // Set handle
     pstcDt = (stc_dtn_t*)&DT0;

     // Set Channel 0 and enable
     Dt_SetChannelConfig(pstcDt, &stcDtConfig0, 0);
     Dt_EnableChannel(pstcDt, 0);

     // Set Channel 1 and enable
     Dt_SetChannelConfig(pstcDt, &stcDtConfig1, 1);
     Dt_EnableChannel(pstcDt, 1);

     . . .

     // Retrigger one-shot
     Dt_WriteLoadChannel(pstcDt, &stcDtConfig1, 1);

     . . .

     // Read Channel 1
     Main_u32ReadValue = Dt_ReadValueChannel(pstcDt, &stcDtConfig1, 1);

     . . .

     // Disable Dual Timer
     Dt_DisableChannel(pstcDt, 0);
     Dt_DisableChannel(pstcDt, 1);
     Dt_DeInit(pstcDt, TRUE);
}
```

In this code example the Dual Timer's channel 0 was set in chapter 5.12.13.1 to periodic mode and thus only needs a single `Dt_EnableChannel()` function. The other channel was set to one-shot mode, so it can be retriggered by the function `Dt_WriteLoadChannel()`.

5.13 (EMAC, ETHPHY) Ethernet Macro, Ethernet PHY

For the Ethernet functionality please refer to Ethernet application notes:

- AN706-00056-2v0-E

  Running the free licensed open-source TCP/IP stack LwIP (Lightweight IP) including an interactive webserver and a small AJAX example over Ethernet on FM3.

- AN706-00059-1V0-E

  The Ethernet Driver is used as a low level driver for FM3 devices. It is available as a stand- alone version or integrated within the FM3 Low Level Library (LLL). This is the user manual.

- AN706-00066-1v0-E

  This AppNote gives background information about the Ethernet Switch code example.

5.14 (EXINT) External Interrupts and NMI

Type Definition	–
Configuration Type	`stc_exint_config_t`
Address Operator	–

With `Exint_Init()` the external interrupts are enabled as given in the configuration. Note, that both external interrupt groups (0-7 and 8-15) can be enabled individually.

If a single channel may be disabled and re-enabled during runtime the functions `Exint_DisableChannel()` and `Exint_EnableChannel()` can be used.

The external interrupts are disabled globally by `Exint_DeInit()`.

Each channel provides an individually callback function which is handled in the interrupt service routine.

`Exint_Nmi_Init()` "initializes" the None-Maskable Interrupt, which means that the callback function is set to the dedicated internal data and if set by `stc_exint_nmi_config_t::bTouchNVIC` the NVIC is initialized. `Exint_Nmi_DeInit()` resets the NVIC depending on the setting in `stc_exint_nmi_config_t::bTouchNVIC`.

Note:

- The NMI shares with the Hardware Watchdog Timer the same interrupt vector.
- The driver recently only support External Interrupt channels from 0 – 15

5.14.1 Configuration Structures

5.14.1.1 External Interrupts

The EXINT module uses the following channel configuration structure of the type `stc_exint_config_t` for external interrupt except NMI:

Type	Field	Possible Values	Description

`boolean_t`	`abEnable…`
	`[L3_EXINT_CHANNELS]`
		`TRUE`
		`FALSE`
			Ext-Int. channel enabled
			Ext-Int. channel disabled

`en_exint_…`
`level_t`
	`aenLevel…`
	`[L3_EXINT_CHANNELS]`
		`ExIntLowLevel`
		`ExIntHighLevel`
		`ExIntRisingEdge`
		`ExIntFallingEdge`
			Low Level detection
			High Level detection
			Rising Edge detection
			Falling Edge detection

`func_ptr_t`	`apfnExintCallback…`
	`[L3_EXINT_CHANNELS]`
		–	Array of callback function pointers for each channel

`L3_EXINT_CHANNELS` is defined in *l3_user.h* and determines the maximum number of available external interrupt channels.

5.14.1.2 None-maskable Interrupt (NMI)

The EXINT module uses the following channel configuration structure of the type `stc_exint_nmi_config_t` for NMI:

Type	Field	Possible Values	Description
`boolean_t`	`bTouchNVIC`	`TRUE` `FALSE`	NVIC is touched NVIC is not touched
`func_ptr_t`	`pfnNMICallback`	-	Callback function pointer for NMI

Please also read the NMI handling during RAMCODE execution in chapter5.23.

5.14.2 Exint_Init()

This function sets the External Interrupt according the configuration structure of the type of `stc_exint_config_t`.

Prototype	
`en_result_t Exint_Init(stc_exint_config_t* pstcConfig)`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to EXINT configuration structure
Return Values	**Description**
`Ok`	External Interrupts successfully initialized
`ErrorInvalidParameter`	• `pstcConfig == NULL` • Illegal Mode

5.14.3 Exint_DeInit()

This function de-initializes all External Interrupts.

Prototype	
`en_result_t Exint_DeInit(void)`	
Return Values	**Description**
`Ok`	All External Interrupts successfully de-initialized

5.14.4 Exint_EnableChannel()

To change the activation state of an External Interrupt during runtime, this function can be used to enable an individual channel. `Exint_Init()` with the correct configuration must be called before.

Prototype	
`en_result_t Exint_EnableChannel(uint8_t u8Channel)`	
Parameter Name	**Description**
`[in] u8Channel`	Channel of External Interrupt
Return Values	**Description**
`Ok`	External Interrupt channel successfully enabled
`ErrorInvalidParameter`	A not activated channel was tried to be enabled

5.14.5 Exint_DisableChannel()

To change the activation state of an External Interrupt during runtime, this function can be used to disable an individual channel. `Exint_Init()` with the correct configuration must be called before.

Prototype	
`en_result_t Exint_DisableChannel(uint8_t u8Channel)`	
Parameter Name	**Description**
`[in] u8Channel`	Channel of External Interrupt
Return Values	**Description**
`Ok`	External Interrupt channel successfully disabled
`ErrorInvalidParameter`	A not activated channel was tried to be disabled

5.14.6 Exint_ClearChannel()

This function clears an interrupt of a certain channel. It may be called before enabling a channel to prevent an immediate interrupt.

Prototype	
`en_result_t Exint_ClearChannel(uint8_t u8Channel)`	
Parameter Name	**Description**
`[in] u8Channel`	Channel of External Interrupt
Return Values	**Description**
`Ok`	External Interrupt channel successfully cleared
`ErrorInvalidParameter`	A not activated channel was tried to be cleared

5.14.7 Exint_Nmi_Init()

This function sets the NMI according the configuration structure of the type of `stc_exint_nmi_config_t`.

Note:

The NMI shares the interrupt vector with the Hardware Watchog. To avoid cross influence of initialization of the NVIC use `stc_exint_nmi_config_t::bTouchNVIC` to determine whether to touch the corresponding NVIC registers or not.

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

5.14.8 Exint_Nmi_DeInit()

This function disables the NMI.

Note:

The NMI shares the interrupt vector with the Hardware Watchog. To avoid cross influence of de-initialization of the NVIC use `stc_exint_nmi_config_t::bTouchNVIC` to determine whether to touch the corresponding NVIC registers or not.

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

5.14.9 ExIntCallback()

This function is called, if `stc_exint_config_t::apfnExintCallback[n]` is defined.

Callback Function
`void ExIntCallbackn( void )`

5.14.10 NmiCallback()

This function is called, if `stc_nmi_config_t::pfnNMICallbackis` is defined.

Callback Function
`void NmiCallback( void )`

5.14.11 Example Code

5.14.11.1 External Interrupts

The following code excerpt shows how to use the EXINT functions. Assume External Interrupt channel 0 and 1 should be set to falling edge sensitivity. Both should get an own callback function.

```
#include "exint.h"

void ExintCallback0(void)
{
  // Do something ...
}

void ExintCallback1(void)
{
  // Do something ...
}

function0
{
  stc_exint_config_t stcExintConfig;

  L3_ZERO_STRUCT(stcExintConfig);

  stcExintConfig.abEnable[0]           = TRUE;
  stcExintConfig.aenLevel[0]           = ExIntFallingEdge;
  stcExintConfig.apfnExintCallback[0] = &ExintCallback0;

  stcExintConfig.abEnable[1]           = TRUE;
  stcExintConfig.aenLevel[1]           = ExIntFallingEdge;
  stcExintConfig.apfnExintCallback[1] = &ExintCallback1;

  Exint_Init(&stcExintConfig);

  . . .

}
```

5.14.11.2 NMI

The following code excerpt shows how to initialize the NMI. Assume the NVIC should be initialized by the init function and the NMI should get a callback function.

```
#include "exint.h"

void NmiCallback(void)
{
  // Do something ...
}

function0
{
  stc_exint_nmi_config_t stcExintNmiConfig;

  L3_ZERO_STRUCT(stcExintNmiConfig);

  stcExintNmiConfig.bTouchNVIC        = TRUE;
  stcExintNmiConfig.apfnExintCallback = &NmiCallback;

  Exint_Nmi_Init(&stcExintNmiConfig);

  . . .

}
```

Please also see NMI handling, when RAMCODE is executed in chapter 5.23.1.

5.15 (EXTIF) External Bus Interface

Type Definition	–
Configuration Type	`stc_extif_area_config_t`
Address Operator	–

An chip select area is configured with all its parameter in a structure of the type `stc_extif_area_config_t`. This area is then initialized by calling `Extif_InitArea()` together with the chip select number (area number) and the pointer to the configuration. `Extif_DeInitArea()` clears all register of an chip select area.

Note:

- This driver does not check the availability of a chip select area of a certain MCU. The user has to take care of using the chip select areas correctly.
- For Type0 device please check `SetPinFunc…()` macros, which contain accesses to `EPFR11`. Do ***not*** use them for your code! Possible missing pins are enabled by setting `EPFR10_UEFEFB` to `1`.

Attention:

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:

```
bL3_GPIO_EPFR11_UERLC = 1;
```

5.15.1 Configuration Structure

An external memory area has the following configuration structure of the type of `stc_extif_area_config_t`:

Type	Field	Possible Values	Description
`en_extif_width_t`	`enWidth`	`Extif8Bit` `Extif16Bit`	8-Bit data bus 16-Bit data bus
`oolean_t`	`bMultplexMode`	`TRUE` `FALSE`	Multiplex Mode Non-multiplex Mode
`oolean_t`	`bReadByteMask`	`TRUE` `FALSE`	Read Byte Mask enable Read Byte Mask disable
`oolean_t`	`bWriteEnableOff`	`TRUE` `FALSE`	Write enable disabled Write enable supported
`oolean_t`	`bNandFlash`[1]	`TRUE` `FALSE`	NAND Flash bus enable NOR Flash/SRAM bus enable
`oolean_t`	`bPageAccess`	`TRUE` `FALSE`	NOR Flash Page access mode Normal bus mode
`oolean_t`	`bRdyOn`[2]	`TRUE` `FALSE`	RDY mode enabled No RDY mode
`oolean_t`	`bStopDataOut…` `AtFirstIdle`	`TRUE` `FALSE`	Stop wr data at 1st idle cycle Extend wr data to last idle

			cycle
oolean_t	bAleInvert[2]	TRUE FALSE	Invert ALE signal ALE signal high active
oolean_t	bAddrOnData… LinesOff[2]	TRUE FALSE	Adr/Data Hi-Z during ALC cycle No Hi-Z cycle
oolean_t	bMpxcsOff[2]	TRUE FALSE	No MCSX during ALC cycle MCSX during ALC cycle
oolean_t	bMoexWidthAsFradc[2]	TRUE FALSE	MOEX width with FRADC MOEX width with RACC-RADC
en_extif_cycle_t	enWriteIdleCycle	Extif0Cycle Extif1Cycle Extif2Cycle … Extif15Cycle Extif16Cycle ExtifDisabled	Write Idle Cycle Timing
en_extif_cycle_t	enWriteEnableCycle	*see above*	Write Enable Cycle Timing
en_extif_cycle_t	enWriteAddress… SetupCycle	*see above*	Write Address Setup Cycle Timing
en_extif_cycle_t	enWriteAccessCycle	*see above*	Write Access Cycle Timing
en_extif_cycle_t	enReadIdleCycle	*see above*	Read Idle Cycle Timing
en_extif_cycle_t	enFirstRead… AddressCycle	*see above*	First Read Address Cycle Timing
en_extif_cycle_t	enReadAddress… SetupCycle	*see above*	Read Address Setup Cycle Timing
en_extif_cycle_t	enReadAccessCycle	*see above*	Read Address Cycle Timing
en_extif_cycle_t	enAdressLatch… EnableCycle[2]	*see above*	Address Latch Enable Cycle Timing
en_extif_cycle_t	enAddressLatch… EnableSetupCycle[2]	*see above*	Address Latch Enable Setup Cycle Timing
en_extif_cycle_t	enAddressLatch… Cycle[2]	*see above*	Address Latch Cycle Timing
oolean_t	bMclkDivision… Enable[2]	TRUE FALSE	Enable MCLK division Disable MCLK division
uint8_t	u8MclkDivision[2]	0 1 ... 14 15	MCLK divided by 1 MCLK divided by 2 … MCLK divided by 15 MCLK divided by 16
en_extif_mask_t	enAreaMask	Extif1MB Extif2MB Extif4MB ... Extif64MB Extif128MB	1 Mbyte external Area 2 Mbyte external Area 4 Mbyte external Area … 64 Mbyte external Area 128 Mbyte external Area
uint8_t	u8AreaAddress	-	Address Bits [27:20]

[1] Only available at devices which support NAND Flash I/F (auto-checked by LLL)
[2] Only available for Device Type1

5.15.2 Extif_InitArea()

This function initializes an external memory area with the given configuration of type of `stc_extif_area_config_t`.

Prototype	
`en_result_t Extif_InitArea( uint8_t u8Area, stc_extif_area_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] u8Area`	Area (chip select) number. Must be correspond to supported area of device!
`[in] pstcConfig`	External area configuration
Return Values	**Description**
`Ok`	External Interrupts successfully initialized
`ErrorInvalidParameter`	• `pstcConfig == NULL` • Area number exceeds `EXTIF_MAX_AREA` • Other illegal settings

5.15.3 Extif_DeInitArea()

This function clears an external memory area.

Prototype	
`en_result_t Extif_DeInitArea( uint8_t u8Area )`	
Parameter Name	**Description**
`[in] u8Area`	Area (chip select) number. Must be correspond to supported area of device!
Return Values	**Description**
`Ok`	External Interrupts successfully initialized
`ErrorInvalidParameter`	Area number exceeds `EXTIF_MAX_AREA`

5.15.4 Extif_Nand_SetCommand()

This pseudo function sets an 8-Bit command cycle to the external bus, if configured as NAND Flash bus.

Definition

```
#define Extif_Nand_SetCommand(base, cmd)                              \
   {*(volatile unsigned char*)(base + EXITIF_NAND_CMD_OFFSET) =       \
   (unsigned char)(cmd);}
```

Parameter Name	Description
`[in] base`	Base address of used chip select area
`[in] cmd`	NAND Flash command. Refer to NAND Flash data sheet for details.

5.15.5 Extif_Nand_SetAddress()

This pseudo function sets an 8-Bit address cycle to the external bus, if configured as NAND Flash bus. The number of “calls” to this function depends on the used address bit width of the NAND Flash memory.

Definition	
`#define Extif_Nand_SetAddress(base, adr) \ {*(volatile unsigned char*)(base + EXITIF_NAND_ADDR_OFFSET) = \ (unsigned char)(addr);}`	
Parameter Name	**Description**
`[in] base`	Base address of used chip select area
`[in] adr`	8-Bit address (part).

5.15.6 Extif_Nand_ReadData()

This pseudo function reads 8-Bit data from the external bus, if configured as NAND Flash bus.

Definition	
`#define Extif_Nand_ReadData(base) \ (*(volatile unsigned char*)(base + EXITIF_NAND_DATA_OFFSET))`	
Parameter Name	**Description**
`[in] base`	Base address of used chip select area
Return Value	**Description**
`char`	8-Bit read data

5.15.7 Extif_Nand_WriteData()

This pseudo function writes 8-Bit data to the external bus, if configured as NAND Flash bus.

Definition	
`#define Extif_Nand_WriteData(base, data) \ {*(volatile unsigned char*)(base + EXITIF_NAND_DATA_OFFSET) = \ (unsigned char)(data);}`	
Parameter Name	**Description**
`[in] base`	Base address of used chip select area
`[in] data`	8-Bit data to be written

5.15.8 Extif_Nand_ClearAle()

This pseudo function de-asserts the `ALE` (`MNALE`) signal.

Definition	
`#define Extif_Nand_ClearAle(base) \ {*(volatile unsigned char*)(base + EXITIF_NAND_ALE_OFFSET) = \ (unsigned char)0;}`	
Parameter Name	**Description**
`[in] base`	Base address of used chip select area

5.15.9 Example Codes

5.15.9.1 SRAM

The following code excerpt shows how to use the EXTIF functions for SRAM connected to chips select 1.

```
#include "extif.h"

#define EXTIF_ADDRESS 0x61000000  // External memory connected to CSX1

function
{
  stc_extif_area_config_t stcExtIF;

  L3_ZERO_STRUCT(stcExtIF);

  // Initialize pins here: MDATA0, ..., MDATA7; MAD00, ..., MAD24,
  // MCSX1

  // Type 0 device is used, so not all configuration is done yet.
  stcExtIF.enWidth                    = Extif8Bit;
  stcExtIF.bMultplexMode              = FALSE;   // Not available for Type 0
  stcExtIF.bReadByteMask              = FALSE;
  stcExtIF.bWriteEnableOff            = FALSE;
#if (L3_NAND_SUPPORT == L3_ON)
  stcExtIF.bNandFlash                 = FALSE;   // Not available for Type 0
#endif
  stcExtIF.bPageAccess                = FALSE;
  stcExtIF.enWriteIdleCycle           = Extif3Cycle;
  stcExtIF.enWriteEnableCycle         = Extif3Cycle;
  stcExtIF.enWriteAddressSetupCycle = Extif3Cycle;
  stcExtIF.enWriteAccessCycle         = Extif3Cycle;
  stcExtIF.enReadIdleCycle            = Extif3Cycle;
  stcExtIF.enFirstReadAddressCycle   = Extif3Cycle;
  stcExtIF.enReadAddressSetupCycle   = Extif3Cycle;
  stcExtIF.enReadAccessCycle          = Extif3Cycle;
  stcExtIF.enAreaMask                 = Extif1MB;
  stcExtIF.u8AreaAddress              = (uint8_t)(EXTIF_ADDRESS >> 20);

  if (Ok == Extif_InitArea(1, &stcExtIF))        // CS1
  {
    *(uint16_t*)EXTIF_ADDRESS = 0x1234;          // First memory address
    u16SramTest = *((uint16_t*) EXTIF_ADDRESS); // Read back

    if (u16SramTest != 0x1234)
    {
      // Error handling ...
    }

    Extif_DeInitArea(1);
  }
}
```

5.15.9.2 NAND Flash

The following code excerpt shows how to use the EXTIF functions for NAND Flash connected to chips select 0.

```
#include "extif.h"

#define EXTIF_NAND_ADDRESS 0x60000000  // External NAND Flash connected to CS0
#define EXTIF_AREA         0

// NAND Flash commands
#define NAND_CMD_READ0     0x00
#define NAND_CMD_READ1     0x01
#define NAND_CMD_PAGEPROG  0x10
#define NAND_CMD_READOOB   0x50
#define NAND_CMD_ERASE1    0x60
#define NAND_CMD_STATUS    0x70
#define NAND_CMD_SEQIN     0x80
#define NAND_CMD_READID    0x90
#define NAND_CMD_READID1   0x91
#define NAND_CMD_ERASE2    0xD0
#define NAND_CMD_RESET     0xFF

#define NAND_FLASH_OK      0
#define NAND_FLASH_ERROR   1

// Helper functions

static void Extif_Wait(volatile uint32_t u32Count)
{
    while(u32Count--);
}

static void Nand_Reset(void)
{
    Extif_Nand_SetCommand(EXTIF_NAND_ADDRESS, NAND_CMD_RESET);
    Extif_Wait(10000);                          // wait for Trst (depends on HCLK)
}

static unsigned char Nand_ReadStatus(void)
{
  uint32_t u32Timeout = 0;

  Extif_Nand_SetCommand(EXTIF_NAND_ADDRESS, NAND_CMD_STATUS);

  while(!(Extif_Nand_ReadData(EXTIF_NAND_ADDRESS) & 0x40))
  {
    u32Timeout++;

    if(0x00080000 == u32Timeout)
    {
      return NAND_FLASH_ERROR;
    }
  }

  if(Extif_Nand_ReadData(EXTIF_NAND_ADDRESS) & 0x01)
  {
    return NAND_FLASH_ERROR;
  }

  return NAND_FLASH_OK;
}
```
▼

```
static void Nand_ReadID(uint8_t *u8MarkerCode, uint8_t *u8Id)
{
    Extif_Nand_SetCommand(EXTIF_NAND_ADDRESS, NAND_CMD_READID);
    Extif_Nand_SetAddress(EXTIF_NAND_ADDRESS, 0x00);
    Extif_Nand_ClearAle(EXTIF_NAND_ADDRESS);
    Extif_Wait(10);
    *u8MarkerCode = Extif_Nand_ReadData(EXTIF_NAND_ADDRESS);
    *u8Id         = Extif_Nand_ReadData(EXTIF_NAND_ADDRESS);
}

static uint32_t Nand_EraseBlock(uint32_t u32Block)
{
    uint32_t u32BlockPage = (u32Block << 5);

    Extif_Nand_SetCommand(EXTIF_NAND_ADDRESS, NAND_CMD_ERASE1); // erase command
    Extif_Nand_SetAddress(EXTIF_NAND_ADDRESS, 0);
    Extif_Nand_SetAddress(EXTIF_NAND_ADDRESS, u32BlockPage & 0xFF);
    Extif_Nand_SetAddress(EXTIF_NAND_ADDRESS, (u32BlockPage >> 8) & 0xFF);
    // Possible further address cycles here ...
    Extif_Nand_SetCommand(EXTIF_NAND_ADDRESS, NAND_CMD_ERASE2); // start erase

    if(NAND_FLASH_ERROR == Nand_ReadStatus())
    {
       Nand_Reset();
       return NAND_FLASH_ERROR;
    }

    return NAND_FLASH_OK;
}

static void Nand_ReadPage(uint32_t u32Block, uint32_t u32Page, uint8_t *pu8Buf)
{
    uint8_t* pu8BufPtr = pu8Buf;
    uint32_t u32BlockPage;
    uint32_t u32i;
    volatile uint32_t au32Se[16];

    u32BlockPage = (u32Block << 5) + u32Page;      // 1 block = 32 pages
    Extif_Nand_SetCommand(EXTIF_NAND_ADDRESS, NAND_CMD_READ0); // send read data
    Extif_Nand_SetAddress(EXTIF_NAND_ADDRESS, 0);
    Extif_Nand_SetAddress(EXTIF_NAND_ADDRESS, u32BlockPage & 0xFF);
    Extif_Nand_SetAddress(EXTIF_NAND_ADDRESS, (u32BlockPage >> 8) & 0xFF);
    // Possible further address cycles here ...
    Extif_Nand_ClearAle(EXTIF_NAND_ADDRESS);

    Extif_Wait(500);

    for(u32i = 0; u32i < 512; u32i++)
    {
        // read 512 bytes
        pu8BufPtr[u32i] = Extif_Nand_ReadData(EXTIF_NAND_ADDRESS);
    }
    for(u32i = 0; u32i < 16; u32i++)
    {
        // read dummy 16 byte
        au32Se[u32i] = Extif_Nand_ReadData(EXTIF_NAND_ADDRESS);
    }
}
```

▲

```
uint32_t Nand_WritePage(uint32_t u32Block, uint32_t u32Page, uint8_t *pu8Buf)
{
    uint8_t* pu8BufPtr = pu8Buf;
    uint32_t u32BlockPage;
    uint32_t u32i;
    uint32_t au32Se[16];
    uint8_t  u8Data;

    u32BlockPage=(u32Block << 5) + u32Page;

    Extif_Nand_SetCommand(EXTIF_NAND_ADDRESS, 0x00);     // set programming area
    Extif_Nand_SetCommand(EXTIF_NAND_ADDRESS, NAND_CMD_SEQIN); // write command
    Extif_Nand_SetAddress(EXTIF_NAND_ADDRESS, 0);
    Extif_Nand_SetAddress(EXTIF_NAND_ADDRESS, u32BlockPage & 0xFF);
    Extif_Nand_SetAddress(EXTIF_NAND_ADDRESS, (u32BlockPage >> 8) & 0xFF);
    // Possible further address cycles here ...
    Extif_Nand_ClearAle(EXTIF_NAND_ADDRESS);

    for(u32i = 0; u32i < 512; u32i++)
    {
      Extif_Nand_WriteData(EXTIF_NAND_ADDRESS, *pu8BufPtr++); // write data
    }

    for(u32i = 0; u32i < 16; u32i++)
    {
      Extif_Nand_WriteData(EXTIF_NAND_ADDRESS, au32Se[u32i]); // dummy write
    }

    // start programming
    Extif_Nand_SetCommand(EXTIF_NAND_ADDRESS, NAND_CMD_PAGEPROG);

    if(NAND_FLASH_ERROR == Nand_ReadStatus())
    {
       return NAND_FLASH_ERROR;
    }

    // verify the write data
    pu8BufPtr = pu8Buf;

    Extif_Nand_SetCommand(EXTIF_NAND_ADDRESS, NAND_CMD_READ0); // read command
    Extif_Nand_SetAddress(EXTIF_NAND_ADDRESS, 0);
    Extif_Nand_SetAddress(EXTIF_NAND_ADDRESS, u32BlockPage & 0xFF);
    Extif_Nand_SetAddress(EXTIF_NAND_ADDRESS, (u32BlockPage >> 8) & 0xFF);
    // Possible further address cycles here ...
    Extif_Nand_ClearAle(EXTIF_NAND_ADDRESS);

    Extif_Wait(500);

    for(u32i = 0; u32i < 512; u32i++)
    {
      u8Data = Extif_Nand_ReadData(EXTIF_NAND_ADDRESS); // verify 1-512 byte

      if(u8Data != pu8BufPtr[u32i])
      {
        return NAND_FLASH_ERROR;
      }
    }

    for(u32i = 0; u32i < 16; u32i++)
    {
      // verify 16 byte dummy data
      u8Data = Extif_Nand_ReadData(EXTIF_NAND_ADDRESS);
```

▼

▲
```
      if(u8Data != au32Se[u32i])
      {
        return NAND_FLASH_ERROR;
      }
    }

    return NAND_FLASH_OK;
}
```

function

```
{
  stc_extif_area_config_t  stcExtIF;
  uint8_t                  u8MarkerCode     = 0xAA;
  uint8_t                  u8Id             = 0x55;
  uint8_t                  au8Buffer[512];
  uint32_t                 u32i;
  uint32_t                 u32Page;
  uint32_t                 u32Block;

  L3_ZERO_STRUCT(stcExtIF);

  SetPinFunc_MADATA00_0();   // NAND I/O0
  SetPinFunc_MADATA01_0();   // NAND I/O1
  SetPinFunc_MADATA02_0();   // NAND I/O2
  SetPinFunc_MADATA03_0();   // NAND I/O3
  SetPinFunc_MADATA04_0();   // NAND I/O4
  SetPinFunc_MADATA05_0();   // NAND I/O5
  SetPinFunc_MADATA06_0();   // NAND I/O6
  SetPinFunc_MADATA07_0();   // NAND I/O7

  SetPinFunc_MCSX0_0();      // NAND CEx
  SetPinFunc_MNALE_0();      // NAND ALE
  SetPinFunc_MNCLE_0();      // NAND CLE
  SetPinFunc_MNREX_0();      // NAND REx
  SetPinFunc_MNWEX_0();      // NAND WEx

  stcExtIF.enWidth                     = Extif8Bit;
  stcExtIF.bMultplexMode               = FALSE;
  stcExtIF.bReadByteMask               = FALSE;
  stcExtIF.bWriteEnableOff             = FALSE;
  stcExtIF.bNandFlash                  = TRUE;
  stcExtIF.bPageAccess                 = FALSE;
  stcExtIF.bRdyOn                      = FALSE;
  stcExtIF.enWriteIdleCycle            = Extif3Cycle;
  stcExtIF.enWriteEnableCycle          = Extif3Cycle;
  stcExtIF.enWriteAddressSetupCycle = Extif3Cycle;
  stcExtIF.enWriteAccessCycle          = Extif3Cycle;
  stcExtIF.enReadIdleCycle             = Extif3Cycle;
  stcExtIF.enFirstReadAddressCycle   = Extif3Cycle;
  stcExtIF.enReadAddressSetupCycle   = Extif3Cycle;
  stcExtIF.enReadAccessCycle           = Extif3Cycle;
  stcExtIF.enAddressLatchCycle        = Extif3Cycle;
  stcExtIF.u8MclkDivision              = 4;
  stcExtIF.enAreaMask                  = Extif1MB;
  stcExtIF.u8AreaAddress               = (uint8_t)(EXTIF_NAND_ADDRESS >> 20);

  for (u32i = 0; u32i < 512; u32i++)     // Clear buffer
  {
    au8Buffer[u32i] = 0;
  }

  if (Ok == Extif_InitArea(EXTIF_AREA, &stcExtIF))
  {
    Nand_ReadID(&u8MarkerCode, &u8Id);  // Read NAND Flash ID
```
▼

```
                                                                        ▲
    u32Block = 0;     // Just a value ...
    u32Page  = 11;    // Just a value ...

    if (NAND_FLASH_ERROR == Nand_EraseBlock(u32Block))
    {
      // Erase failed!
    }

    Nand_ReadPage(u32Block, u32Page, &au8Buffer[0]);

    // au8Buffer[] now should contain all 0xFF (erased NAND Flash cells)

    for (u32i = 0; u32i < 512; u32i++)  // Make some data ...
    {
      au8Buffer[u32i] = (uint8_t)(0xFF & u32i);
    }

    if (NAND_FLASH_ERROR == Nand_WritePage(u32Block, u32Page, &au8Buffer[0]))
    {
      // Write failed!
    }

    for (u32i = 0; u32i < 512; u32i++)  // Clear buffer
    {
      au8Buffer[u32i] = 0;
    }

    Nand_ReadPage(u32Block, u32Page, &au8Buffer[0]);

    // au8Buffer[] now should contain written data

    if (NAND_FLASH_ERROR == Nand_EraseBlock(u32Block))
    {
      // Erase failed
    }

    Nand_ReadPage(u32Block, u32Page, &au8Buffer[0]);

    // au8Buffer[] now should contain all 0xFF (erased NAND Flash cells)

    . . .

  }
}
```

5.16 (FLASH) Flash Erase and Programming Routines

Type Definition	–
Configuration Type	–
Address Operator	–

`Flash_Erase()` erases a sector with the given address. `Flash_Write8()` and `Flash_Write16()` program a single byte or 16-bit word to a given Flash address. Within the functions it is checked, if a Flash bit cell is tried to program from '`0`' to '`1`', which is not possible. An error is returned in this case and the Flash address is not programmed. `Flash_WriteBlock8()` and `Flash_WriteBlock16()` program a block of data with a given length. The write functions from above are used.

Caution:

If the used device has an ECC Flash the only allowed function calls are:

- Flash_Erase()
- Flash_WriteEccBlock()

Notes:

- If `L3_FLASH_DMA_HALT == L3_ON` and a Flash routine returns `ErrorTimeout` the application has to check, whether possible DMA transfers are properly resumed again.
- If `L3_NMI_RAM_VECTOR_HANDLE == L3_ON` the NMI vector is temporarily linked to RAM area during RAMCODE execution. If `stcExintNMIInternData.pfnNMICallback != NULL` the NMI callback is used. Therefore also activate `L3_PERIPHERAL_ENABLE_EXINT0_7` in *l3_user.h*. Read chapter 5.13 for more details.

5.16.1 Flash_Erase()

This function erases a Flash sector. If `u32FlashAddress` is within the Work Flash memory area the corresponding ROM Flash routines are called (if Work Flash is available and its area addressed).

Prototype	
`en_result_t Flash_Erase( uint32_t u32FlashAddress )`	
Parameter Name	**Description**
`[in] u32FlashAddress`	An address within the Flash sector to be erased
Return Values	**Description**
`Ok`	Flash erase was successful
`ErrorTimeout`	Automatic erase algorithm timed out
`ErrorInvalidParameter`	If `L3_NO_FLASH_RAMCODE == L3_ON` and address not within Work Flash memory area

5.16.2 Flash_Write8()

This function writes a byte to the Flash memory. If `u32FlashAddress` is within the Work Flash memory area the corresponding ROM Flash routines are called (if Work Flash is available and its area addressed).

Prototype	

`en_result_t Flash_Write8( uint32_t u32FlashAddress,`
`                           uint8_t  u8Data )`	

Parameter Name	**Description**
`[in] u32FlashAddress`	Flash address
`[in] u8Data`	8-Bit Flash data to be programmed
Return Values	**Description**
`Ok`	Flash programming was successful
`ErrorTimeout`	Automatic programming algorithm timed out
`ErrorOperationInProgress`	Flash not ready to be programmed
`ErrorInvalidParameter`	• Flash data cannot be programmed, because one or more bits cannot be programmed from ‘`0`’ to ‘`1`’. • `L3_NO_FLASH_RAMCODE == L3_ON` and address not within Work Flash memory area.

Note, this function is only able to (re)program a lower or upper byte of a 16-bit Flash memory word, if the content equals to `0xFF`. Otherwise an error is returned.

5.16.3 Flash_Write16()

This function writes a 16-Bit word to the Flash memory. If `u32FlashAddress` is within the Work Flash memory area the corresponding ROM Flash routines are called (if Work Flash is available and its area addressed).

The address `u32FlashAddress` *must* be 16-Bit aligned (even address).

Prototype	

`en_result_t Flash_Write16( uint32_t u32FlashAddress,`
`                            uint16_t u16Data )`	

Parameter Name	**Description**
`[in] u32FlashAddress`	Flash address
`[in] u16Data`	16-Bit Flash data to be programmed
Return Values	**Description**
`Ok`	Flash programming was successful
`ErrorTimeout`	Automatic programming algorithm timed out
`ErrorAddressAlignment`	u32FlashAddress not an even address
`ErrorOperationInProgress`	Flash not ready to be programmed
`ErrorInvalidParameter`	• Flash data cannot be programmed, because one or more bits cannot be programmed from ‘`0`’ to ‘`1`’. • `L3_NO_FLASH_RAMCODE == L3_ON` and address not within Work Flash memory area.

5.16.4 Flash_Block8()

This function writes a data block to the Flash memory. The data and count is 8-Bit width.

The return code `ErrorInvalidParameter` is returned when an input parameter is incorrect or if one data value can not be written because of an invalid ‘0‘ bit to ‘1‘ bit transition.

If `u32FlashAddress` is within the Work Flash memory area the corresponding ROM Flash routines are called (if Work Flash is available and its area addressed).

Prototype	
`en_result_t Flash_WriteBlock8( uint32_t u32FlashAddress, uint8_t* pu8Data, uint32_t u32DataLength )`	
Parameter Name	**Description**
`[in] u32FlashAddress`	Flash address
`[in] pu8Data`	Pointer to 8-Bit data block
`[in] u32DataLength`	8-Bit data block length
Return Values	**Description**
`Ok`	Flash programming was successful
`ErrorTimeout`	Automatic programming algorithm timed out
`ErrorOperationInProgress`	Flash not ready to be programmed
`ErrorInvalidParameter`	• `pu8Data == NULL` • `u32DataLength = 0` • Flash data cannot be programmed, because one or more bits cannot be programmed from ‘0’ to ‘1’. • `L3_NO_FLASH_RAMCODE == L3_ON` and address not within Work Flash memory area.

5.16.5 Flash_Block16()

This function writes a data block to the Flash memory. The data and count is 16-Bit width.

The return code `ErrorInvalidParameter` is returned when an input parameter is incorrect or if one data value can not be written because of an invalid ‘0‘ bit to ‘1‘ bit transition.

If `u32FlashAddress` is within the Work Flash memory area the corresponding ROM Flash routines are called (if Work Flash is available and its area addressed).

The address `u32FlashAddress` *must* be 16-Bit aligned (even address)

Prototype	
`en_result_t Flash_WriteBlock8( uint32_t u32FlashAddress, uint16_t* pu16Data, uint32_t u32DataLength )`	
Parameter Name	**Description**
`[in] u32FlashAddress`	Flash address
`[in] pu16Data`	Pointer to 16-Bit data block
`[in] u32DataLength`	16-Bit data block length
Return Values	**Description**
`Ok`	Flash programming was successful
`ErrorTimeout`	Automatic programming algorithm timed out
`ErrorAddressAlignment`	u32FlashAddress not an even address
`ErrorOperationInProgress`	Flash not ready to be programmed
`ErrorInvalidParameter`	• `pu8Data == NULL` • `u32DataLength = 0` • Flash data cannot be programmed, because one or more bits cannot be programmed from ‘`0`’ to ‘`1`’. • `L3_NO_FLASH_RAMCODE == L3_ON` and address not within Work Flash memory area.

5.16.6 Flash_WriteEccBlock()

This function writes a data block to the ECC Flash memory. The data and count is 16-Bit width.

The return code `ErrorInvalidParameter` is returned when an input parameter is incorrect or if one data value can not be written because of an not empty ECC 32-bit cell. ECC does not support Flash cell update without ECC error!

The address `u32FlashAddress` *must* be 16-Bit aligned (even address).

Note, this function is only available if the device supports ECC Flash, otherwise this function is excluded by the preprocessor.

Prototype	
`en_result_t Flash_WriteEccBlock ( uint32_t u32FlashAddress, uint16_t* pu16Data, uint32_t u32DataLength )`	
Parameter Name	**Description**
`[in] u32FlashAddress`	Flash address
`[in] pu16Data`	Pointer to 16-Bit data block
`[in] u32DataLength`	16-Bit data block length
Return Values	**Description**
`Ok`	Flash programming was successful
`ErrorTimeout`	Automatic programming algorithm timed out
`ErrorAddressAlignment`	u32FlashAddress not an even address
`ErrorOperationInProgress`	Flash not ready to be programmed
`ErrorInvalidParameter`	• `pu8Data == NULL` • `u32DataLength = 0` • Flash data cannot be programmed, because one or more bits cannot be programmed from ‘`0`’ to ‘`1`’. • `L3_NO_FLASH_RAMCODE == L3_ON` and address not within Work Flash memory area.

5.16.7 Example Code

The following code excerpt shows how to use the FLASH functions.

5.16.7.1 Devices with Single Flash

```
#include "flash.h"

function
{
  uint8_t  au8FlashPattern[8]  = {0, 1, 2, 3, 4, 5, 6, 7};
  uint16_t au16FlashPattern[4] = {0x1234, 0x2345, 0x3456, 0x4567};

  // Erase Sector SA6, SA7
  Flash_Erase(0x4000);
  Flash_Erase(0x4004);

  // Program 16-bit word to 0x4000
  Flash_Write16(0x4000, 0x1234);

  // Write even, then odd address
  Flash_Write8(0x4004, 0x55);
  Flash_Write8(0x4005, 0xAA);

  // Write even address, even # of data
  Flash_WriteBlock8(0x4008, au8FlashPattern, 8);

  // Write even address, odd # of data
  Flash_WriteBlock8(0x4010, au8FlashPattern, 3);

  // Write odd address, even # of data
  Flash_WriteBlock8(0x401D, au8FlashPattern, 2);

  // Write odd address, odd # of data
  Flash_WriteBlock8(0x4025, au8FlashPattern, 3);

  // Write Flash_WriteBlock16()
  Flash_WriteBlock16(0x4030, au16FlashPattern, 3);

  . . .

}
```

5.16.7.2 Devices with Work Flash

```
#include "flash.h"

function
{
  uint8_t  au8FlashPattern[8]  = {0, 1, 2, 3, 4, 5, 6, 7};
  uint16_t au16FlashPattern[4] = {0x1234, 0x2345, 0x3456, 0x4567};

  // Erase Sector SA6, SA7 with wait
  Flash_Erase(0x4000);
  Flash_Erase(0x4004);

  // Program 16-bit word to 0x4000
  Flash_Write16(0x4000, 0x1234);

  // Write even, then odd address
  Flash_Write8(0x4004, 0x55);
  Flash_Write8(0x4005, 0xAA);

  // Test even address, even # of data
  Flash_WriteBlock8(0x4008, au8FlashPattern, 2);

  // Test even address, odd # of data
  Flash_WriteBlock8(0x4010, au8FlashPattern, 3);

  // Test odd address, even # of data
  Flash_WriteBlock8(0x401D, au8FlashPattern, 2);

  // Test odd address, odd # of data
  Flash_WriteBlock8(0x4025, au8FlashPattern, 3);

  // Test of Flash_WriteBlock16()
  Flash_WriteBlock16(0x4030, au16FlashPattern, 3);

  // ****** Test Work Flash Area ******

  // Erase Sector SA0 with wait
  Flash_Erase(0x200C0000);

  // Write even, then odd address
  Flash_Write8(0x200C0004, 0x55);
  Flash_Write8(0x200C0005, 0xAA);

  . . .

}
```

5.16.7.3 Devices with Dual Operation Flash

```
#include "flash.h"

function
{
  uint8_t  au8FlashPattern[8]  = {0, 1, 2, 3, 4, 5, 6, 7};
  uint16_t au16FlashPattern[4] = {0x1234, 0x2345, 0x3456, 0x4567};

  // Erase Sector SA8 with wait (upper-bank)
  Flash_Erase(0x4000);

  // Program 16-bit word to 0x4000
  Flash_Write16(0x4000, 0x1234);

  // Write even, then odd address
  Flash_Write8(0x4004, 0x55);
  Flash_Write8(0x4005, 0xAA);

  // Test even address, even # of data
  Flash_WriteBlock8(0x4008, au8FlashPattern, 2);

  // Test even address, odd # of data
  Flash_WriteBlock8(0x4010, au8FlashPattern, 3);

  // Test odd address, even # of data
  Flash_WriteBlock8(0x401D, au8FlashPattern, 2);

  // Test odd address, odd # of data
  Flash_WriteBlock8(0x4025, au8FlashPattern, 3);

  // Test of Flash_WriteBlock16()
  Flash_WriteBlock16(0x4030, au16FlashPattern, 3);

  // Erase Sector SA4 with wait (lower-bank)
  Flash_Erase(0x00200000);

  // Write even, then odd address
  Flash_Write8(0x00200004, 0x55);
  Flash_Write8(0x00200005, 0xAA);

  . . .

}
```

5.17 (FLASHTIMING) Flash Timing Functions

Type Definition	–
Configuration Type	–
Address Operator	–

The following functions can set Flash timings for Main and Work Flash. Depending on the Device Type some functions are available and some not. The following table shows an overview of the functions and availability depending on the Device Type.

Function	**Device Type**									
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`Flashtiming_GetFlashBufferStatus()`			√		√					
`Flashtiming_EnableFlashBuffer()`			√		√					
`Flashtiming_DisableFlashBuffer()`			√		√					
`Flashtiming_SetFlashSyncDown()`	√	√	√	√	√	√		√		
`Flashtiming_SetFlashWait()`	√		√		√		√		√	√
`Flashtiming_SetWorkFlashWait()`					√	√				

5.17.1 Flashtiming_GetFlashBufferStatus()

This function returns the status of the Flash Accelerator Buffer.

Note, this function is only available for Device Types 2 and 4.

Prototype	
`en_flashtiming_flash_buffer_status_t Flashtiming_GetFlashBufferStatus ( void )`	
Return Values	**Description**
`FlashBufferStopInit`	Flash Buffer is in stop or initialization status and disabled.
`FlashBufferReady`	Flash Buffer is allowed to be enabled (state is not really meaningful, should never happen)
`FlashBufferEnable`	Flash Buffer is enabled, but still initialization status
`FlashBufferRunning`	Flash Buffer is enabled and running
`FlashBufferError`	Switch-Case default, will never occur

5.17.2 Flashtiming_EnableFlashBuffer ()

This function enables the Flash Accelerator Buffer.

Note, this function is only available for Device Types 2 and 4.

Prototype	
`en_result_t Flashtiming_EnableFlashBuffer( void )`	
Return Values	**Description**
`Ok`	Flash Buffer enabled

5.17.3 Flashtiming_DisableFlashBuffer ()

This function disables the Flash Accelerator Buffer.

Note, this function is only available for Device Types 2 and 4.

Prototype	
`en_result_t Flashtiming_DisableFlashBuffer( void )`	
Return Values	**Description**
`Ok`	Flash Buffer disabled

5.17.4 Flashtiming_SetFlashSyncDown ()

This function allows to slow down the Flash access with synchronous wait states for power management.

Note, this function is not available for Device Types 6, 8 and 9.

Prototype	
`en_result_t Flashtiming_SetFlashSyncDown( en_flashtiming_flash_sync_down_t enFlashSyncDown )`	
Parameter Name	**Description**
`[in] enFlashSyncDown`	Number of Wait States. Possible enumerators are: • `FlashWait0` • `FlashWait1` • `FlashWait3` • `FlashWait5` • `FlashWait7`
Return Values	**Description**
`Ok`	Flash wait state was set successfully
`ErrorInvalidParameter`	Wrong enumerator used

5.17.5 Flashtiming_SetFlashWait ()

This function allows to insert wait states, which may necessary for higher HCLK. For Device Type 2 and 4 also the Flash Accelerator has to be enabled by this function.

Note, this function is not available for Device Types 6, 8 and 9.

Prototype	
`en_result_t Flashtiming_SetFlashWait( en_flashtiming_flash_wait_mode_t enFlashMode )`	
Parameter Name	**Description**
`[in] enFlashMode`	Wait State Mode. Possible enumerators are: • `Flash0CycleMode` (All Device Types) • `FlashAcceleratorMode` (Type 2, 4) • `Flash2CycleMode` (Type 0, 5) • `FlashPreFetch20More` (Type 6, 8 for HCLK ≥20 MHz) • `FlashPreFetch20To40` (Type 9 for HCLK ≥20 MHz and <40MHz) • `FlashPreFetch40More` (Type 9 for HCLK ≥40 MHz)
Return Values	**Description**
`Ok`	Flash wait state was set successfully
`ErrorInvalidParameter`	Wrong enumerator used

5.17.6 Flashtiming_SetWorkFlashWait ()

This function allows to insert wait states for a Work-Flash area.

Note, this function is only available for Device Types 4 and 5.

Prototype	
`en_result_t Flashtiming_SetWorkFlashWait ( en_flashtiming_work_flash_wait_mode_t enFlashMode )`	
Parameter Name	**Description**
`[in] enFlashMode`	Wait State Mode. Possible enumerators are: • `WorkFlash0CycleMode` (0 WS, Type 4, 5) • `WorkFlash2CycleMode` (2 WS, Type 4, 5) • `WorkFlash4CycleMode` (4 WS,Type 5 only)
Return Values	**Description**
`Ok`	Work-Flash wait state was set successfully
`ErrorInvalidParameter`	Wrong enumerator used

5.17.7 Example Code

```
#include "flashtiming.h"

function1  // Device Types 2, 4
{
  en_flashtiming_flash_buffer_status_t enFlashStatus;

  . . .

  enFlashStatus = Flashtiming_GetFlashBufferStatus();

  . . .

  Flashtiming_EnableFlashBuffer();

  . . .
}

function2  // Device Types 0, 1, 2, 3, 4, 5, 7
{
  . . .

  Flashtiming_SetFlashSyncDown(FlashWait3);

  . . .
}

function3  // Device Types 0, 2, 4, 6, 8, 9
{
  . . .

  Flashtiming_SetFlashWait(Flash0CycleMode);

  . . .

}

function4  // Device Types 4, 5
{
  . . .

  Flashtiming_SetWorkFlashWait(WorkFlash0CycleMode);

  . . .

}
```

5.18 (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_mb9[ab]f[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 block 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:
```
bL3_GPIO_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.

5.18.1 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_N*pxy*	inverted logic
settings	Settings	
	Gpio1pin_InitPullup()	Pull-up: 0 = OFF, 1 = ON

5.18.2 Gpio1pin_InitOut()

This macro sets a Port to digital output.

Macro		
Gpio1pin_InitOut(p, settings)		
Parameter Name	**Description**	
p	Port Pin Name	
	GPIO1PIN_P*xy*	normal logic
	GPIO1PIN_NP*xy*	inverted logic
settings	Settings	
	Gpio1pin_InitVal()	Initial value 0 or 1

5.18.3 Gpio1pin_Get()

This macro reads out a GPIO pin via PDIR.

Macro		
Gpio1pin_Get(p)		
Parameter Name	**Description**	
p	Port Pin Name	
	GPIO1PIN_P*xy*	normal logic
	GPIO1PIN_NP*xy*	inverted logic
Return Value		
	Value	
	0 or 1	

5.18.4 Gpio1pin_Put()

This macro set a GPIO port via PDOR.

Macro		
Gpio1pin_Put(p, v)		
Parameter Name	**Description**	
p	Port Pin Name	
	GPIO1PIN_P*xy*	normal logic
	GPIO1PIN_NP*xy*	inverted logic
v	Value	
	0 or 1	

5.18.5 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.

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

Assume the waveform generator 01 output relocation 1 is located on port P24 and shared its pin with the analog input AN1D. The macro

```
SetPinFunc_RTO01_1();
```

sets the `EPRF01.RTO01E` to `RTO01_1` usage, sets `PFR2.P4` to `1`, and switches off `ADE.AN1D`.

5.18.6 Example Code

The following code excerpt shows how to use the GPIO macros. It sets Port10 to output and Port2F to input with enabled internal pull-up resistor. Additionally the MFS0 pins at relocation 2 are enabled.

```
#include "gpio1pin.h"

#define ON  1
#define OFF 0

function
{
  // Init P10 to Output
  Gpio1pin_InitOut( GPIO1PIN_P10, Gpio1pin_InitVal( OFF ) );

  // Init P2F to pulled-up input
  Gpio1pin_InitIn ( GPIO1PIN_P2F, Gpio1pin_InitPullup( ON ) );

  . . .

  // Put logical '1' to P10
  Gpio1pin_Put( GPIO1PIN_P10, ON );

  // Check pin state of P2F
  if (0 == Gpio1pin_Get( GPIO1PIN_P2F ))
  {
    . . .
  }

  // Activate MFS0 TX and RX pins at relocation '2'
  SetPinFunc_SIN0_2();
  SetPinFunc_SOT0_2();

  . . .

}
```

5.19 (LVD) Low Voltage Detection

Type Definition	–
Configuration Type	`stc_lvd_config_t`
Address Operator	–

`Lvd_InitIrq()` initializes the LVD interrupt with the given voltage threshold and `Lvd_DeInitIrq()` disables the LVD interrupt. In case of an interrupt a callback function is called, if its pointer != `NULL`.

5.19.1 Configuration Structure

The LVD module uses the following configuration structure of the type `stc_lvd_config_t`:

Type	Field	Possible Values	Description
`en_lvd_irq_voltage_t`	`enIrqVoltage`	*see below*	Threshold Voltage for Interrupt
`en_lvd_res_voltage_t`	`enResVoltage*`	*see below*	Threshold Voltage for Reset*
`boolean_t`	`bEnableLvdReset*`	`TRUE` `FALSE`	LVD Reset enabled* LVD Reset disabled*
`func_ptr_t`	`pfnCallback`	–	LVD Interrupt callback function pointer

* Only available for Device Type 3, 6, 7

5.19.1.1 Interrupt Voltage Enumerators

The following enumerators are used for `en_lvd_irq_voltage_t enIrqVoltage`. There are existing 3 configuration types.

- (A) Device Type 0, 1, 2, 4, 5
- (B) Device Type 3, 7
- I Device Type 6

Enumerator	Description	Configuration Type
`LvdIrqVoltage170`	Interrupt voltage in the vivinity of 1.70 Volts	(C)
`LvdIrqVoltage175`	Interrupt voltage in the vivinity of 1.75 Volts	(C)
`LvdIrqVoltage180`	Interrupt voltage in the vivinity of 1.80 Volts	(C)
`LvdIrqVoltage185`	Interrupt voltage in the vivinity of 1.85 Volts	(C)
`LvdIrqVoltage190`	Interrupt voltage in the vivinity of 1.90 Volts	(C)
`LvdIrqVoltage195`	Interrupt voltage in the vivinity of 1.95 Volts	(C)
`LvdIrqVoltage200`	Interrupt voltage in the vivinity of 2.00 Volts	(C)
`LvdIrqVoltage205`	Interrupt voltage in the vivinity of 2.05 Volts	(C)
`LvdIrqVoltage250`	Interrupt voltage in the vivinity of 2.50 Volts	(C)
`LvdIrqVoltage260`	Interrupt voltage in the vivinity of 2.60 Volts	(C)
`LvdIrqVoltage270`	Interrupt voltage in the vivinity of 2.70 Volts	(C)
`LvdIrqVoltage280`	Interrupt voltage in the vivinity of 2.80 Volts	(C)
`LvdIrqVoltage290`	Interrupt voltage in the vivinity of 2.90 Volts	(C)
`LvdIrqVoltage300`	Interrupt voltage in the vivinity of 3.00 Volts	(C)
`LvdIrqVoltage310`	Interrupt voltage in the vivinity of 3.10 Volts	(C)
`LvdIrqVoltage320`	Interrupt voltage in the vivinity of 3.20 Volts	(C)
`LvdIrqVoltage20`	Interrupt voltage in the vivinity of 2.0 Volts	(B)
`LvdIrqVoltage21`	Interrupt voltage in the vivinity of 2.1 Volts	(B)

`LvdIrqVoltage22`	Interrupt voltage in the vivinity of 2.2 Volts	(B)
`LvdIrqVoltage23`	Interrupt voltage in the vivinity of 2.3 Volts	(B)
`LvdIrqVoltage24`	Interrupt voltage in the vivinity of 2.4 Volts	(B)
`LvdIrqVoltage25`	Interrupt voltage in the vivinity of 2.5 Volts	(B)
`LvdIrqVoltage26`	Interrupt voltage in the vivinity of 2.6 Volts	(B)
`LvdIrqVoltage28`	Interrupt voltage in the vivinity of 2.8 Volts	(A)+(B)
`LvdIrqVoltage30`	Interrupt voltage in the vivinity of 3.0 Volts	(A)+(B)
`LvdIrqVoltage32`	Interrupt voltage in the vivinity of 3.2 Volts	(A)+(B)
`LvdIrqVoltage36`	Interrupt voltage in the vivinity of 3.6 Volts	(A)+(B)
`LvdIrqVoltage37`	Interrupt voltage in the vivinity of 3.7 Volts	(A)+(B)
`LvdIrqVoltage40`	Interrupt voltage in the vivinity of 4.0 Volts	(A)+(B)
`LvdIrqVoltage41`	Interrupt voltage in the vivinity of 4.1 Volts	(A)+(B)
`LvdIrqVoltage42`	Interrupt voltage in the vivinity of 4.2 Volts	(A)+(B)

5.19.1.2 Reset Voltage Enumerators

The following enumerators are used for `en_lvd_res_voltage_t enResVoltage`. There are existing 2 configuration types.

- (B) Device Type 3, 7
- I Device Type 6

Enumerator	**Description**	**Configuration Type**
`LvdResVoltage150`	Reset voltage in the vivinity of 1.50 Volts	(C)
`LvdResVoltage155`	Reset voltage in the vivinity of 1.55 Volts	(C)
`LvdResVoltage160`	Reset voltage in the vivinity of 1.60 Volts	(C)
`LvdResVoltage165`	Reset voltage in the vivinity of 1.65 Volts	(C)
`LvdResVoltage170`	Reset voltage in the vivinity of 1.70 Volts	(C)
`LvdResVoltage175`	Reset voltage in the vivinity of 1.75 Volts	(C)
`LvdResVoltage180`	Reset voltage in the vivinity of 1.80 Volts	(C)
`LvdResVoltage185`	Reset voltage in the vivinity of 1.85 Volts	(C)
`LvdResVoltage190`	Reset voltage in the vivinity of 1.90 Volts	(C)
`LvdResVoltage195`	Reset voltage in the vivinity of 1.95 Volts	(C)
`LvdResVoltage200`	Reset voltage in the vivinity of 2.00 Volts	(C)
`LvdResVoltage205`	Reset voltage in the vivinity of 2.05 Volts	(C)
`LvdResVoltage250`	Reset voltage in the vivinity of 2.50 Volts	(C)
`LvdResVoltage260`	Reset voltage in the vivinity of 2.60 Volts	(C)
`LvdResVoltage270`	Reset voltage in the vivinity of 2.70 Volts	(C)
`LvdResVoltage280`	Reset voltage in the vivinity of 2.80 Volts	(C)
`LvdResVoltage290`	Reset voltage in the vivinity of 2.90 Volts	(C)
`LvdResVoltage300`	Reset voltage in the vivinity of 3.00 Volts	(C)
`LvdResVoltage310`	Reset voltage in the vivinity of 3.10 Volts	(C)
`LvdResVoltage320`	Reset voltage in the vivinity of 3.20 Volts	(C)
`LvdResVoltage153`	Reset voltage in the vivinity of 1.53 Volts	(B)
`LvdResVoltage193`	Reset voltage in the vivinity of 1.93 Volts	(B)

5.19.2 Lvd_InitIrq()

Initialization of the LVD Interrupt at user defined voltage threshold. If LVD reset voltage is supported by the device, this threshold voltage is also set by this function.

Prototype	
`en_result_t Lvd_InitIrq( stc_lvd_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to LVD configuration structure
Return Values	**Description**
`Ok`	LVD Interrupt initialization successfully done
`ErrorInvalidParameter`	• `pstcConfig == NULL` • Wrong or not supported Interrupt voltage • Wrong of not supported Reset voltage

5.19.3 Lvd_DeInitIrq()

Disables the LVD interrupt. Note that the user's reset voltage (if supported) is retained.

Prototype	
`en_result_t Lvd_DeInitIrq( void )`	
Return Values	**Description**
`Ok`	LVD Interrupt de-initialization successfully done

5.19.4 LvdCallback()

This function is called, if `stc_lvd_config_t::pfnCallbackis` is defined.

Callback Function
`void LvdCallback( void )`

5.19.5 Example Code

The following code excerpt shows how to use the LVD functions.

```
#include "lvd.h"

void LvdCallback(void)
{
  // Hit, if voltage is below 4.0 Volts
}

function
{
  stc_lvd_config_t stcLdvConfig;

  L3_ZERO_STRUCT(stcLdvConfig);

  stcLdvConfig.enIrqVoltage = LvdIrqVoltage40;  // Threshold 4.0 Volts
  stcLdvConfig.pfnCallback  = &LvdCallback;

  Lvd_InitIrq(&stcLdvConfig);
}
```

5.20 (MFS) Multi Function Serial Interface

Type Definition	`stc_mfsn_t`
Configuration Type	`stc_mfs_config_t`
Address Operator	`MFS`*n*

`Mfs_Init()` must be used for activation and setup of an MFS channel. All MFS connection related parameters can be set with parameter `pstcConfig` of type `stc_mfs_config_t`.

The following restrictions have to be considered:

- There must be a start bit in MFS asynchronous mode.
- The clock source must be `MfsUseExternalSerialClockSourceAsItIs` in MFS synchronous slave mode.
- Continuous clock output must only be active in MFS synchronous master mode.

On initialization, the API has to provide a memory location for receive and transmit buffer handling (also for DMA usage). The memory location will be set on initialization with `stc_mfs_config_t::pcTxBuffer`, `stc_mfs_config_t::u16TxBufferSize`, `stc_mfs_config_t::pcRxBuffer` and `stc_mfs_config_t::u16RxBufferSize`.

The MFS initialization will create a receive and transmit ring buffer on this memory location and handle all ring buffer related parameters (`stc_mfs_buffer_t`). So each MFS instance will consist of a 2 step buffer handling by combination of the ring buffer with the hardware internal fifo buffer. The length of the receive and transmit hardware FIFO is `MFS_FIFO_MAX_VAL` (usually 16).

The receive and transmit interrupt level of each hardware FIFO is set internally by the MFS implementation and will be changed according to the transmit and receive data length while any transmission or reception of data is ongoing. Beside the `Mfs_Read()` and `Mfs_Write()` function it is possible to simulate a SPI transfer by using the function `Mfs_SynchronousTransfer()`. The MFS has to be initialized in synchronous mode for using `Mfs_SynchronousTransfer()`. To change the mode of MFS (synchronous to asynchronous and vice versa) or any other MFS connection related settings it will be necessary to call first `Mfs_DeInit()` to reset all MFS related register and to call `Mfs_Init()` afterwards with new mode and settings.

Any ongoing or pending transmission or reception will be cancelled and any data in the hardware FIFOs and ring buffer will be lost. The API can call `Mfs_Purge()` to interrupt an ongoing transmission or reception without changing the MFS mode and settings. All transmission and reception related buffer will be reset and all data in the buffer will be lost.

If the MFS module is used in synchronous receive and/or transmit mode (`stc_mfs_config_t::pfnTxCallback` and/or `stc_mfs_config_t::pfnRxCallback` are `NULL`) then the corresponding read and write functions (`Mfs_Read()`, `Mfs_Write()` and `Mfs_SynchronousTransfer()`) have to be called with parameter `bWait` set to `TRUE`. Note that this meaning of synchronous or asynchronous API handling is not the same like MFS synchronous or asynchronous data transfer. The MFS module uses the receive and transmit buffer provided by the API to create ring buffers. So there are 2 receive and transmit steps for sending and receiving data: the ring software buffer and the hardware FIFO. All necessary parameter to configure a MFS channel can be set with the `Mfs_Init::pstcConfig` parameter, all other interfaces (`Mfs_Read()`, `Mfs_Write()`, `Mfs_Purge()` and `Mfs_SynchronousTransfer()`) will only operate with data to be sent or received over the configured MFS channel.

The API can handle the receive and transmit communication by using the `Mfs_Read()`, `Mfs_Write()` and `Mfs_SynchronousTransfer()` either in synchronous or asynchronous way. There is no need for setting the callback function `stc_mfs_config_t::pfnTxCallback` when using the `Mfs_Write()` function in synchronous handling mode by setting the parameter `bWait` to `TRUE`. Note that in synchronous API mode the function will block until all data is put into the transmit hardware FIFO.

The `stc_mfs_config_t::pfnTxCallback` must be set to a valid value if the API will use the `Mfs_Write()` function in an asynchronous way by setting the parameter `bWait` to `FALSE`.

The same for `Mfs_Read()` function and `Mfs_SynchronousTransfer()` function (since `Mfs_SynchronousTransfer()` uses receive interrupt to signal transmission complete). The API can use `Mfs_Read()` and `Mfs_SynchronousTransfer()` in an asynchronous way by setting the `bWait` parameter to `FALSE`. In this case the `stc_mfs_config_t::pfnRxCallback` must be set to a valid value on MFS initialization.

The function `Mfs_Read()` will block until the amount of received data matches the Mfs_Read#u16ReadCount value if used in synchronous way. The function `Mfs_SynchronousTransfer()` will only block if it is used in asynchronous way and data should be received. The `Mfs_SynchronousTransfer()` function will neither block in asynchronous nor in synchronous mode if it is used only for sending data (`Mfs_Transfer::pu8RxData` set to `NULL`) due to the `Mfs_Transfer::u8TransferSize` restriction (`Mfs_Transfer::u8TransferSize` maximum length is the hardware FIFO length).

For each transmission function (`Mfs_Read()`, `Mfs_Write()`) is only one asynchronous operation possible at a time.

The I^{2}C functionality is done w/o interrupts or FIFO usage. It uses only the following parameter of the MFS configuration: `stc_mfs_config_t::enMode` and `stc_mfs_config_t::u32DataRate`. After initialization of the MFS to I^{2}C master mode, a start condition has to be sent via `Mfs_I2C_Start()`. Then depending on the protocol bytes can be sent via `Mfs_I2C_WriteSync()`. If a continued (or repeated) start condition has to be sent, `Mfs_I2C_Continue()` has to be used. Any communication has to be finished by the Stop Condition, which is sent out by `Mfs_I2C_Stop()` with the parameter `en_mfs_i2c_datadirection_t::I2Cwrite` or `en_mfs_i2c_datadirection_t::I2Cread`.

`Mfs_I2C_SetNoiseFilter()` sets the MFS's noise filter if supported by the device type.

For using MFS's LIN Mode use only function with 'Mfs_Lin_' prefix in the name (Except for `Mfs_Read()` and `Mfs_Write()`) Also `stc_mfs_lin_config_t` has to be used as the MFS configuration instead of `stc_mfs_config_t`.

`Mfs_Lin_Init()` is used to initialize an MFS instance to LIN mode with its dedicated LIN configuration (`stc_mfs_lin_config_t`).

`Mfs_Lin_SetBreak()` set the LIN Break in LIN Master mode. The baud rate divisor (not the rate itself!) can be adjusted by `Mfs_Lin_SetNewBaudDivisor()` after measurement with a dedicated ICU in LIN Slave mode.

Note that the LIN functionality only works properly when the MFS is connected to a LIN transceiver, which means, that the SOT line as always read-back by SIN input!

`Mfs_Lin_DisableRxInterrupt()` is used to disable 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_Lin_TransferRxBuffer()` transfers the reception data from the internal ring buffer to a user buffer. This function can be used for LIN Master and Slave mode, because of the external LIN transceiver, every data (transmission and/or reception) is always read completely to the reception buffer.

5.20.1 Transmission Ring Buffer Principle

The following illustration shows, how the transmission ring buffer works.

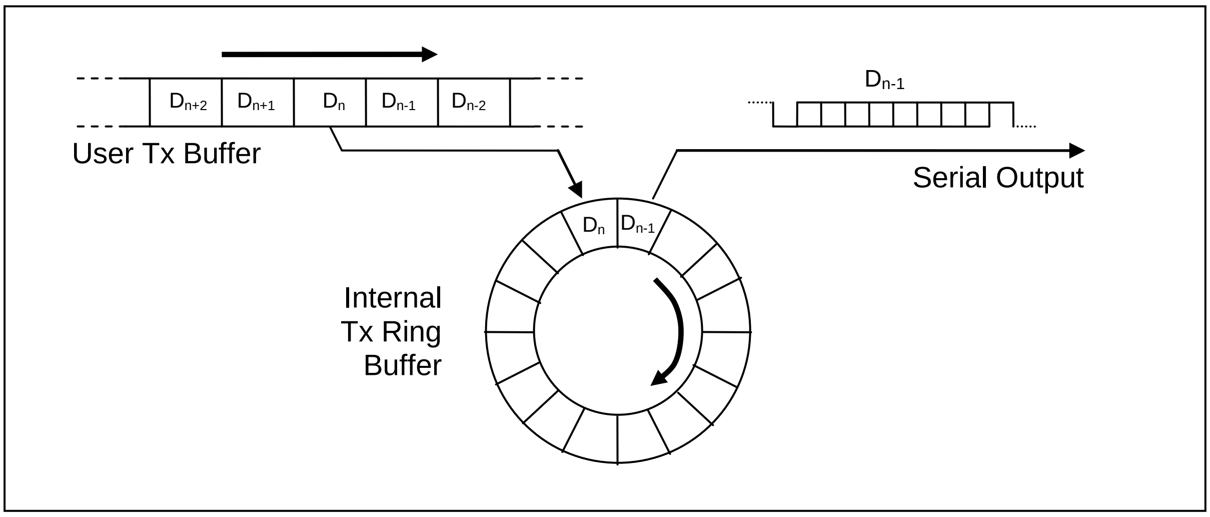


Figure 5-2: Transmission Ring Buffer Principle

The transmission data is transferred from the user transmission buffer to the internal ring buffer. This internal buffer is used then, to generate the serial output afterwards. Note that the hardware Tx FIFO between ring buffer and serial output is not drawn here.

5.20.2 Reception Ring Buffer Principle

The following illustration shows, how the reception ring buffer works.

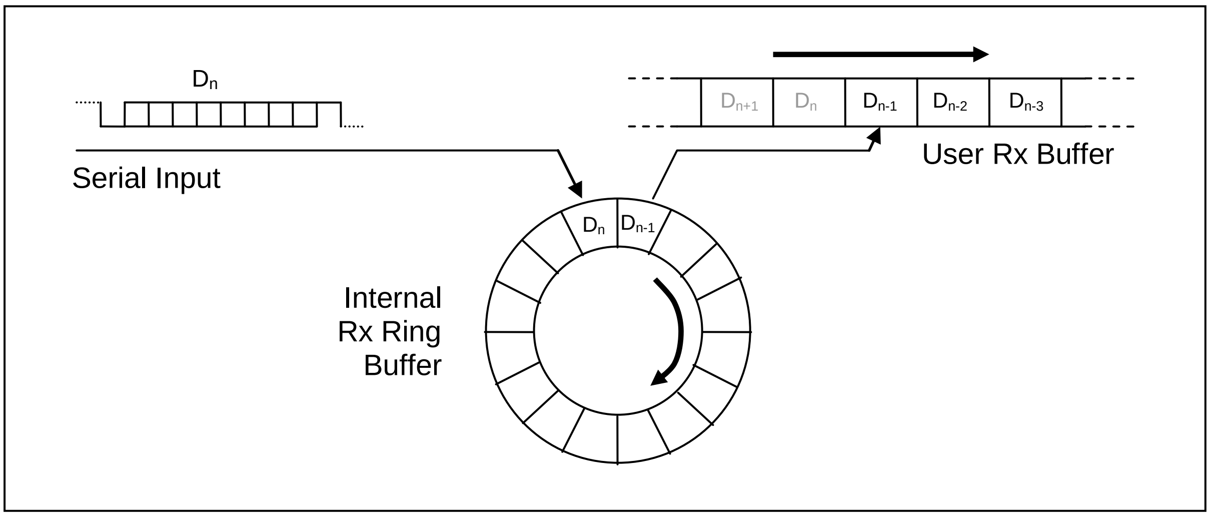


Figure 5-3: Reception Ring Buffer Principle

The reception data first is stored in the internal ring buffer. From this ring buffer the data is transferred to the user reception buffer. Note that the hardware Rx FIFO acting as intermediate buffer after reception and ring buffer is not drawn here.

5.20.3 Configuration Structures

The MFS module uses the following configuration structures.

5.20.3.1 Asynchronous, synchronous, and I^{2}C Mode

The configuration structure of the type of `stc_mfs_t` for all modes other than Lin mode is:

Type	Field	Possible Values	Description
en_mfs_mode_t	enMode	MfsAsynchronous0 MfsAsynchronousMulti1 MfsSynchronousMaster2 MfsSynchronousSlave2 MfsI2cMaster MfsI2cSlave	UART mode 0 UART mode 1 (Master/Slave) CSIO mode 2 (Master) CSIO mode 2 (Slave) I2C mode 3 (Master) I2C mode 3 (Slave)
en_mfs_… clk_src_t	enClock… Source	MfsUseInternalBaudRate… Generator MfsUseExternalSerial… ClockSourceWithBaudRateGenerator MfsUseExternalSerial… ClockSourceAsItIs	Clock for asynchronous operation Clock for synchronous operation Clock for synchronous slave operation
uint32_t	u32DataRate	–	Bits per second
en_mfs_… parity_t	enParity	MfsParityNone MfsParityEven MfsParityOdd	No parity used in mode 0 Even parity in mode 0 Odd parity in mode 0
en_mfs_… start_… stopbit_t	enStart… StopBit	MfsNoStartNoStop MfsOneStartOneStop MfsOneStartTwoStop MfsOneStartThreeStop MfsOneStartFourStop	No start and no stop bit (mode 2) 1 start and 1 stop bit (mode 0, 1) 1 start and 2 stop bits (mode 0, 1) 1 start and 3 stop bits (mode 0, 1) 1 start and 4 stop bits (mode 0, 1)
en_mfs_… character… length_t	u8Char… Length	MfsFifeBits MfsSixBits MfsSevenBits MfsEightBits MfsNineBits	5 Data bits 6 Data bits 7 Data bits 8 Data bits 9 Data bits
oolean_t	bBit… Direction	TRUE FALSE	LSB first MSB first
oolean_t	bSync… Clock… Inversion	TRUE FALSE	Sampling on falling edge (mode 2) Sampling on rising edge (mode 2)
en_mfs_… sync_wait_… time_t	u8Sync… Clock… WaitTime	MfsSyncWaitZero MfsSyncWaitOne MfsSyncWaitTwo MfsSyncWaitThree	0 wait cycle insertion (mode 2) 1 wait cycle insertion (mode 2) 2 wait cycle insertion (mode 2) 3 wait cycle insertion (mode 2)
oolean_t	bSignal… System	TRUE FALSE	NRZI NRZ
oolean_t	bSync… Clock… Output… Delay	TRUE FALSE	Clock delay half cycle (SPI mode 2) No Clock delay (mode2)
char_t*	pcTxBuffer	-	Pointer to user transmission buffer
uint16_t	u16Tx… BufferSize	-	User transmission buffer size

char_t*	pcRxBuffer	-	Pointer to user reception buffer
uint16_t	u16Rx… BufferSize	-	User reception buffer size
uint16_t	u16Rx… Callback… Buffer… FillLevel	0 ≤ x ≤ u16RxBufferSize	Number of Data into Buffer until RX callback function will be called
en_mfs_tx_… callback_… mode_t	enTx… Callback… Mode	MfsOnTxBufferEmpty MfsOnTxFinished	Callb., if last data was sent to shifter Callback, if last buffer data was sent out
mfs_tx_cb_… func_ptr_t	pfnTx… Callback	-	Pointer to transmission callback function.
Mfs_rx_cb_… func_ptr_t	pfnRx… Callback	-	Pointer to reception callback function.
Mfs_err_… cb_func_… ptr_t	pfnErr… Callback	-	Pointer to error reception callback function.
oolean_t	bSubstLF… withCRLF	TRUE FALSE	LF is sent with CR+LF (MS DOS format) LF is sent as it is

5.20.3.2 Lin Mode

The configuration structure of the type of `stc_mfs_lin_t` for Lin mode is:

Type	Field	Possible Values	Description
en_lin_mode_t	enLinMode	LinMaster LinSlave	Lin Master Mode Lin Slave Mode
oolean_t	bExtWakeUp	TRUE FALSE	External Wake-up function No external Wake-up
uint32_t	u32DataRate	–	Bits per Second
en_lin_… stop_bit_… length_t	enLin… StopBits	LinOneStopBit LinTwoStopBits LinThreeStopBits LinFourStopBits	1 Stop Bit used 2 Stop Bits used 3 Stop Bits used 4 Stop Bits used
en_lin_… break_… length_t	enLin… Break… Length	LinBreakLength13 LinBreakLength14 LinBreakLength15 LinBreakLength16	Lin Break 13 bit times (Lin Master) Lin Break 14 bit times (Lin Master) Lin Break 15 bit times (Lin Master) Lin Break 16 bit times (Lin Master)
en_lin_… delimiter_… length_t	enLin… Delimiter… Length	LinDelimiterLength1 LinDelimiterLength2 LinDelimiterLength3 LinDelimiterLength4	Lin Break to Filed delimiter: 1 bit time Lin Break to Filed delimiter: 2 bit times Lin Break to Filed delimiter: 3 bit times Lin Break to Filed delimiter: 4 bit times
oolean_t	bLin… BreakIrq… Enable	TRUE FALSE	Enable Break interrupt (callback) Disable Break interrupt (callback)
char_t*	pcLinTx… Buffer	-	Pointer to transmission buffer
uint16_t	u16LinTx… Buffer… Size	-	Length of transmission buffer
char_t*	pcLinRx… Buffer	-	Pointer to reception buffer
uint16_t	u16LinRx…	-	Length of reception buffer

	Buffer… Size		
mfs_rx_cb_… func_ptr_t	pfnLin… Break… Callback	–	Pointer to Lin Break callback function
mfs_rx_cb_… func_ptr_t	pfnLinRx… Callback	–	Pointer to reception callback function
mfs_rx_cb_… func_ptr_t	pfnLinTx… Callback	–	Pointer to transmission callback function

5.20.3.3 I^{2}C Noise Filter Configuration

Because each MFS has an own noise filter (if supported by device type), these noise filters have an own configuration structure of the type of `stc_mfs_dnf_t`:

Type	Field	Possible Values	Description
en_mfs_dnf_t	enDnf0	DnfNoFilter Dnf1StepFilter Dnf2StepFilter Dnf3StepFilter	No noise filter, APB1 clock: 40 ≤ MHz 1-step noise filter, APB1 clock: 60 ≤ MHz 2-step noise filter, APB1 clock: 80 ≤ MHz 3-step noise filter, APB1 clock: 100 ≤ MHz

5.20.4 Mfs_Init()

This Function oolean e s the MFS according the Options setup in the passed configuration oolean e. Several checks are done before that and an error is returned if invalid modes are requested.

These rules are checked:

- A start bit must be used in asynchronous mode.
- External clock source as it is must be used in synchronous slave mode.
- Pointer to receive and transmit buffer must be valid.

The required timing settings for data rate are calculated automatically with respect to the current peripheral clock #0.

All necessary parameter to configure a MFS channel can be set with the `Mfs_Init::pstcConfig` parameter, all other interfaces (`Mfs_Read()`, `Mfs_Write()`, `Mfs_Purge()` and `Mfs_SynchronousTransfer()`) will only operate with data to be sent or received over the configured MFS channel.

Preconditions:

- Peripheral Clock 0 must be configured before to correctly set data rate.
- For MFS asynchronous mode the peripheral clock must be at least 5 times faster than the data rate to ensure proper oversampling.
- The Application must configure corresponding MFS pins (`SIN`, `SOT`, `SCK`) according to requirements and settings of MFS instance!

.

Prototype	
`en_result_t Mfs_Init( stc_mfsn_t* pstcMfs, const stc_mfs_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
`[in] pstcConfig`	Pointer to MFS configuration structure
Return Values	**Description**
`Ok`	MFS initialization successfully done
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pstcConfig == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved) • `pstcConfig->pcTxBuffer == NULL` • `pstcConfig->pcRxBuffer == NULL` • One or more configuration data wrong • Wrong enumerator used or enumerator not supported

5.20.5 Mfs_DeInit()

De-Initializes the MFS. Use this function for mode change or other changed settings.

Prototype	
`en_result_t Mfs_DeInit( stc_mfsn_t* pstcMfs )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
Return Values	**Description**
`Ok`	MFS de-initialization successfully done
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved)

5.20.6 Mfs_Read()

The received data is copied from internal RX buffer (filled by RX interrupt) into the provided data buffer `Mfs_Read::pcData`. The size is defined by Mfs_Read#pu16ReadCount. Depending on the `Mfs_Read::bWait` parameter, the function behavior is different.

For an asynchronous (non-blocking) call (`Mfs_Read::bWait = FALSE`), the function will return immediately after all currently available characters (in SW ring buffer and HW FIFO) are copied into the provided buffer (`Mfs_Read::pcData`) or the maximum count (`Mfs_Read::pu16ReadCount`) is reached. The value returned by `Mfs_Read::puReadCount` gives the count of characters that was read actually. If the referenced MFS does not have a FIFO single data is read.

For a synchronous (blocking) call (`Mfs_Read::bWait == TRUE`), the function will return after the requested count of characters (`Mfs_Read::pu16ReadCount`) is received completely. This should be used with caution as the full application can get stuck if no further data is received.

The reception interrupt is used in any case.

Prototype	
`en_result_t Mfs_Read( stc_mfsn_t* pstcMfs, char_t* pcData, uint16_t* pu16DataCount, uint16_t u16ReadCount, oolean_t bWait )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
`[in][out] pcData`	Pointer to reception character (data) buffer
`[out] pu16DataCount`	Pointer to variable of characters (data) being actually read
`[in] u16ReadCount`	Maximum number of characters (data) to read (buffer size must be sufficient!)
`[in] bWait`	`TRUE`: Synchronously wait until u16ReadCount bytes have been received. `FLASE`: Read all available data and return immediately.
Return Values	**Description**
`Ok`	Read data successfully done or started.
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved)
`ErrorOperationInProgress`	An asynchronous transmission is still ongoing while another asynchronous operation should be started.

5.20.7 Mfs_Write()

The data provided by `Mfs_Write::pcData` is copied into the internal TX buffer and the transmission (via TX interrupt) is started, if not ongoing already. Depending on the `Mfs_Write::bWait` parameter, the function return behavior is different.

For an asynchronous (non-blocking) call (`Mfs_Write::bWait = FALSE`), the free size of the internal buffer must be sufficient to take all data (Mfs_Write::pcData) of length `Mfs_Write::u16DataLength`, otherwise the function will return `ErrorBufferFull`. After all data is copied into the internal buffer, the function will return immediately. The transmission may be pending when the function returns.

For a synchronous (blocking) call (`Mfs_Write::bWait = TRUE`), the function will wait until all 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 single data is written.

Prototype

```
en_result_t Mfs_Write( stc_mfsn_t*   pstcMfs,
                       const_char_t* pcData,
                       uint16_t      u16WriteCount,
                        oolean_t     bWait )
```

Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
`[in] pcData`	Pointer to transmission character (data) buffer
`[in] u16WriteCount`	Number of characters (data) to be written. Must be > `0`.
`[in] bWait`	`TRUE`: Synchronously wait until transmission is finished. `FLASE`: Put all data to transmission internal ring buffer and return.
Return Values	**Description**
`Ok`	Transmission data successfully sent or ongoing.
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved) • `u16WriteCount == 0`
`ErrorBufferFull`	Insufficient free size of TX buffer to take all data (in case of `bWait = FALSE` only).

5.20.8 Mfs_SynchronousTransfer()

This function will transmit and receive the same amount of data, based on the serial (shift) clock signal (MFS `SCK` pin) in synchronous master or slave mode. The operation mode of the MFS instance must be setup for synchronous master or slave mode.

While the data in `Mfs_SynchronousTransfer::pu8TxData` is transmitted the input data is received and stored in `Mfs_SynchronousTransfer::pu8RxData`. The function operates in synchronous (blocking) mode only i.e. it will wait until the amount of data defined by `Mfs_SynchronousTransfer::u8TransferSize` is transmitted/received. No MFS internal buffers or transfer counters are used.

The Tx/Rx callback functions will not be called. Note, that in synchronous slave mode, the transfer is controlled by the external master device, providing the serial shift clock. This may cause this function to block very long time.

Because this function uses blocking method, no interrupts are needed and therefore are not used. Also no FIFO operation is performed.

Note:

Asynchronous (non-blocking) Tx/Rx operations are provided by using the functions `Mfs_Write()` and `Mfs_Read()` for MFS synchronous master and slave modes, too. Note, that these functions do not support full-duplex operation!

Prototype	
`en_result_t Mfs_SynchronousTransfer( stc_mfsn_t* pstcMfs, uint8_t* pu8TxData, uint8_t* pu8RxData, uint8_t u8TransferSize )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
`[in] pu8TxData`	Pointer to transmission data buffer
`[in] pu8RxData`	Pointer to reception data buffer
`[in] u8TransferSize`	Number of transfers. Must be > `0`.
Return Values	**Description**
`Ok`	Transmission data successfully sent or ongoing.
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved) • `pu8TxData == 0` • `u8TransferSize == 0`
`ErrorOperationInProgress`	A transmission is still pending or the Rx buffer is not empty.
`ErrorInvalidMode`	MFS module was not initialized in synchronous mode.

5.20.9 Mfs_PurgeRx()

The hardware reception FIFO and the software buffer are cleared. Any data which is in the receive buffer at this time will be lost!

All settings done by `Mfs_Init()` will be preserved.

DMA operation for read will be disabled and read access to receive buffer by CPU will be enabled.

Prototype	
`en_result_t Mfs_PurgeRx( stc_mfsn_t* pstcMfs )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
Return Values	**Description**
`Ok`	Purge successfully done.
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved)

5.20.10 Mfs_PurgeTx()

The hardware transmission FIFO and the software buffer are cleared. Any data which is not transferred at this time will be lost!

All settings done by `Mfs_Init()` will be preserved.

DMA operation for write will be disabled and write access to transmit buffer by CPU will be enabled..

Prototype	
`en_result_t Mfs_PurgeTx( stc_mfsn_t* pstcMfs )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
Return Values	**Description**
`Ok`	Purge successfully done.
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved)

5.20.11 Mfs_Purge()

The hardware transmission and reception FIFOs and the software buffers are cleared. Any data which is not transferred at this time will be lost!

All settings done by `Mfs_Init()` will be preserved.

DMA operation for write and read will be disabled and write access to transmit buffer by CPU will be enabled..

Prototype	
`en_result_t Mfs_Purge( stc_mfsn_t* pstcMfs )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
Return Values	**Description**
`Ok`	Purge successfully done.
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved)

5.20.12 Mfs_BuffersEmpty()

This Function checks all Buffers (software and hardware) for an pending Operation.

Prototype	
`en_result_t Mfs_BuffersEmpty( stc_mfsn_t* pstcMfs )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
Return Values	**Description**
`Ok`	Buffers are empty
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved)
`ErrorOperationInProgress`	Buffers are not empty

5.20.13 Mfs_ReadChar()

Reads a single character from MFS module synchronously or asynchronously.

The received data is returned top the caller, no error checking is done.

For an asynchronous (non-blocking) call (`Mfs_ReadChar::bWait = FALSE`), the function will return immediately after any available character (in the internal Rx ring buffer and hardware FIFO [if available]) is found.

For a synchronous (blocking) call (`Mfs_ReadChar::bWait == TRUE`), the function will return after receipt of a single character.

The reception interrupt is used in any case.

Note:

No error checking is done while reading from the MFS channel.

Prototype	

`char_t Mfs_ReadChar( stc_mfsn_t* pstcMfs,`
`                      boolean_t   bWait )`	

Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
`[in] bWait`	`TRUE`: Synchronously wait until reception complete. `FALSE`: Do not wait.
Return Values	**Description**
`char_t`	Read character
`'\0'`	No character received if `bWait == FALSE`

5.20.14 Mfs_WriteString()

Writes a '`\0`'-terminated string to MFS module synchronously or asynchronously.

The characters provided by `Mfs_WriteString::pcData` is copied into the internal Tx buffer and the transmission (via Tx interrupt) is started, if not ongoing already. Depending on the `Mfs_WriteString::bWait` parameter, the function return behavior is different.

For an asynchronous (non-blocking) call (`Mfs_WriteString::bWait = FALSE`), the free size of the internal buffer must be sufficient to take all data (`Mfs_Write::pcData`) of length `Mfs_Write::u16DataLength`, otherwise the function will return ErrorBufferFull.

After all data is copied into the internal buffer, the function will return immediately. The transmission may be pending when the function returns.

For a synchronous (blocking) call (`Mfs_WriteString::bWait = TRUE`), the function will wait until all 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, single data is transferred.

Note, no error checking is done while reading from the MFS.

Prototype	
`en_result_t Mfs_WriteString( stc_mfsn_t* pstcMfs, char_t* pszString, boolean_t bWait )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
`[in] pszString`	Pointer to string buffer. Must be '`\0`'-terminated!
`[in] bWait`	`TRUE`: Synchronously wait until reception complete. `FALSE`: Do not wait.
Return Values	**Description**
`Ok`	String successfully transmitted.
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pszString == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved)

5.20.15 Mfs_ReadString()

Reads an EOL-terminated string from the MFS module (a)synchronously

The character buffer provided by `Mfs_ReadString::pcData` is used to copy the received data.

The free size of the buffer must be sufficient to take all data (`Mfs_Read::pcData`) of length `Mfs_Read::u16MaxChars +1` (null-Terminator).

After `Mfs_Read::u16MaxChars` data is received, the function will return immediately without terminating the buffer.

If the provided terminator is detected the function will null-terminate the buffer and return.

Notes:

- No error checking is done while reading from the MFS channel.
- Given character array will be null terminated by the function itself.
- The reception FIFO is disabled because of random number of characters before receiving the terminator character.

Prototype	
`int16_t Mfs_ReadString( stc_mfsn_t* pstcMfs, char_t* pcData, uint16_t u16MaxChars, char_t cTerminator, boolean_t bEcho )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
`[in] pcData`	Pointer to string to be received
`[in] u16MaxChars`	Maximum number of characters to read in
`[in] cTerminator`	Terminator character (automatically set to '`\0`' in string destination).
`[in] bEcho`	`TRUE`: Echo the reception `FALSE`: Do not echo the reception.
Return Values	**Description**
`> 0`	Number of read characters
`= 0`	Parameter `u16MaxChars = 0` or no data except terminator received
`-1`	• `pstcMfs == NULL` • `pcData == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved)

5.20.16 Mfs_AsyncInit()

This function sets up a local configuration for asynchronous communication and fills the necessary contents from its arguments. Afterwards `Mfs_Init()` is called. Therefore the user does not need to setup a configuration.

Prototype	
`en_result_t Mfs_AsyncInit( stc_mfsn_t* pstcMfs, uint32_t u32DataRate, en_mfs_parity_t enParity, en_mfs_start_stopbit_t enStartStopBit, mfs_tx_cb_func_ptr_t pfnTxCallback, mfs_rx_cb_func_ptr_t pfnRxCallback )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
`[in] u32DataRate`	Bits per seconds
`[in] enParity`	Use parity or not
`[in] enStartStopBit`	Determines number of stop bits
`[in] pfnTxCallback`	Pointer to transmission callback routine
`[in] pfnRxCallback`	Pointer to reception callback routine
Return Values	**Description**
`Ok`	Asynchronous initialization successfully done
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pcData == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved)

5.20.17 Mfs_I2C_Start()

Generates an I^{2}C start condition.

Note:

This function must only be called, if the recent MFS instance was initialized to I^{2}C mode! Otherwise the behavior of this function is undefined!

Prototype	
`en_result_t Mfs_I2C_Start( volatile stc_mfsn_t* pstcI2c, uint8_t u8DestAddress, en_mfs_i2c_datadirection_t enAccess )`	
Parameter Name	**Description**
`[in] pstcI2c`	Pointer to MFS instance
`[in] u8DestAddress`	Address of external I^{2}C device (7 Bits, bit#0 aligned)
`[in] enAccess`	access type • `I2CRead` • `I2CWrite`
Return Values	**Description**
`Ok`	Start condition successfully sent.
`ErrorTimeout`	• Arbitration lost • No acknowledge • Bus error • Transmission finished timed out by `I2C_TIME_OUT_POLLING_TRIALS_INT` defined in *mfs.h* • Too much trials by `I2C_TIME_OUT_POLLING_TRIALS` defined in *mfs.h*
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pcData == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved)

5.20.18 Mfs_I2C_Continue()

Generates a continued (repeated) start condition (after a normal start condition is already created).

Note:

This function must only be called, if the recent MFS instance was initialized to I^{2}C mode! Otherwise the behavior of this function is undefined!

Prototype	
`en_result_t Mfs_I2C_Continue ( volatile stc_mfsn_t* pstcI2c, uint8_t u8DestAddress, en_mfs_i2c_datadirection_t enAccess )`	
Parameter Name	**Description**
`[in] pstcI2c`	Pointer to MFS instance
`[in] u8DestAddress`	Address of external I^{2}C device (7 Bits, bit#0 aligned)
`[in] enAccess`	access type • `I2CRead` • `I2CWrite`
Return Values	**Description**
`Ok`	Stop condition successfully sent.
`Error`	• Arbitration lost • No acknowledge • Bus error • Transmission finished timed out by `I2C_TIME_OUT_POLLING_TRIALS_INT` defined in *mfs.h*
`ErrorInvalidParameter`	• `pstcI2c == NULL` • `pcData == NULL` • `enAccess` has wrong enumerator • `pstcMfsInternData == NULL` (Instance could not be resolved)

5.20.19 Mfs_I2C_Stop()

Generates a stop condition (for write and read communication).

Note:

This function must only be called, if the recent MFS instance was initialized to I^{2}C mode! Otherwise the behavior of this function is undefined!

Prototype	
`en_result_t Mfs_I2C_Stop( volatile stc_mfsn_t* pstcI2c, en_mfs_i2c_datadirection_t enAccess )`	
Parameter Name	**Description**
`[in] pstcI2c`	Pointer to MFS instance
`[in] enAccess`	access type • `I2CRead` • `I2CWrite`
Return Values	**Description**
`Ok`	Stop condition successfully sent.
`ErrorInvalidParameter`	• `pstcI2c == NULL` • `enAccess` has wrong enumerator

5.20.20 Mfs_I2C_WriteSync()

This function writes n data to a selected device via I^{2}C.

Note:

This function must only be called, if the recent MFS instance was initialized to I^{2}C mode! Otherwise the behavior of this function is undefined!

Prototype	
`en_result_t Mfs_I2C_WriteSync( volatile stc_mfsn_t* pstcI2c, uint8_t* pu8Data, uint8_t u8DataLength )`	
Parameter Name	**Description**
`[in] pstcI2c`	Pointer to MFS instance
`[in] pu8Data`	Pointer to byte data to be sent
`[in] u8DataLength`	Number of bytes to be sent
Return Values	**Description**
`Ok`	All data successfully sent.
`Error`	• Arbitration lost • No acknowledge • Bus error
`ErrorTimeout`	Transmission finished timed out by `I2C_TIME_OUT_POLLING_TRIALS_INT` defined in *mfs.h*
`ErrorInvalidParameter`	`pstcI2c == NULL`

5.20.21 Mfs_I2C_ReadSync()

Read data to a selected I^{2}C device.

Note:

This function must only be called, if the recent MFS instance was initialized to I^{2}C mode! Otherwise the behavior of this function is undefined!

Prototype	
`en_result_t Mfs_I2C_WriteSync( volatile stc_mfsn_t* pstcI2c, uint8_t* pu8Data, uint8_t u8DataLength )`	
Parameter Name	**Description**
`[in] pstcI2c`	Pointer to MFS instance
`[in] pu8Data`	Pointer to byte data to be read
`[in] u8DataLength`	Number of bytes to be read
Return Values	**Description**
`Ok`	All data successfully read.
`Error`	• Arbitration lost • No acknowledge • Bus error
`ErrorTimeout`	Transmission finished timed out by `I2C_TIME_OUT_POLLING_TRIALS_INT` defined in *mfs.h*
`ErrorInvalidParameter`	`pstcI2c == NULL`

5.20.22 Mfs_I2C_SetNoiseFiler()

Set I^{2}C Auxiliary Noise Filters according configuration of type of `stc_mfs_dnf_t`.

Prototype	
`en_result_t Mfs_I2C_SetNoiseFilter( stc_mfs_dnf_t* pstcMfsDnf )`	
Parameter Name	**Description**
`[in] pstcMfsDnf`	Pointer to I^{2}C auxiliary noise filters
Return Values	**Description**
`Ok`	All noise filters successfully set.
`ErrorInvalidParameter`	• `pstcMfsDnf == NULL` • Wrong enumerator used

5.20.23 Mfs_Lin_Init()

For Lin master and slave functionality of an MFS instance `Mfs_Lin_Init()` must be used instead of `Mfs_Init()`. Also the Lin configuration has the own type `stc_mfs_lin_config_t`.

Prototype	

`en_result_t Mfs_Lin_Init( stc_mfsn_t* pstcMfs,`
`const stc_mfs_lin_config_t* pstcConfig )`	

Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
`[in] pstcConfig`	Pointer to MFS Lin configuration structure
Return Values	**Description**
`Ok`	MFS Lin initialization successfully done
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pstcConfig == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved) • One or more configuration data wrong • Wrong enumerator used or enumerator not supported

5.20.24 Mfs_Lin_SetBreak()

This Function sets a LIN break and break delimiter length with the configuration by the previous initialization.

Note:

This function must only be called, if the recent MFS instance was initialized to Lin master mode! Otherwise the behavior of this function is undefined!

Prototype	
`en_result_t Mfs_Lin_SetBreak( stc_mfsn_t* pstcMfs )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
Return Values	**Description**
`Ok`	MFS Lin Break set/being generated
`ErrorInvalidParameter`	`pstcMfs == NULL`
`ErrorInvalidMode`	MFS not in Lin master mode
`ErrorOperationInProgress`	MFS is not ready for generating a Lin break (`TBI != 0`)

5.20.25 Mfs_Lin_SetNewBaudDivisor()

This Function sets a new (calculated) baud divisor, if the MFS is in LIN Slave mode.

Note:

This function should only be called:

- MFS is initialized to Lin slave mode *and*
- After a complete LIN frame was received and before next LIN Break *and*
- Shortly after the second ICU interrupt within the LIN Synch Field and before the next start bit of the LIN Header byte!

Prototype	

`en_result_t Mfs_Lin_SetNewBaudDivisor( stc_mfsn_t* pstcMfs,`
`                                        uint32_t    u32BaudDivisor )`	

Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
`[in] u32BaudDivisor`	New (calculated) bit rate divisor
Return Values	**Description**
`Ok`	New bit rate was set successfully
`ErrorInvalidParameter`	`pstcMfs == NULL`
`ErrorInvalidMode`	MFS not in Lin slave mode

5.20.26 Mfs_Lin_TransferRxBuffer()

This function transfers n bytes of recent receive buffer to user buffer.

Note:

This function must only be called, if the recent MFS instance was initialized to Lin master or slave mode! Otherwise the behavior of this function is undefined!

Prototype	
`en_result_t Mfs_Lin_TransferRxBuffer( stc_mfsn_t* pstcMfs, char_t* pcData, uint16_t u16ReadCount )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance
`[in] pcData`	Pointer to user reception buffer
`[in] u16ReadCount`	Number of bytes to be transferred to reception buffer
Return Values	**Description**
`Ok`	Data was successfully transferred.
`ErrorInvalidParameter`	`pstcMfs == NULL`

5.20.27 Mfs_Lin_DisableRxInterrupt()

This function disables the reception interrupt.

Note:

This function must only be called, if the recent MFS instance was initialized to Lin master or slave mode! Otherwise the behavior of this function is undefined!

Prototype	
`void Mfs_Lin_DisableRxInterrupt( stc_mfsn_t* pstcMfs )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance (not checked in this function!).
Return Values	**Description**
`void`	-.

5.20.28 Mfs_Lin_DeInit()

This function just uses `Mfs_DeInit()` after checking LIN mode.

Prototype	
`en_result_t Mfs_Lin_DeInit( stc_mfsn_t* pstcMfs )`	
Parameter Name	**Description**
`[in] pstcMfs`	Pointer to MFS instance.
Return Values	**Description**
`Ok`	MFS de-initialized
`ErrorInvalidMode`	MFS not in Lin master or slave mode
`ErrorInvalidParameter`	• `pstcMfs == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved)

5.20.29 TxCallback()

This function is called, if `stc_mfs_config_t::pfnTxCallback` is defined.

Callback Function
`void TxCallback( void )`

5.20.30 RxCallback()

This function is called, if `stc_mfs_config_t::pfnTxCallback` is defined.

Callback Function
`void RxCallback( void )`

5.20.31 ErrorCallback()

This function is called, if `stc_mfs_config_t::pfnErrCallback` is defined.

Callback Function
`void ErrorCallback( void )`

Error enumerators will be provided in future versions of the LLL.

5.20.32 LinBreakCallback()

This function is called, if `stc_mfs_lin_config_t::pfnLinBreakCallback` is defined.

Callback Function
`void LinBreakCallback( void )`

5.20.33 LinTxCallback()

This function is called, if `stc_mfs_lin_config_t::pfnLinTxCallback` is defined.

Callback Function
`void LinTxCallback( void )`

5.20.34 LinRxCallback()

This function is called, if `stc_mfs_lin_config_t::pfnLinRxCallback` is defined.

Callback Function
`void LinRxCallback( void )`

5.20.35 Example Codes

The following code excerpt examples shows how to handle the MFS functions.

5.20.35.1 Asynchronous Communication via Configuration

This code shows some functions for asynchronous communication. It assumes a connected serial remote station, which can also receive and transmit.

```
#include "mfs.h"

#define BUFFERSIZE 256

char cRxBuffer[BUFFERSIZE];  // Reception Ring Buffer
char cTxBuffer[BUFFERSIZE];  // Transmission Ring Buffer

uint32_t u32CountTxCb = 0;
uint32_t u32CountRxCb = 0;

void Mfs0TxCallback(void)
{
  u32CountTxCb++;
}

void Mfs0RxCallback(void)
{
  u32CountRxCb++;
}

function
{
  stc_mfs_config_t stcConfig;        // MFS configuration
  stc_mfsn_t*      pstcMfs = NULL;   // MFS instance

  L3_ZERO_STRUCT(stcConfig);

  char           au8ReadBuffer[256] ;
  const uint8_t cszString[] = "Hello world!";
  uint16_t       u16ReadCount;

  stcConfig.enMode                          = MfsAsynchronous0;
  stcConfig.enClockSource                   = MfsUseInternalBaudRateGenerator;
  stcConfig.u32DataRate                     = 115200;
  stcConfig.enParity                        = MfsParityNone;
  stcConfig.enStartStopBit                  = MfsOneStartOneStop;
  stcConfig.u8CharLength                    = MfsEightBits;
  stcConfig.bBitDirection                   = 0;          // LSB first
  stcConfig.bSyncClockInversion             = FALSE;
  stcConfig.bSyncClockOutputDelay           = FALSE;
  stcConfig.pcTxBuffer                      = cTxBuffer;  // Tx Ring Buffer
  stcConfig.u16TxBufferSize                 = BUFFERSIZE;
  stcConfig.pcRxBuffer                      = cRxBuffer;  // Rx Ring Buffer
  stcConfig.u16RxBufferSize                 = BUFFERSIZE;
  stcConfig.enTxCallbackMode                = MfsOnTxBufferEmpty;
  stcConfig.pfnTxCallback                   = &Mfs0TxCallback;
  stcConfig.u16RxCallbackBufferFillLevel = 1;
  stcConfig.pfnRxCallback                   = &Mfs0RxCallback;

  if (Ok == Mfs_Init((stc_mfsn_t*)&MFS0, &stcConfig)
  {
    pstcMfs = (stc_mfsn_t*)&MFS0;  // Setup pointer to USART instance

    // Print string
    Mfs_WriteString(pstcMfs, (char*) cszString, 0);

    // Wait for transmission finished via TX call back function
    while(0 == Main_u32CountTxCb);

    // Read string with 256 chars max., <RETURN> as terminator, echo input
    Mfs_ReadString(pstcMfs, (char*) &au8ReadBuffer, 256, '\r', 1);

    // Print string
    Mfs_WriteString(pstcMfs, "\n\rReceived String: ", 0);
    u32CountTxCb = 0;
```

▼

```
                                                                         ▲

    Mfs_WriteString(pstcMfs, (char*) &au8ReadBuffer, 0);
    while(0 == u32CountTxCb);      // wait for finish ...

    // Mfs_Read() with dedicated number of characters (9)
    u32CountTxCb = 0;
    Mfs_WriteString(pstcMfs, "\n\rNow send 9 Characters (not echoed!).", 0);
    while(0 == u32CountTxCb);      // wait for finish ...

    u32CountRxCb = 0;
    u16ReadCount = 0;

    // Read 9 characters in buffer, blocking
    Mfs_Read(pstcMfs, (char*) &au8ReadBuffer, &u16ReadCount, 9, TRUE);

    // Print message
    Mfs_WriteString(pstcMfs, "\n\rSent String: ", 0);

    // Print via Mfs_Write() scanned string (9 characters)
    u32CountTxCb = 0;
    Mfs_Write(pstcMfs, (char*) &au8ReadBuffer, 9, FALSE);
    while(0 == u32CountTxCb);      // wait for finish ...

      . . .

    Mfs_DeInit(pstcMfs);
  }
}
```

5.20.35.2 Asynchronous Communication via Mfs_AsyncInit()

This example shows how to use the MFS initialization via `Mfs_AsyncInit()`. The same code from above (5.20.35.1) can be used for communication.

```
#include "mfs.h"

#define BUFFERSIZE 256

// No ring buffer definitions needed for Mfs_AsyncInit()

uint32_t u32CountTxCb = 0;
uint32_t u32CountRxCb = 0;

void Mfs0TxCallback(void)
{
  u32CountTxCb++;
}

void Mfs0RxCallback(void)
{
  u32CountRxCb++;
}

function
{
  // No user MFS configuration needed for Mfs_AsyncInit()
  stc_mfsn_t*      pstcMfs = NULL;  // MFS instance

  char           au8ReadBuffer[256] ;

                                                                          ▼
```

```
                                                                    ▲
  const uint8_t cszString[] = "Hello world!";
  uint16_t      u16ReadCount;

  if (Ok == Mfs_AsncInit((stc_mfsn_t*)&MFS0,   // usual instance pointer
                        115200,               // bits per second
                        MfsParityNone,        // no parity
                        MfsOneStartStop,      // 1 start bit, 1 stop bit
                        &Mfs0TxCallback,      // Tx Callback pointer
                        &Mfs0RxCallback)      // Rx Callback pointer
  {
    pstcMfs = (stc_mfsn_t*)&MFS0;  // Setup pointer to USART instance

    // *** Same code as above ***

    Mfs_DeInit(pstcMfs);
  }
}
```

5.20.35.3 Synchronous Communication

The following code shows an example of SPI master communication. Assume the SPI master send a 4-byte command and the next 2 frames are the 2-byte slave response.

```
#include "mfs.h"

#define BUFFERSIZE 256

char cRxBuffer[BUFFERSIZE];  // Reception Ring Buffer
char cTxBuffer[BUFFERSIZE];  // Transmission Ring Buffer

function
{
  stc_mfs_config_t stcConfig;        // MFS configuration
  stc_mfsn_t*      pstcMfs = NULL;   // MFS instance

  L3_ZERO_STRUCT(stcConfig);

  char au8RxBuffer[4];
  char au8TxBuffer[4];

  stcConfig.enMode                         = MfsSynchronousMaster2;
  stcConfig.enClockSource                  = MfsUseInternalBaudRateGenerator;
  stcConfig.u32DataRate                    = 1000000; // 1 MBit/s
  stcConfig.enParity                       = MfsParityNone;
  stcConfig.enStartStopBit                 = MfsNoStartNoStop;
  stcConfig.u8CharLength                   = MfsEightBits;
  stcConfig.bBitDirection                  = 0;       // LSB first
  stcConfig.bSyncClockInversion            = FALSE;
  stcConfig.bSyncClockOutputDelay          = TRUE;    // SPI
  stcConfig.pcTxBuffer                     = cTxBuffer;
  stcConfig.u16TxBufferSize                = BUFFERSIZE_SYNC;
  stcConfig.pcRxBuffer                     = cRxBuffer;
  stcConfig.u16RxBufferSize                = BUFFERSIZE_SYNC;
  stcConfig.enTxCallbackMode               = MfsOnTxBufferEmpty; // not used
  stcConfig.pfnTxCallback                  = NULL;
  stcConfig.u16RxCallbackBufferFillLevel = 1;
  stcConfig.pfnRxCallback                  = NULL;

  if (Ok == Mfs_Init((stc_mfsn_t*)&MFS1, &stcConfig)
  {
    pstcMfs = (stc_mfsn_t*)&MFS1;   // Setup pointer to MFS instance

                                                                      ▼
```

```
                                                                         ▲
    // Fill Tx Buffer
    au8TxBuffer[0] = 0x11;  // Some meaningful command for SPI slave ...
    au8TxBuffer[1] = 0x22;
    au8TxBuffer[2] = 0x33;
    au8TxBuffer[3] = 0x44;
    au8RxBuffer[0] = 0;   // Clear reception buffer (not needed)
    au8RxBuffer[1] = 0;
    au8RxBuffer[2] = 0;
    au8RxBuffer[3] = 0;

    // Transfer 4 bytes (SPI 'command')
    Mfs_SynchronousTransfer(pstcMfs,
                            (uint8_t*) &cTxBuffer,
                            (uint8_t*) &cRxBuffer,
                            4);
    // Fill Tx Buffer with dummy bytes (SOT remains mark level on next 2 frames)
    au8TxBuffer[0] = 0xFF;
    au8TxBuffer[1] = 0xFF;

    // Now receive 2 bytes (SPI slave response)
    Mfs_SynchronousTransfer(pstcMfs,
                            (uint8_t*) &cTxBuffer,
                            (uint8_t*) &cRxBuffer,
                            2);

    // au8RxBuffer contains the SPI slave response now
    if ((au8RxBuffer[0] == 0x99) &&
        (au8RxBuffer[1] == 0x88))
    {
      // That was the answer we expected from the SPI slave ...
    }

    . . .

    Mfs_DeInit(pstcMfs);
  }
}
```

5.20.35.4 I^{2}C Communication

The following code example shows how to establish an I^{2}C communication acting the MFS as master. It is assumed, that the slave is an serial I^{2}C EEPROM.

The example does the following steps:

1. Write a single byte (0x12) to the EEPROM’s address 0x80 and wait ~10ms.
2. Read back this data from address 0x80.
3. Perform a sequential write to the EEPROM with 8 bytes of ‘random’ data starting from address 0x80 and wait ~10ms.
4. Read back sequentially these data from EEPROM starting from address 0x80.

```
#include "mfs.h"

#define BUFFERSIZE        256
#define I2C_SLAVE_ADDRESS 0          // EEPROM I2C slave address
#define EEPROM_24C04      0x50       // upper address for EEPROM

char cRxBuffer[BUFFERSIZE];  // Reception Ring Buffer
char cTxBuffer[BUFFERSIZE];  // Transmission Ring Buffer

// Delay function (may be replaced by timer)
void MfsWait(volatile uint32_t u32WaitCount)
{
  while(u32WaitCount--);
}

function
{
  stc_mfs_config_t stcConfig;        // MFS configuration
  stc_mfsn_t*       pstcMfs = NULL;  // MFS instance

  L3_ZERO_STRUCT(stcConfig);

  char      au8RxBuffer[8];
  char      au8TxBuffer[8];
  uint32_t u32ErrorCode;
  uint8_t  u8Counter;

  stcConfig.enMode                          = MfsI2cMaster;
  stcConfig.enClockSource                   = MfsUseInternalBaudRateGenerator;
  stcConfig.u32DataRate                     = 400000; // 400 KBit/s

  if (Ok == Mfs_Init((stc_mfsn_t*)&MFS1, &stcConfig)
  {
    pstcMfs = (stc_mfsn_t*)&MFS1;  // Setup pointer to USART instance

    // *** Step 1: Single data write
    au8TxBuffer [0] = 0x80;    // EEPROM memory address
    au8TxBuffer [1] = 0x12;    // EEPROM memory data
    au8RxBuffer [0] = 0;       // reception data

    // Write data to EEPROM (generate start condition)
    u32ErrorCode = Mfs_I2C_Start(pstcMfs,
                              (EEPROM_24C04 | I2C_SLAVE_ADDRESS), I2CWrite);
    if (Ok != u32ErrorCode)
    {
      // Error Handling
    }

    // Write data to EEPROM (data write)
    u32ErrorCode = Mfs_I2C_WriteSync(pstcMfs, (uint8_t*) au8TxBuffer, 2);
    if (Ok != u32ErrorCode)
    {
      // Error Handling
    }

    // Write data to EEPROM (generate stop condition)
    u32ErrorCode = Mfs_I2C_Stop(pstcMfs, I2CWrite);
    if (Ok != u32ErrorCode)
    {
      // Error Handling
    }
```

▼

▲

```
MfsWait(400000);  // Give EEPROM ~10ms time for writing data internally.
                  //   No read access is allowed in this time!

// *** Step 2: Read back data from EEPROM
u32ErrorCode = Mfs_I2C_Start(pstcMfs,
                             EEPROM_24C04 | I2C_SLAVE_ADDRESS, I2CWrite);
if (Ok != u32ErrorCode)
{
  // Error Handling
}

u32ErrorCode = Mfs_I2C_WriteSync(pstcMfs, (uint8_t*) au8TxBuffer, 1);
if (Ok != u32ErrorCode)
{
  // Error Handling
}

u32ErrorCode = Mfs_I2C_Continue(pstcMfs,
                                EEPROM_24C04 | I2C_SLAVE_ADDRESS, I2CRead);
if (Ok != u32ErrorCode)
{
  // Error Handling
}

u32ErrorCode = Mfs_I2C_ReadSync(pstcMfs, (uint8_t*) au8RxBuffer, 1);
if (Ok != u32ErrorCode)
{
  // Error Handling
}

u32ErrorCode = Mfs_I2C_Stop(pstcMfs, I2CRead);
if (Ok != u32ErrorCode)
{
  // Error Handling
}

// *** Step 3: Sequential write

// Make Dummy data
for (u8Counter = 0; u8Counter < 8; u8Counter++)
{
  au8TxBuffer[u8Counter] = u8Counter * 77 ^ 0xAA + u8Counter * 5;
}

// Write data to EEPROM
u32ErrorCode = Mfs_I2C_Start(pstcMfs,
                             (EEPROM_24C04 | I2C_SLAVE_ADDRESS), I2CWrite);
if (Ok != u32ErrorCode)
{
  // Error Handling
}

// Write address
au8TxBuffer [10] = 0x80;      // Address
u32ErrorCode = Mfs_I2C_WriteSync(pstcMfs, (uint8_t*) &au8TxBuffer[10], 1);
if (Ok != u32ErrorCode)
{
  // Error Handling
}

// Write data sequentially
u32ErrorCode = Mfs_I2C_WriteSync(pstcMfs, (uint8_t*) au8TxBuffer, 8);
```

▼

▲

```
    if (Ok != u32ErrorCode)
    {
      // Error Handling
    }


    u32ErrorCode = Mfs_I2C_Stop(pstcMfs, I2CWrite);
    if (Ok != u32ErrorCode)
    {
      // Error Handling
    }

    MfsWait(400000);  // Give EEPROM ~10ms time for writing data internally.
                      //   No read access is allowed in this time!

    // *** Step 4: Sequential read back data from EEPROM
    u32ErrorCode = Mfs_I2C_Start(pstcMfs,
                                 EEPROM_24C04 | I2C_SLAVE_ADDRESS, I2CWrite);
    if (Ok != u32ErrorCode)
    {
      // Error Handling


    // Write address
    u32ErrorCode = Mfs_I2C_WriteSync(pstcMfs, (uint8_t*) &au8TxBuffer [10], 1);
    if (Ok != u32ErrorCode)
    {
      // Error Handling
    }

    u32ErrorCode = Mfs_I2C_Continue(pstcMfs,
                                    EEPROM_24C04 | I2C_SLAVE_ADDRESS, I2CRead);
    if (Ok != u32ErrorCode)
    {
      // Error Handling
    }

    // Read data sequentially
    u32ErrorCode = Mfs_I2C_ReadSync(pstcMfs, (uint8_t*) au8RxBuffer, 8);
    if (Ok != u32ErrorCode)
    {
      // Error Handling
    }

    u32ErrorCode = Mfs_I2C_Stop(pstcMfs, I2CRead);
    if (Ok != u32ErrorCode)
    {
      // Error Handling
    }

    . . .
```

5.20.35.5 LIN Master communication

The following code shows an example of LIN master communication. It sends out a master task Lin frame and then a slave task Lin frame. Note, that an external Lin transceiver is assumed, which echoes every transmission.

```
#include "mfs.h"

char cRxBuffer[8];  // Reception Ring Buffer
char cTxBuffer[8];  // Transmission Ring Buffer

uint32_t u32CountRxCb;

void Mfs6LbrCallback(void)
{
  // do something ...
}

void Mfs6RxCallback(void)
{
  u32CountRxCb++;
}

function
{
  stc_mfs_lin_config_t stcLinConfig1;       // MFS Lin configuration
  stc_mfsn_t*          pstcMfs = NULL;      // MFS instance

  L3_ZERO_STRUCT(stcLinConfig1);

  char      au8RxBuffer[8];
  char      au8TxBuffer[8];
  uint16_t u16ReadCount;

  stcLinConfig1.enLinMode             = LinMaster;
  stcLinConfig1.bExtWakeUp            = FALSE;
  stcLinConfig1.u32DataRate           = 19200;
  stcLinConfig1.enLinStopBits         = LinOneStopBit;
  stcLinConfig1.enLinBreakLength      = LinBreakLength16;
  stcLinConfig1.enLinDelimiterLength = LinDelimiterLength1;
  stcLinConfig1.bLinBreakIrqEnable   = FALSE;
  stcLinConfig1.pcLinTxBuffer         = cTxBuffer;
  stcLinConfig1.u16LinTxBufferSize   = 8;
  stcLinConfig1.pcLinRxBuffer         = cRxBuffer;
  stcLinConfig1.u16LinRxBufferSize   = 8;
  stcLinConfig1.pfnLinBreakCallback  = &Mfs6LbrCallback;
  stcLinConfig1.pfnLinRxCallback      = &Mfs6RxCallback;
  stcLinConfig1.pfnLinTxCallback      = NULL;

  if (Ok == Mfs_Lin_Init((stc_mfsn_t*)&MFS6, &stcLinConfig1)
  {
    pstcMfs = (stc_mfsn_t*)&MFS6;  // Setup pointer to USART instance

    // LIN Master Task
    au8TxBuffer[0] = 0x55;  // Synch Field
    au8TxBuffer[1] = 'M';   // Header (no LIN meaning, just test byte)
    au8TxBuffer[2] = 'S';   // Data (no LIN meaning, just test byte)
    au8TxBuffer[3] = 'T';   // Checksum (no LIN meaning, just test byte)

    // First Set LIN Break for master task
    Mfs_Lin_SetBreak(pstcMfs);

    // Prepare LIN Master Reception Buffer
    Mfs_Read(pstcMfs, (char*) &au8RxBuffer, &u16ReadCount, 4, FALSE);
    u32CountRxCb = 0;

    // Write rest of master task LIN Frame
    Mfs_Write(pstcMfs, (char*) &au8TxBuffer, 4, FALSE);

    // Wait for all data read back master task
    while(u32CountRxCb < 4);
```

▼

```
                                                                          ▲

    au8TxBuffer[1] = 'S';   // Header (no LIN meaning, just test byte)

    // Set LIN Break for slave task
    Mfs_Lin_SetBreak(pstcMfs);

    // Prepare LIN Master Reception Buffer (complete frame)
    Mfs_Read(pstcMfs, (char*) &au8RxBuffer, &u16ReadCount, 4, FALSE);
    u32CountRxCb = 0;

    // Write rest of slave task LIN Frame (synch field and header)
    Mfs_Write(pstcMfs, (char*) &au8TxBuffer, 2, FALSE);

    // Wait for all data read back: field, header, and slave task
    while(u32CountRxCb < 4);

    . . .

  }
}
```

5.20.35.6 LIN Slave communication

The following code shows an example of LIN master communication. It receives a master task Lin frame and then sends a slave task Lin frame. Note, that an external Lin transceiver is assumed, which echoes every transmission.

```
#include "mfs.h"

char cRxBuffer[8];  // Reception Ring Buffer
char cTxBuffer[8];  // Transmission Ring Buffer

uint32_t u32CountRxCb;
uint32_t u32LinBreak;

void Mfs4LbrCallback(void)
{
  u32LinBreak = 1;
}

void Mfs4RxCallback(void)
{
  u32CountRxCb++;
}

function
{
  stc_mfs_lin_config_t stcLinConfig2;       // MFS Lin configuration
  stc_mfsn_t*          pstcMfs = NULL;      // MFS instance

  L3_ZERO_STRUCT(stcLinConfig2);

  char      au8RxBuffer[8];
  char      au8TxBuffer[8];
  uint16_t u16ReadCount;

  stcLinConfig2.enLinMode             = LinSlave;
  stcLinConfig2.bExtWakeUp            = FALSE;
  stcLinConfig2.u32DataRate           = stcLinConfig1.u32DataRate;
  stcLinConfig2.enLinStopBits         = stcLinConfig1.enLinStopBits;
  stcLinConfig2.enLinDelimiterLength = stcLinConfig1.enLinDelimiterLength;
  stcLinConfig2.bLinBreakIrqEnable    = TRUE;
  stcLinConfig2.pcLinTxBuffer         = cTxBuffer2;
  stcLinConfig2.u16LinTxBufferSize    = 8;
  stcLinConfig2.pcLinRxBuffer         = cRxBuffer2;
  stcLinConfig2.u16LinRxBufferSize    = 8;
  stcLinConfig2.pfnLinBreakCallback   = &Mfs4LbrCallback;
  stcLinConfig2.pfnLinRxCallback      = &Mfs4RxCallback2;
  stcLinConfig2.pfnLinTxCallback      = NULL;

  if (Ok == Mfs_Lin_Init((stc_mfsn_t*)&MFS4, &stcLinConfig2)
  {
    pstcMfs = (stc_mfsn_t*)&MFS4;  // Setup pointer to USART instance

    while(0 == u32LinBreak);       // Wait for Lin master’s break
    u32LinBreak = 0;

    // Prepare read of Lin master task
    u32CountRxCb = 0;
    Mfs_Read(pstcMfs, (char*) &au8RxBuffer, &u16ReadCount, 4, FALSE);

    // Wait for all data read
    while(u32CountRxCb < 4);

    // Process data ...

    while(0 == u32LinBreak);       // Wait for next Lin master’s break
    u32LinBreak = 0;
```

▼

▲

```
    // Prepare read of Lin slave task
    u32CountRxCb = 0;
    Mfs_Read(pstcMfs, (char*) &au8RxBuffer, &u16ReadCount, 2, FALSE);

    // Prepare slave task data
    au8TxBuffer[0] = 'S';   // Data (no LIN meaning, just test byte)
    au8TxBuffer[1] = 'T';   // Checksum (no LIN meaning, just test byte)

    // ... and send data
    u32CountRxCb = 0;
    Mfs_Write(pstcMfs, (char*) &au8TxBuffer, 2, FALSE);

    // Wait for all data read
    while(u32CountRxCb < 2);

    . . .

  }
}
```

5.21 (MFT) Multi Function Timer

Type Definition	`stc_mftn_t`
Configuration Type (FRT and OCU)	`stc_mft_frt_ocu_config_t`
Configuration Type (WFG)	`stc_mft_wfg_config_t`
Configuration Type (PPG)	`stc_mft_ppg_config_t`
Configuration Type (ICU)	`stc_mft_icu_config_t`
Configuration Type (ADCMP)	`stc_mft_adcmp_config_t`
Configuration Type (NZCL)	`stc_mft_nzcl_config_t`
Configuration Type (IGBT)	`stc_mft_igbt_config_t`
Address Operator	`MFT`*n*

A Multi Function Timer instance consists of the following components:

- 3 FRT (3 Free-Run Timer)*
- OCU (6 Output Compare Unit Channels)*
- 3 WFG (3 Waveform Generator Units)
- 3 PPG (3 dedicated PPG Channels)
- 4 ICU (4 Input Capture Channels)
- 3 ACMP (3 ADC Compare Trigger Units)
- 1 NZCL (1 Noise Cancel/Emergency stop Unit)
- 1 IGBT (1 IGBT module for Tpye7 devices only)

The components marked by * above are needed for the minimum configuration stc_mft_frt_ocu_config_t to initialize an MFT instance. Thus the FRT and OCU have to be configured to generate an output signal on the `RTOx` pins. If the WFG's configuration is set to `NULL`, it is automatically initialized to 'through mode'. In this case the PPGs can only generate interrupts. If the PPG shall modulate the OCU signals, the WFG has to to be configured for PPG. For only using PPGs the OCU configuration has to be set to disabled OCU channels.

Therefore in contrast to other resource drivers the MFT has more than one configuration to save configuration memory of not used components of an MFT instance.

The configurations for the MFT components are:

- `stc_mft_frt_ocu_config_t` (FRT and OCU)
- `stc_mft_wfg_config_t` (WFG)
- `stc_mft_ppg_config_t` (PPG)
- `stc_mft_icu_config_t` (ICU)
- `stc_mft_adcmp_config_t` (ADCMP)
- `stc_mft_nzcl_config_t` (NZCL)
- `stc_mft_ppg_igbt_config_t` (IGBT for Type 7 devices only)

`Mft_Init()` initializes an MFT instance. The minimum configuration, which is needed is defined by `stc_mft_frt_ocu_config_t`. In this case, the MFT's corresponding OCUs are routed directly to the `RTO`*`ni`* pins, where *`n`* is the number of the MFT instance and *`i`* the output pin from `0` to `5`. The WFG is set automatically to 'through mode' by `Mft_Init()`. `Mft_FrtStart()` starts the FRT operation together with the OCU's configuration. Note, that because of different start control bit address locations, the (maximum) three FRTs are not started synchronously! This driver starts FRT0 first, then FRT1 and at last FRT2. Same appears to `Mft_FrtStop()`, which stops the FRTs given in the function arguments. FRT0 is stopped first and FRT2 at last.

`Mft_FrtRead()` reads out the recent FRT counter value.

Each FRT and OCU interrupt has an own callback function. The pointer to these (user) functions can be defined in the `stc_mft_frt_ocu_config_t` configuration. If an interrupt is enabled in the configuration, the driver's ISRs are taking care of clearing the interrupt cause and calling the callback functions, if their pointers are unequal to `NULL`.

The OCU channels can be enabled by `Mft_Init()` or later enabled or disabled individually by `Mft_OcuStartStop()` with their configuration `stc_mft_frt_ocu_config_t`.

The ICU channels are not enabled by `Mft_Init()`. For individually en- and disabling ICU channels by its configuration `stc_mft_icu_config_t` the function `Mft_IcuStartStop()` is used.

`Mft_IcuRead()` reads out the captured FRT value of an selected ICU channel and can be called in the ICU's channels corresponding callback function.

This driver only configures and handles the PPG channels `0`, `2`, and `4` for MFT0 and channels `8`, `10`, `12` for MFT1, because only these channels are connected to the WFG (outputs). Defining configurations for all PPGs (for internal counting purposes only) would oversize the configuration itself and the RAM consumption.

`Mft_PpgSwTrigger()` triggers selected PPG channels, if configured to `en_mft_ppg_trigger_t::MftPpgTriggerRegister`. `Mft_PpgTimerStart()` starts PPG channels with the up-counter delay synchronously, if configured to `en_mft_ppg_trigger_t::MftPpgTimingGenerator`. `Mft_PpgTimerStop()` stops selected PPGs.

With `Mft_PpgSetValue8()`, `Mft_PpgSetValue16()`, and `Mft_PpgSetValue8plus8()` new pulse values can be set corresponding to the PPG mode defined in `en_mft_ppg_mode_t`.

Note that for generating PPG-only signals `bRt[`*n*`]HighLevel` of `stc_mft_frt_ocu_config_t` has to be set to `TRUE`.

`Mft_DeInit()` deinitializes an MFT instance. Note, that also here `NULL` pointers to not defined configurations cause no touch of the corresponding MFT parts.

On Tpye7 devices the following IGBT functions are also available:

- `Mft_PpgIgbt_Init()`
- `Mft_PpgIgbt_Enable()`
- `Mft_PpgIgbt_Disable()`
- `Mft_PpgIgbt_DeInit()`

Note:

To save configuration memory 16 + 16 bit mode is only available if the definition `L3_PPG_1616_MODE_AVAILABLE` in *l3_user.h* is set to `L3_ON`.

Caution:

This driver only supports Interrupt A ISR handling for MFT!

5.21.1 Configuration Structures

The MFT subresources uses the following configuration structures:

5.21.1.1 Free-Run Timer and Output Compare Unit Configuration

The FRT and OCU share the configuration of the type of `stc_mft_frt_ocu_config_t`:

Type	Field	Possible Values	Description

`en_mft_clock_t`	`enClock0`	`MftPclk1`
		`MftPclk2`
		`MftPclk4`
		`MftPclk8`
		`MftPclk16`
		`MftPclk32`
		`MftPclk64`
		`MftPclk128`
		`MftPclk256`
			FRT0 count clock cycle: `PCLK`
			FRT0 count clock cycle: `PCLK * 2`
			FRT0 count clock cycle: `PCLK * 4`
			FRT0 count clock cycle: `PCLK * 8`
			FRT0 count clock cycle: `PCLK * 16`
			FRT0 count clock cycle: `PCLK * 32`
			FRT0 count clock cycle: `PCLK * 64`
			FRT0 count clock cycle: `PCLK * 128`
			FRT0 count clock cycle: `PCLK * 256`

`en_mft_clock_t`	`enClock1`	*see above*	*see above for FRT1*
`en_mft_clock_t`	`enClock2`	*see above*	*see above for FRT2*
`en_mft_mode_t`	`enMode0`	`MftUpCount` `MftUpDownCount`	FRT0 up-count mode FRT0 up-/down-count mode
`en_mft_mode_t`	`enMode1`	*see above*	*see above for FRT1*
`en_mft_mode_t`	`enMode2`	*see above*	*see above for FRT2*

`boolean_t`	`bTccp0…`
	`Buffer…`
	`Enable`
		`TRUE`
		`FALSE`
			Enables `TCCP0` buffering
			Disables `TCCP0` buffering

`boolean_t`	`bTccp1…`
	`Buffer…`
	`Enable`
		`TRUE`
		`FALSE`
			Enables `TCCP1` buffering
			Disables `TCCP1` buffering

`boolean_t`	`bTccp2…`
	`Buffer…`
	`Enable`
		`TRUE`
		`FALSE`
			Enables `TCCP2` buffering
			Disables `TCCP2` buffering

`boolean_t`	`bTcsa0…`
	`Iclr…`
	`Interrupt…`
	`Enable`
		`TRUE`
		
		`FALSE`
			Enables interrupt generation, if '`1`' is set to `TCSA0.ICLR`
			Disables interrupt generation

`boolean_t`	`bTcsa1…`
	`Iclr…`
	`Interrupt…`
	`Enable`
		`TRUE`
		
		`FALSE`
			Enables interrupt generation, if '`1`' is set to `TCSA1.ICLR`
			Disables interrupt generation

`boolean_t`	`bTcsa2…`
	`Iclr…`
	`Interrupt…`
	`Enable`
		`TRUE`
		
		`FALSE`
			Enables interrupt generation, if '`1`' is set to `TCSA2.ICLR`
			Disables interrupt generation

`boolean_t`	`bTcsa0…`
	`Irqzf…`
	`Interrupt…`
	`Enable`
		`TRUE`
		
		`FALSE`
			Enables interrupt generation, if '`1`' is set to `TCSA0.IRQZF`
			Disables interrupt generation

`boolean_t`	`bTcsa1…`
	`Irqzf…`
	`Interrupt…`
	`Enable`
		`TRUE`
		
		`FALSE`
			Enables interrupt generation, if '`1`' is set to `TCSA1.IRQZF`
			Disables interrupt generation

`boolean_t`	`bTcsa2…`
	`Irqzf…`
	`Interrupt…`
	`Enable`
		`TRUE`
		
		`FALSE`
			Enables interrupt generation, if '`1`' is set to `TCSA2.IRQZF`
			Disables interrupt generation

`boolean_t`	`bExternal…` `Clock0…`	`TRUE` `FALSE`	Enables external clock for FRT0 Uses internal clock for FRT0

	Enable		
boolean_t	bExternal… Clock1… Enable	TRUE FALSE	Enables external clock for FRT1 Uses internal clock for FRT1
boolean_t	bExternal… Clock2… Enable	TRUE FALSE	Enables external clock for FRT2 Uses internal clock for FRT2
uint16_t	u16Frt0… Cycle	–	FRT0 cycle setting
uint16_t	u16Frt1… Cycle	–	FRT1 cycle setting
uint16_t	u16Frt2… Cycle	–	FRT2 cycle setting
en_mft_ocu_… connect_t	enFrtOcu0… Connection	MftFrt0Ocu MftFrt0Ocu MftFrt0Ocu MftOcuFrtMft1 MftOcuFrtMft1	Connects FRT0 to OCU0 Connects FRT0 to OCU0 Connects FRT0 to OCU0 Connects FRT of external MFT1 Connects FRT of external MFT2
...	...	...	*Same for OCU1, OCU2, OCU3, OCU4*
en_mft_ocu_… connect_t	enFrtOcu5… Connection	MftFrt0Ocu MftFrt0Ocu MftFrt0Ocu MftOcuFrtMft1 MftOcuFrtMft1	Connects FRT0 to OCU5 Connects FRT0 to OCU5 Connects FRT0 to OCU5 Connects FRT of external MFT1 Connects FRT of external MFT2
boolean_t	bOcu0… Enable	TRUE FALSE	Enables operation of OCU0 Disable operation of OCU0
...	...	...	*Same for OCU1, OCU2, OCU3, OCU4*
boolean_t	bOcu5… Enable	TRUE FALSE	Enables operation of OCU5 Disable operation of OCU5
boolean_t	bOccp0… Buffer… Enable	TRUE FALSE	Enables operation of OCU0 Disable operation of OCU0
...	...	...	*Same for OCU1, OCU2, OCU3, OCU4*
boolean_t	bOccp5… Buffer… Enable	TRUE FALSE	Enables operation of OCU5 Disable operation of OCU5
boolean_t	bOcsaIop0… Interrupt… Enable	TRUE FALSE	Enables interrupt generation, if '1' is set to OCSA.IOP0 Disables interrupt generation
...	...	...	*Same for IOP1, IOP2, IOP3, IOP4*
boolean_t	bOcsaIop5… Interrupt… Enable	TRUE FALSE	Enables interrupt generation, if '1' is set to OCSA.IOP5 Disables interrupt generation
boolean_t	bRt0High… Level	TRUE FALSE	Sets RT0 to high level (only if OCSA.CST0 == 0) Does not set high level
...	...	...	*Same for RT1, RT2, RT3, RT4*
boolean_t	bRt5High… Level	TRUE FALSE	Sets RT5 to high level (only if OCSA.CST5 == 0) Does not set high level
boolean_t	bCmod01	TRUE FALSE	Sets common mode of RT0 and RT1 Does not set common mode

`boolean_t`	`bCmod23`	`TRUE` `FALSE`	Sets common mode of RT2 and RT3 Does not set common mode
`boolean_t`	`bCmod45`	`TRUE` `FALSE`	Sets common mode of RT4 and RT5 Does not set common mode
`boolean_t`	`bOccp0Buf… Transfer… Peak… Detection`	`TRUE` `FALSE`	Performs buffer transfer of OCCP0 upon peak value detection Performs buffer transfer of OCCP0 upon zero value detection
...	...	...	*Same for OCU1, OCU2, OCU3, OCU4*
`boolean_t`	`bOccp5Buf… Transfer… Peak… Detection`	`TRUE` `FALSE`	Performs buffer transfer of OCCP5 upon peak value detection Performs buffer transfer of OCCP5 upon zero value detection
`boolean_t`	`bMode0`	`TRUE` `FASLE`	operation mode of OCU0 in combination with `bCmod01` No combination with `bCmod01`
...	...	...	*Same for OCU1, OCU2, OCU3, OCU4*
`boolean_t`	`bMode5`	`TRUE` `FASLE`	operation mode of OCU5 in combination with `bCmod45` No combination with `bCmod45`
`uint16_t`	`u16Ocu0… Compare… Value`	-	Sets the OCU0 compare value
...	...	...	*Same for OCU1, OCU2, OCU3, OCU4*
`uint16_t`	`u16Ocu5… Compare… Value`	-	Sets the OCU5 compare value
`func_ptr_t`	`pfnFrt0… Peak… Callback`	-	Callback function pointer of FRT0 peak detection interrupt
`func_ptr_t`	`pfnFrt1… Peak… Callback`	-	Callback function pointer of FRT1 peak detection interrupt
`func_ptr_t`	`pfnFrt2… Peak… Callback`	-	Callback function pointer of FRT2 peak detection interrupt
`func_ptr_t`	`pfnFrt0… Zero… Callback`	-	Callback function pointer of FRT0 zero detection interrupt
`func_ptr_t`	`pfnFrt1… Zero… Callback`	-	Callback function pointer of FRT1 zero detection interrupt
`func_ptr_t`	`pfnFrt2… Zero… Callback`	-	Callback function pointer of FRT2 zero detection interrupt
`func_ptr_t`	`pfnOcu0… Callback`	-	Callback function pointer of OCU0 interrupt
...	...	...	*Same for OCU1, OCU2, OCU3, OCU4*
`func_ptr_t`	`pfnOcu5… Callback`	-	Callback function pointer of OCU5 interrupt

5.21.1.2 Waveform Generator Configuration

The WFG has the configuration of the type of `stc_mft_wfg_config_t`:

Type	Field	Possible Values	Description

`en_mft_wfg_clock_t`	`enWfgClock01`	`MftWfgPclk1`
		`MftWfgPclk2`
		`MftWfgPclk4`
		`MftWfgPclk8`
		`MftWfgPclk16`
		`MftWfgPclk32`
		`MftWfgPclk64`
			WFG01 count clock cycle: `PCLK`
			WFG01 count clock cycle: `PCLK * 2`
			WFG01 count clock cycle: `PCLK * 4`
			WFG01 count clock cycle: `PCLK * 8`
			WFG01 count clock cycle: `PCLK * 16`
			WFG01 count clock cycle: `PCLK * 32`
			WFG01 count clock cycle: `PCLK * 64`

`en_mft_wfg_clock_t`	`enWfgClock23`	*see above*	*see above for WFG23*
`en_mft_wfg_clock_t`	`enWfgClock45`	*see above*	*see above for WFG45*

`en_mft_wfg_mode_t`	`enWfgMode01`	`MftWfgThrough…`
		`Mode`
		`MftWfgRtPpg…`
		`Mode`
		`MftWfgTimer…`
		`PpgMode`
		`MftWfgRtDead…`
		`TimerMode`
		`MftWfgPpgDead…`
		`TimerMode`
			WFG01 operation in through mode
			
			WFG01 operation in RT-PPG mode
			
			WFG01 operation in Timer-PPG mode
			
			WFG01 operation in RT-Dead Timer mode
			
			WFG01 operation in PPG-Dead Timer mode

`en_mft_wfg_mode_t`	`enWfgMode23`	*see above*	*see above for WFG23*
`en_mft_wfg_mode_t`	`enWfgMode23`	*see above*	*see above for WFG45*

`en_mft_wfg_gate_t`	`enWfgGate01`	`MftWfgLowLevel`
		`MftWfgRt0Flag0`
		`MftWfgRt1Flag1`
		`MftWfgRt10Flag10`
			Outputs low level signal
			Outputs RT(0), Timer active Flag0
			Outputs RT(0), Timer active Flag1
			Or'ed signal from Flag0 and Flag1

`en_mft_wfg_gate_t`	`enWfgGate23`	`MftWfgLowLevel`
		`MftWfgRt0Flag0`
		`MftWfgRt1Flag1`
		`MftWfgRt10Flag10`
			Outputs low level signal
			Outputs RT(2), Timer active Flag0
			Outputs RT(2), Timer active Flag1
			Or'ed signal from Flag0 and Flag1

`en_mft_wfg_gate_t`	`enWfgGate34`	`MftWfgLowLevel`
		`MftWfgRt0Flag0`
		`MftWfgRt1Flag1`
		`MftWfgRt10Flag10`
			Outputs low level signal
			Outputs RT(4), Timer active Flag0
			Outputs RT(4), Timer active Flag1
			Or'ed signal from Flag0 and Flag1

`en_mft_wfg_psel_t`	`enWfgPpg…`
	`Sel01`
		`MftWfgGatePpg0`
		`MftWfgGatePpg2`
		`MftWfgGatePpg4`
			WFG01 Gate PPG0 enable
			WFG01 Gate PPG2 enable
			WFG01 Gate PPG4 enable

`en_mft_wfg_psel_t`	`enWfgPpg…`
	`Sel23`
		`MftWfgGatePpg0`
		`MftWfgGatePpg2`
		`MftWfgGatePpg4`
			WFG23 Gate PPG0 enable
			WFG23 Gate PPG2 enable
			WFG23 Gate PPG4 enable

`en_mft_wfg_psel_t`	`enWfgPpg…`
	`Sel45`
		`MftWfgGatePpg0`
		`MftWfgGatePpg2`
		`MftWfgGatePpg4`
			WFG45 Gate PPG0 enable
			WFG45 Gate PPG2 enable
			WFG45 Gate PPG4 enable

`en_mft_wfg_pgen_t`	`enWfgPpg…`
	`Enable01`
		`MftWfgRt1Rt0`
		`MftWfgRt1Ppg`
		`MftWfgPpgRt0`
		`MftWfgPpgPpg`
			RTO1 = RT1, RTO0 = RT0
			RTO1 = RT1, PTO0 = PPGn
			RTO1 = PPGm, RTO0 = RTO0
			RTO1 = PPGm, RTO0 = PPGn

`en_mft_wfg_pgen_t`	`enWfgPpg…`
	`Enable23`
		`MftWfgRt1Rt0`
		`MftWfgRt1Ppg`
		`MftWfgPpgRt0`
		`MftWfgPpgPpg`
			RTO3 = RT3, RTO2 = RT2
			RTO3 = RT3, PTO2 = PPGn
			RTO3 = PPGm, RTO2 = RTO2
			RTO3 = PPGm, RTO2 = PPGn

en_mft_wfg_pgen_t	enWfgPpg…
	Enable23
		MftWfgRt1Rt0
		MftWfgRt1Ppg
		MftWfgPpgRt0
		MftWfgPpgPpg
			RTO5 = RT5, RTO4 = RT4
			RTO5 = RT5, PTO4 = PPGn
			RTO5 = PPGm, RTO4 = RTO4
			RTO5 = PPGm, RTO4 = PPGn

boolean_t	bNon… Overlap… High… Polarity0	TRUE FALSE	Sets output of the non-overlap to active high polarity FTR0. Does not set non-overlap output.
boolean_t	bNon… Overlap… High… Polarity1	TRUE FALSE	Sets output of the non-overlap to active high polarity FTR1. Does not set non-overlap output.
boolean_t	bNon… Overlap… High… Polarity2	TRUE FALSE	Sets output of the non-overlap to active high polarity FTR2. Does not set non-overlap output.
uint16_t	u16Wfg01… TimerValue	-	WFG0/1 timer value
uint16_t	u16Wfg23… TimerValue	-	WFG2/3 timer value
uint16_t	u16Wfg45… TimerValue	-	WFG4/5 timer value
func_ptr_t	pfnWfg01… Callback	-	Callback function pointer of WFG 0/1 interrupt
func_ptr_t	pfnWfg23… Callback	-	Callback function pointer of WFG 2/3 interrupt
func_ptr_t	pfnWfg45… Callback	-	Callback function pointer of WFG 4/5 interrupt

5.21.1.3 Programmable Pulse Generator Configuration

The PPG has the configuration of the type of `stc_mft_ppg_config_t`:

Type	**Field**	**Possible Values**	**Description**
en_mft_ppg_mode_t	enPpg0Mode	MftPpg8Bit MftPpg8plus8Bit MftPpg16Bit MftPpg16plus16Bit	PPG0 : 8 Bit Mode PPG0 : 8 + 8 Bit Mode PPG0 : 16 Bit Mode PPG0 : 16 + 16 Bit Mode
en_mft_ppg_mode_t	enPpg2Mode	*see above*	PPG2 : 8 Bit Mode PPG2 : 8 + 8 Bit Mode PPG2 : 16 Bit Mode PPG2 : 16 + 16 Bit Mode
en_mft_ppg_mode_t	enPpg4Mode	*see above*	PPG4 : 8 Bit Mode PPG4 : 8 + 8 Bit Mode PPG4 : 16 Bit Mode PPG4 : 16 + 16 Bit Mode
en_mft_ppg_clock_t	enPpg0Clock	MftPpgPclk2 MftPpgPclk8 MftPpgPclk32 MftPpgPclk64	PPG0: Up count clock: PCLK / 2 PPG0: Up count clock: PCLK / 8 PPG0: Up count clock: PCLK / 32 PPG0: Up count clock: PCLK / 64
en_mft_ppg_clock_t	enPpg2Clock	*see above*	PPG2: Up count clock: PCLK / 2 PPG2: Up count clock: PCLK / 8 PPG2: Up count clock: PCLK / 32 PPG2: Up count clock: PCLK / 64
en_mft_ppg_clock_t	enPpg4Clock	*see above*	PPG4: Up count clock: PCLK / 2

			PPG4: Up count clock: PCLK / 8 PPG4: Up count clock: PCLK / 32 PPG4: Up count clock: PCLK / 64
boolean_t	bPpg0… Invert… Polarity	TRUE FALSE	Invert Polarity to high, if PPG0 is stopped No inversion of Polarity
boolean_t	bPpg2… Invert… Polarity	TRUE FALSE	Invert Polarity to high, if PPG2 is stopped No inversion of Polarity
boolean_t	bPpg4… Invert… Polarity	TRUE FALSE	Invert Polarity to high, if PPG4 is stopped No inversion of Polarity
boolean_t	bPpg0… Interrupt… Enable	TRUE FALSE	Enables PPG0 interrupt Disables PPG0 interrupt
boolean_t	bPpg2… Interrupt… Enable	TRUE FALSE	Enables PPG2 interrupt Disables PPG2 interrupt
boolean_t	bPpg4… Interrupt… Enable	TRUE FALSE	Enables PPG4 interrupt Disables PPG4 interrupt
boolean_t	bPpg0… Interrupt… Mode	TRUE FALSE	Interrupt on PPLH0 underflow Interrupt on PPLL0 underflow
boolean_t	bPpg2… Interrupt… Mode	TRUE FALSE	Interrupt on PPLH2 underflow Interrupt on PPLL2 underflow
boolean_t	bPpg4… Interrupt… Mode	TRUE FALSE	Interrupt on PPLH4 underflow Interrupt on PPLL4 underflow
uint16_t uint8_t uint8_t	u16Ppg0Low u8Ppg0Low u8Ppg0High	– – –	PPG0 low count (16 Bit Mode) PPG0 low count (8, 8+8 Bit mode) PPG0 high count (8, 8+8 Bit mode)
uint16_t uint8_t uint8_t	u16Ppg0High u8Ppg0… Prescaler… Low u8Ppg0… Prescaler… High	– – –	PPG0 high count (16 Bit Mode) PPG0 low count (8, 8+8 Bit mode) PPG0 high count (8, 8+8 Bit mode)
...	...	...	*see above for PPG2*
uint16_t uint8_t uint8_t	u16Ppg4Low u8Ppg4Low u8Ppg4High	– – –	PPG4 low count (16 Bit Mode) PPG4 low count (8, 8+8 Bit mode) PPG4 high count (8, 8+8 Bit mode)
uint16_t uint8_t uint8_t	u16Ppg4High u8Ppg4… Prescaler… Low u8Ppg4… Prescaler… High	– – –	PPG4 high count (16 Bit Mode) PPG4 low count (8, 8+8 Bit mode) PPG4 high count (8, 8+8 Bit mode)
uint16_t	u16Ppg6Low*	–	PPG6 Prescaler in 16+16 Bit Mode*
uint16_t	u16Ppg6High*	–	PPG7 Prescaler in 16+16 Bit Mode*
en_mft_ppg_…	enPpg0…	MftPpgTiming…	Trigger by Timing Generator circuit

trigger_t	Trigger	Generator MftPpgGate… Signal MftPpgTrigger… Register	(Start delay) Trigger by GATE signals of MFT Trigger by TRG register writing (SW trigger)
en_mft_ppg_… trigger_t	enPpg2… Trigger	*see above*	*see above*
en_mft_ppg_… trigger_t	enPpg4… Trigger	*see above*	*see above*
boolean_t	bPpg0… StartLow… Level	TRUE FALSE	PPG0 starts with low level GATE PPG0 starts with high level GATE
boolean_t	bPpg2… StartLow… Level	TRUE FALSE	PPG2 starts with low level GATE PPG2 starts with high level GATE
boolean_t	bPpg4… StartLow… Level	TRUE FALSE	PPG4 starts with low level GATE PPG4 starts with high level GATE
en_mft_ppg_… clock_t	enPpg… Compare… Clock	MftPpgPclk2 MftPpgPclk8 MftPpgPclk32 MftPpgPclk64	PPG: Compare clock: PCLK / 2 PPG: Compare clock: PCLK / 8 PPG: Compare clock: PCLK / 32 PPG: Compare clock: PCLK / 64
uint8_t	u8Ppg0… Compare	-	PPG0 compare value
uint8_t	u8Ppg2… Compare	-	PPG2 compare value
uint8_t	u8Ppg4… Compare	-	PPG4 compare value
func_ptr_t	pfnPpg0… Callback	-	Pointer to PPG0 interrupt callback function
func_ptr_t	pfnPpg2… Callback	-	Pointer to PPG2 interrupt callback function
func_ptr_t	pfnPpg4… Callback	-	Pointer to PPG4 interrupt callback function

* available, if `L3_PPG_1616_MODE_AVAILABLE == L3_ON`

5.21.1.4 Input Capture Unit Configuration

The ICU has the configuration of the type of `stc_mft_icu_config_t`:

Type	Field	Possible Values	Description
en_mft_icu_… connect_t	enIcu0… Connect	MftIcuFrt0 MftIcuFrt1 MftIcuFrt2 MftIcuFrtMft1 MftIcuFrtMft2	Connects FRT0 to ICU0 Connects FRT1 to ICU0 Connects FRT2 to ICU0 Connects FRT0 to external MFT(1) Connects FRT0 to external MFT(2)
. . .	. . .	. . .	*See above for ICU1and ICU2*
en_mft_icu_… connect_t	enIcu3… Connect	MftIcuFrt0 MftIcuFrt1 MftIcuFrt2 MftIcuFrtMft1 MftIcuFrtMft2	Connects FRT0 to ICU3 Connects FRT1 to ICU3 Connects FRT2 to ICU3 Connects FRT0 to external MFT(1) Connects FRT0 to external MFT(2)

en_mft_icu_… edge_t	enIcu0Edge	MftIcuDisable MftIcuRisingEdge MftIcuFallingEdge MftIcuBothEdges	Disables ICU0 Sets ICU0 rising edge detection Sets ICU0 falling edge detection Set ICU0 both edges detection
...	...	...	*See above for ICU1and ICU2*
en_mft_icu_… edge_t	enIcu3Edge	MftIcuDisable MftIcuRisingEdge MftIcuFallingEdge MftIcuBothEdges	Disables ICU3 Sets ICU3 rising edge detection Sets ICU3 falling edge detection Set ICU3 both edges detection
boolean_t	bIcu0… Interrupt… Enable	TRUE FALSE	Enables ICU0 interrupt Disables ICU0 interrupt
...	...	...	*See above for ICU1and ICU2*
boolean_t	bIcu3… Interrupt… Enable	TRUE FALSE	Enables ICU3 interrupt Disables ICU3 interrupt
func_ptr_t	pfnIcu0… Callback	-	Pointer to ICU0 interrupt callback function
...	...	...	*See above for ICU1and ICU2*
func_ptr_t	pfnIcu3… Callback	-	Pointer to ICU3 interrupt callback function

5.21.1.5 ADC Compare Configuration

The ADCMP has the configuration of the type of `stc_mft_adcmp_config_t`:

Type	Field	Possible Values	Description
en_mft_adc_cmp_t	enAdcmp0… Frt	MftAdcCmp… …Disable …MftAdcCmpFrt0 …MftAdcCmpFrt1 …MftAdcCmpFrt2	Disables ADCMP0 Enables ADCMP0 for FRT0 Enables ADCMP0 for FRT1 Enables ADCMP0 for FRT2
en_mft_adc_cmp_t	enAdcmp1… Frt	MftAdcCmp… …Disable …MftAdcCmpFrt0 …MftAdcCmpFrt1 …MftAdcCmpFrt2	Disables ADCMP1 Enables ADCMP1 for FRT0 Enables ADCMP1 for FRT1 Enables ADCMP1 for FRT2
en_mft_adc_cmp_t	enAdcmp2… Frt	MftAdcCmp… …Disable …MftAdcCmpFrt0 …MftAdcCmpFrt1 …MftAdcCmpFrt2	Disables ADCMP2 Enables ADCMP2 for FRT0 Enables ADCMP2 for FRT1 Enables ADCMP2 for FRT2
boolean_t	bFrt0Adc0… Start… Zero… Detection	TRUE FALSE	ADC0 start at zero detection of FRT0 No ADC0 start generation signal
boolean_t	bFrt0Adc1… Start… Zero… Detection	TRUE FALSE	ADC1 start at zero detection of FRT0 No ADC1 start generation signal
boolean_t	bFrt0Adc2… Start… Zero… Detection	TRUE FALSE	ADC2 start at zero detection of FRT0 No ADC2 start generation signal
...	...	...	*See above for ADC0/1/2 and FRT1*

boolean_t	bFrt2Adc0…Start…Zero…Detection	TRUE FALSE	ADC0 start at zero detection of FRT2 No ADC0 start generation signal
boolean_t	bFrt2Adc1…Start…Zero…Detection	TRUE FALSE	ADC1 start at zero detection of FRT2 No ADC1 start generation signal
boolean_t	bFrt2Adc2…Start…Zero…Detection	TRUE FALSE	ADC2 start at zero detection of FRT2 No ADC2 start generation signal
en_mft_adc_cmp_…count_t	enAdcmp0…Count	MftAdcCmpUp…PeakDown MftAdcCmpUp MftAdcCmpPeak…Down MftAdcCmpUp…PeakDown2	ADC0 start, if FRT at Up/Peak/Down-Count ACCP match ADC0 start if FRT at Up-count ACCP match ADC0 start, if FRT is in Peak/Down-Count ACCP match ADC0 start, if FRT is Up-Count ACCP and Peak/Down ACCPDN
...	...	...	*See above for ADC1*
en_mft_adc_cmp_…count_t	enAdcmp2…Count	MftAdcCmpUp…PeakDown MftAdcCmpUp MftAdcCmpPeak…Down MftAdcCmpUp…PeakDown2	ADC2 start, if FRT at Up/Peak/Down-Count ACCP match ADC2 start if FRT at Up-count ACCP match ADC2 start, if FRT is in Peak/Down-Count ACCP match ADC2 start, if FRT is Up-Count ACCP and Peak/Down ACCPDN
boolean_t	bAdcmp0…Buffer…Disable	TRUE FALSE	ACCP0, ACCPDN0 buffering disable ACCP0, ACCPDN0 buffering enable
...	...	...	*See above for ACCP1, ACCPDN1*
boolean_t	bAdcmp2…Buffer…Disable	TRUE FALSE	ACCP2, ACCPDN2 buffering disable ACCP2, ACCPDN2 buffering enable
boolean_t	bAdcmp0…PeakValue…Enable	TRUE FALSE	ACCP0, ACCPDN0 buffer transfer on peak detection ACCP0, ACCPDN0 buffer transfer on zero detection
...	...	...	*See above for ACCP1, ACCPDN1*
boolean_t	bAdcmp2…PeakValue…Enable	TRUE FALSE	ACCP2, ACCPDN2 buffer transfer on peak detection ACCP2, ACCPDN2 buffer transfer on zero detection
uint16_t	u16Adcmp0…Accp	–	ADCMP0 compare value
uint16_t	u16Adcmp1…Accp	–	ADCMP1 compare value
uint16_t	u16Adcmp2…Accp	–	ADCMP2 compare value
uint16_t	u16Adcmp0…Accpdn	–	ADCMPDN0 compare value

uint16_t	u16Adcmp1… Accpdn	–	ADCMPDN1 compare value
uint16_t	u16Adcmp2… Accpdn	–	ADCMPDN2 compare value
en_mft_adc_cmp_… start_t	enAdcmp0… Start	MftAdcCmpAdcmp MftAdcCmpFrt02	ADC0 start on ADCMP signal ADC0 start on or'ed FRT0/2 signal
...	...	...	*See above for ADC1*
en_mft_adc_cmp_… start_t	enAdcmp2… Start	MftAdcCmpAdcmp MftAdcCmpFrt02	ADC2 start on ADCMP signal ADC2 start on or'ed FRT0/2 signal
en_mft_adc_cmp_… start_prio_t	enAdcmp0… StartPrio	MftAdcCmp… AdcmpPrio MftAdcCmp… Frt02Prio	ADC0 priority scan start on ADCMP signal ADC0 priority scan start on or'ed FRT0/2 signal
...	...	...	*See above for ADC1*
en_mft_adc_cmp_… start_prio_t	enAdcmp2… StartPrio	MftAdcCmp… AdcmpPrio MftAdcCmp… Frt02Prio	ADC2 priority scan start on ADCMP signal ADC2 priority scan start on or'ed FRT0/2 signal

5.21.1.6 Noise Canceler Configuration

The NZCL has the configuration of the type of `stc_mft_nzcl_config_t`:

Type	Field	Possible Values	Description
boolean_t	bDtif… Interrupt… Enable	TRUE FALSE	Enable DTIF IRQ from `DTTIX` pin Disables DTIF interrupt
en_mft_nzcl_… time_t	enNoice… CancelTime	MftNzclNo… NoisCancel MftNzclNoise… Cancel4Pclk MftNzclNoise… Cancel8Pclk MftNzclNoise… Cancel16Pclk MftNzclNoise… Cancel32Pclk	Direct DTTIX pin connection w/o noise canceling Noise canceling with 4 PCLK cycles Noise canceling with 8 PCLK cycles Noise canceling with 16 PCLK cycles Noise canceling with 32 PCLK cycles
func_ptr_t	pfnNzcl… Callback	–	Callback function pointer of DTIF interrupt

5.21.1.7 IGBT Configuration

The IGBT has the configuration of the type of `stc_mft_igbt_config_t`. Note that this configuration is only available for Device Type 7.

Type	Field	Possible Values	Description
boolean_t	bStop… ProhibMode	TRUE FALSE	Stop prohibition mode in output active Normal operation
en_mft_ppg_… igbt_… noisefilter_t	enNoise… Filter	MftPpgIgbt… NoFilter MftPpgIgbt…	No noise filter operation

		4PclkFilter MftPpgIgbt… 8PclkFilter MftPpgIgbt… 16PclkFilter MftPpgIgbt… 32PclkFilter	Noise filter with 4 PCLK cycles Noise filter with 8 PCLK cycles Noise filter with 16 PCLK cycles Noise filter with 32 PCLK cycles
boolean_t	bInverted… Output0	TRUE FALSE	Output0 level inverted Output0 level normal
boolean_t	bInverted… Output1	TRUE FALSE	Output1 level inverted Output1 level normal
boolean_t	bInverted… Input	TRUE FALSE	Trigger input level inverted Trigger input normal

5.21.2 Mft_Init()

This function initializes an MFT instance with the given configurations. At least the FRT/OCU configuration of the type `stc_mft_frt_ocu_config_t` must be eqal and its pointer **unequal** to `NULL`. Not used parts of the MFT can be stated by `NULL` pointer to its configurations.

Prototype

```
en_result_t Mft_Init( stc_mftn_t*                 pstcMft,
                      stc_mft_frt_ocu_config_t*   pstcFrtOcuConfig,
                      stc_mft_wfg_config_t*       pstcWfgConfig,
                      stc_mft_nzcl_config_t*      pstcNzclConfig,
                      stc_mft_icu_config_t*       pstcIcuConfig,
                      stc_mft_adcmp_config_t*     pstcAdcmpConfig,
                      stc_mft_ppg_config_t*       pstcPpgConfig )
```

Parameter Name	Description
`[in] pstcMft`	Pointer to MFT instance
`[in] pstcFrtOcuConfig`	Pointer to FRT and OCU configuration structure. This configuration is mandatory!
`[in] pstcWfgConfig`	Pointer to WFG configuration structure. This pointer can be `NULL`.
`[in] pstcNzclConfig`	Pointer to NZCL configuration structure. This pointer can be `NULL`.
`[in] pstcIcuConfig`	Pointer to ICU configuration structure. This pointer can be `NULL`.
`[in] pstcAdcmpConfig`	Pointer to ADCMP configuration structure. This pointer can be `NULL`.
`[in] pstcPpgConfig`	Pointer to PPG configuration structure. This pointer can be `NULL`.

Return Values	Description
`Ok`	MFT initialization successfully done
`ErrorInvalidParameter`	• `pstcMft == NULL` • `pstcFrtOcuConfig == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved) • One or more configuration data wrong • Wrong enumerator used or enumerator not supported

5.21.3 Mft_DeInit()

This function de-initializes an MFT instance and disables all according interrupts.

Prototype	
`en_result_t Mft_DeInit( stc_mftn_t* pstcMft, stc_mft_frt_ocu_config_t* pstcFrtOcuConfig, stc_mft_wfg_config_t* pstcWfgConfig, stc_mft_nzcl_config_t* pstcNzclConfig, stc_mft_icu_config_t* pstcIcuConfig, stc_mft_adcmp_config_t* pstcAdcmpConfig, stc_mft_ppg_config_t* pstcPpgConfig )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
`[in] pstcFrtOcuConfig`	Pointer to FRT and OCU configuration structure. This pointer can be `NULL`.
`[in] pstcWfgConfig`	Pointer to WFG configuration structure. This pointer can be `NULL`.
`[in] pstcNzclConfig`	Pointer to NZCL configuration structure. This pointer can be `NULL`.
`[in] pstcIcuConfig`	Pointer to ICU configuration structure. This pointer can be `NULL`.
`[in] pstcAdcmpConfig`	Pointer to ADCMP configuration structure. This pointer can be `NULL`.
`[in] pstcPpgConfig`	Pointer to PPG configuration structure. This pointer can be `NULL`.
Return Values	**Description**
`Ok`	MFT de-initialization successfully done
`ErrorInvalidParameter`	`pstcMft == NULL`

5.21.4 Mft_FrtStart()

This functions starts given FRTs of an MFT instance. `Mft_init()` must be called before.

Prototype	
`en_result_t Mft_FrtStart( stc_mftn_t* pstcMft, boolean_t bStartFrt0, boolean_t bStartFrt1, boolean_t bStartFrt2 )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
`[in] bStartFrt0`	`TRUE`: Start FRT0 `FALSE`: No FRT0 operation
`[in] bStartFrt1`	`TRUE`: Start FRT1 `FALSE`: No FRT1 operation
`[in] bStartFrt2`	`TRUE`: Start FRT2 `FALSE`: No FRT2 operation
Return Values	**Description**
`Ok`	Dedicated FRTs started.
`ErrorInvalidParameter`	• `pstcMft == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved)

5.21.5 Mft_FrtStop()

This functions stops given FRTs of an MFT instance. `Mft_init()` must be called before.

Prototype	
`en_result_t Mft_FrtStop( stc_mftn_t* pstcMft, boolean_t bStartFrt0, boolean_t bStartFrt1, boolean_t bStartFrt2 )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
`[in] bStartFrt0`	`TRUE`: Stops FRT0 `FALSE`: Preserve FRT0 operation state
`[in] bStartFrt1`	`TRUE`: Stops FRT1 `FALSE`: Preserve FRT1 operation state
`[in] bStartFrt2`	`TRUE`: Stops FRT2 `FALSE`: Preserve FRT2 operation state
Return Values	**Description**
`Ok`	Dedicated FRTs stopped.
`ErrorInvalidParameter`	• `pstcMft == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved)

5.21.6 Mft_FrtRead()

Reads out the current count of a FRT channel of an MFT instance Note, that `0` is returned, if `pstMft == NULL` and `u8FrtChannel > 3` and does not mean the counter value `0`!

Prototype	
`uint16_t Mft_FrtRead( stc_mftn_t* pstcMft, uint8_t u8FrtChannel )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
`[in] u8FrtChannel`	`0, 1, 2` meaning FRT0/1/2
Return Values	**Description**
`0`	• `pstcMft == NULL` • Wrong FRT channel number
`uint16_t`	Recent FRT channel counter value

5.21.7 Mft_OcuStartStop()

Start or Stops OCU operation by configuration The operation of a OCU channel is cotrolled by the FRT/OCU configurations.

Prototype	
`en_result_t Mft_OcuStartStop( stc_mftn_t* pstcMft, stc_mft_frt_ocu_config_t* pstcFrtOcuConfig )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
`[in] pstcFrtOcuConfig`	Pointer to FRT/OCU configuration
Return Values	**Description**
`Ok`	OCU start/stop successfully set.
`ErrorInvalidParameter`	• `pstcMft == NULL` • `pstcFrtOcuConfig == NULL`

5.21.8 Mft_IcuStartStop()

Start or Stops ICU operation by configuration The operation of a ICU channel is cotrolled by the `en_mft_icu_edge_t` configurations.

Prototype	
`en_result_t Mft_IcuStartStop( stc_mftn_t* pstcMft, stc_mft_icu_config_t* pstcIcuConfig )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
`[in] pstcIcuConfig`	Pointer to ICU configuration
Return Values	**Description**
`Ok`	ICU start/stop successfully set.
`ErrorInvalidParameter`	• `pstcMft == NULL` • `pstcFrtOcuConfig == NULL`

5.21.9 Mft_IcuRead()

Reads out the capture count of a ICU channel of an MFT instance Note, that 0 is returned, if `pstMft == NULL` and `u8IcuChannel > 3` and does not mean the counter value `0`!

Prototype	
`uint16_t Mft_IcuRead( stc_mftn_t* pstcMft, uint8_t u8IcuChannel )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
`[in] u8FrtChannel`	`0, 1, 2` meaning ICU0/1/2
Return Values	**Description**
`0`	• `pstcMft == NULL` • Wrong ICU channel number
`uint16_t`	Recent ICU channel counter value

5.21.10 Mft_PpgSwTrigger()

Triggers PPG channels of an MFT instance by Software.

Prototype	
`en_result_t Mft_PpgSwTrigger( stc_mftn_t* pstcMft, boolean_t bPpg0, boolean_t bPpg2, boolean_t bPpg4 )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
`[in] bPpg0`	`TRUE`: Triggers PPG0 `FALSE`: Preserve PPG0 operation state
`[in] bPpg2`	`TRUE`: Triggers PPG2 `FALSE`: Preserve PPG2 operation state
`[in] bPpg4`	`TRUE`: Triggers PPG4 `FALSE`: Preserve PPG4 operation state
Return Values	**Description**
`Ok`	Dedicated PPGs triggered.
`ErrorInvalidParameter`	• `pstcMft == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved) • Intance number of internal data out of range (corrupted)

5.21.11 Mft_PpgSwStop()

Stops PPG channels of an MFT instance by Software.

Prototype	
`en_result_t Mft_PpgSwStop( stc_mftn_t* pstcMft, boolean_t bPpg0, boolean_t bPpg2, boolean_t bPpg4 )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
`[in] bPpg0`	`TRUE`: Stop PPG0 `FALSE`: Preserve PPG0 operation state
`[in] bPpg2`	`TRUE`: Stop PPG2 `FALSE`: Preserve PPG2 operation state
`[in] bPpg4`	`TRUE`: Stop PPG4 `FALSE`: Preserve PPG4 operation state
Return Values	**Description**
`Ok`	Dedicated PPGs stopped.
`ErrorInvalidParameter`	• `pstcMft == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved) • Intance number of internal data out of range (corrupted)

5.21.12 Mft_PpgTimerStart()

Start all 3 PPG timer counters of an MFT instance by Software Start PPG0/2/4 (PPG8/10/12) timer synchronously.

Prototype	
`en_result_t Mft_PpgTimerStart( stc_mftn_t* pstcMft )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
Return Values	**Description**
`Ok`	All PPGs started.
`ErrorInvalidParameter`	• `pstcMft == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved) • Intance number of internal data out of range (corrupted)

5.21.13 Mft_PpgTimerStop()

Stop dedicated PPG timer counters of an MFT instance by Software Start synchronously. Stop PPG0/2/4 (PPG8/10/12) timer individually.

Prototype	
`en_result_t Mft_PpgTimerStop( stc_mftn_t* pstcMft, boolean_t bPpg0, boolean_t bPpg2, boolean_t bPpg4 )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
`[in] bPpg0`	`TRUE`: Stop PPG0 `FALSE`: Preserve PPG0 operation state
`[in] bPpg2`	`TRUE`: Stop PPG2 `FALSE`: Preserve PPG2 operation state
`[in] bPpg4`	`TRUE`: Stop PPG4 `FALSE`: Preserve PPG4 operation state
Return Values	**Description**
`Ok`	Dedicated PPGs stopped.
`ErrorInvalidParameter`	• `pstcMft == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved) • Intance number of internal data out of range (corrupted)

5.21.14 Mft_PpgSetValue8()

Set new 8-bit PPG values of an MFT instance. Use this function only, if PPG is set to 8-Bit mode.

Prototype	
`en_result_t Mft_PpgSetValue8( stc_mftn_t* pstcMft, uint8_t u8PpgNumber, uint8_t u8LowValue, uint8_t u8HighValue )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
`[in] u8PpgNumber`	`0, 2, 4` means PPG0/2/4
`[in] u8LowValue`	8-Bit PPG low pulse value
`[in] u8HighValue`	8-Bit PPG high pulse value
Return Values	**Description**
`Ok`	Dedicated PPGs set with new values.
`ErrorInvalidParameter`	• `pstcMft == NULL` • PPG number out of range or wrong

5.21.15 Mft_PpgSetValue16()

Set new 8-bit PPG values of an MFT instance. Use this function only, if PPG is set to 16-Bit mode.

Prototype	
`en_result_t Mft_PpgSetValue8( stc_mftn_t* pstcMft, uint8_t u8PpgNumber, uint16_t u16LowValue, uint16_t u16HighValue )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
`[in] u8PpgNumber`	`0, 2, 4` means PPG0/2/4
`[in] u16LowValue`	16-Bit PPG low pulse value
`[in] u16HighValue`	16-Bit PPG high pulse value
Return Values	**Description**
`Ok`	Dedicated PPGs set with new values.
`ErrorInvalidParameter`	• `pstcMft == NULL` • PPG number out of range or wrong

5.21.16 Mft_PpgSetValue8plus8()

Set new 8+8-bit PPG values of an MFT instance. Use this function only, if PPG is set to 8+8-Bit mode.

Prototype	
`en_result_t Mft_PpgSetValue8plus8( stc_mftn_t* pstcMft, uint8_t u8PpgNumber, uint8_t u8PscLowValue, uint8_t u8PscHighValue, uint8_t u8LowValue, uint8_t u8HighValue )`	
Parameter Name	**Description**
`[in] pstcMft`	Pointer to MFT instance
`[in] u8PpgNumber`	`0, 2, 4` means PPG0/2/4
`[in] u8PscLowValue`	8-Bit Prescaler low pulse value
`[in] u8PscHighValue`	8-Bit Prescaler high pulse value
`[in] u8LowValue`	8-Bit PPG low pulse value
`[in] u8HighValue`	8-Bit PPG high pulse value
Return Values	**Description**
`Ok`	Dedicated PPGs set with new values.
`ErrorInvalidParameter`	• `pstcMft == NULL` • PPG number out of range or wrong

5.21.17 Mft_PpgIgbt_Init()

Init PPG IGBT module. MFTn PPG must be initialized properly before calling this function! This function does not enable the IGBT functionality. `Mft_PpgIgbt_Enable()` is used for enabling this.

This function is only available for Device Type 7.

Prototype	
`en_result_t Mft_PpgIgbt_Init( stc_mft_ppg_igbt_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to IGBT configuration
Return Values	**Description**
`Ok`	IGBT successflully initialized
`ErrorInvalidParameter`	• `pstcConfig == NULL` • `pstcConfig->enNoiseFilter` undefined enumerator

5.21.18 Mft_PpgIgbt_Enable()

Enables PPG IGBT module.

This function is only available for Device Type 7.

Prototype	
`en_result_t Mft_PpgIgbt_Enable( void )`	
Return Value	**Description**
`Ok`	IGBT successflully enabled

5.21.19 Mft_PpgIgbt_Disable()

Disables PPG IGBT module.

This function is only available for Device Type 7.

Prototype	
`en_result_t Mft_PpgIgbt_Disable( void )`	
Return Value	**Description**
`Ok`	IGBT successflully disabled

5.21.20 Mft_PpgIgbt_DeInit()

De-initializes the PPG IGBT module.

This function is only available for Device Type 7.

Prototype	
`en_result_t Mft_PpgIgbt_DeInit( void )`	
Return Value	**Description**
`Ok`	IGBT successflully de-initialized

5.21.21 Frt*n*PeakCallback()

This function is called, if `stc_mft_frt_ocu_config_t::pfnFrtnPeakCallback` is defined, where *n* is `0`, `1`, or `2`.

Callback Function
`void FrtnPeakCallback( void )`

5.21.22 Frt*n*ZeroCallback()

This function is called, if `stc_mft_frt_ocu_config_t::pfnFrtnZeroCallback` is defined, where *n* is `0`, `1`, or `2`.

Callback Function
`void FrtnZeroCallback( void )`

5.21.23 Ocu*n*Callback()

This function is called, if `stc_mft_frt_ocu_config_t::pfnOcunCallback` is defined, where *n* is `0`, `1`, …, `4`, or `5`.

Callback Function
`void OcunCallback( void )`

5.21.24 Wfg*xy*Callback()

This function is called, if `stc_mft_wfg_config_t::pfnWfgxyCallback` is defined, where *xy* is `01`, `23`, or `45`.

Callback Function
`void WfgxyCallback( void )`

5.21.25 NoiseCancelCallback()

This function is called, if `stc_mft_nzcl_config_t::pfnNzclCallback` is defined.

Callback Function
`void NoiseCancelCallback( void )`

5.21.26 Icu*n*Callback()

This function is called, if `stc_mft_icu_config_t::pfnIcunCallback` is defined, where *n* is `0`, `1`, `2`, or `3`.

Callback Function
`void IcunCallback( void )`

5.21.27 Ppg*n*Callback()

This function is called, if `stc_mft_ppg_config_t::pfnPpgnCallback` is defined, where *n* is `0`, `2`, or `4`.

Callback Function
`void PpgnCallback( void )`

5.21.28 Example Codes

Because the Multifunction Timer has a very wide configuration complexity only some basic examples can be given here.

5.21.28.1 Basic FRT Operation with OCU Output

This code shows how to initialize the MFT for FRT-OCU-Output.

```
#include "mft.h"

function
{
  stc_mft_frt_ocu_config_t  stcMftFrtOcuConfig;  // MFT FRT/OCU config.
  stc_mftn_t*               pstcMft = NULL;      // MFT instance

  L3_ZERO_STRUCT(stcMftFrtOcuConfig);

  // FRT settings
  stcMftFrtOcuConfig.enClock0                     = MftPclk2;
  stcMftFrtOcuConfig.enClock1                     = MftPclk2;
  stcMftFrtOcuConfig.enClock2                     = MftPclk2;
  stcMftFrtOcuConfig.enMode0                      = MftUpDownCount;
  stcMftFrtOcuConfig.enMode1                      = MftUpDownCount;
  stcMftFrtOcuConfig.enMode2                      = MftUpDownCount;
  stcMftFrtOcuConfig.bTccp0BufferEnable           = TRUE;
  stcMftFrtOcuConfig.bTccp1BufferEnable           = TRUE;
  stcMftFrtOcuConfig.bTccp2BufferEnable           = TRUE;
  stcMftFrtOcuConfig.bTcsa0IclrInterruptEnable    = FALSE;
  stcMftFrtOcuConfig.bTcsa1IclrInterruptEnable    = FALSE;
  stcMftFrtOcuConfig.bTcsa2IclrInterruptEnable    = FALSE;
  stcMftFrtOcuConfig.bTcsa0IrqzfInterruptEnable = FALSE;
  stcMftFrtOcuConfig.bTcsa1IrqzfInterruptEnable = FALSE;
  stcMftFrtOcuConfig.bTcsa2IrqzfInterruptEnable = FALSE;
  stcMftFrtOcuConfig.bExternalClock0Enable        = FALSE;
  stcMftFrtOcuConfig.bExternalClock1Enable        = FALSE;
  stcMftFrtOcuConfig.bExternalClock2Enable        = FALSE;
  stcMftFrtOcuConfig.u16Frt0Cycle                 = 0x1234;
  stcMftFrtOcuConfig.u16Frt1Cycle                 = 0x1234;
  stcMftFrtOcuConfig.u16Frt2Cycle                 = 0x1234;
  stcMftFrtOcuConfig.enFrtOcu0Connection          = MftFrt0Ocu;
  stcMftFrtOcuConfig.enFrtOcu1Connection          = MftFrt0Ocu;
  stcMftFrtOcuConfig.enFrtOcu2Connection          = MftFrt1Ocu;
  stcMftFrtOcuConfig.enFrtOcu3Connection          = MftFrt1Ocu;
  stcMftFrtOcuConfig.enFrtOcu4Connection          = MftFrt2Ocu;
  stcMftFrtOcuConfig.enFrtOcu5Connection          = MftFrt2Ocu;

  // OCU settings
  stcMftFrtOcuConfig.bOcu0Enable                      = TRUE;
  stcMftFrtOcuConfig.bOcu1Enable                      = TRUE;
  stcMftFrtOcuConfig.bOcu2Enable                      = TRUE;
  stcMftFrtOcuConfig.bOcu3Enable                      = TRUE;
  stcMftFrtOcuConfig.bOcu4Enable                      = TRUE;
  stcMftFrtOcuConfig.bOcu5Enable                      = TRUE;
  stcMftFrtOcuConfig.bOccp0BufferEnable               = TRUE;
  stcMftFrtOcuConfig.bOccp1BufferEnable               = TRUE;
  stcMftFrtOcuConfig.bOccp2BufferEnable               = TRUE;
  stcMftFrtOcuConfig.bOccp3BufferEnable               = TRUE;
  stcMftFrtOcuConfig.bOccp4BufferEnable               = TRUE;
  stcMftFrtOcuConfig.bOccp5BufferEnable               = TRUE;
  stcMftFrtOcuConfig.bOcsaIop0InterruptEnable         = FALSE;
  stcMftFrtOcuConfig.bOcsaIop1InterruptEnable         = FALSE;
  stcMftFrtOcuConfig.bOcsaIop2InterruptEnable         = FALSE;
  stcMftFrtOcuConfig.bOcsaIop3InterruptEnable         = FALSE;
  stcMftFrtOcuConfig.bOcsaIop4InterruptEnable         = FALSE;
  stcMftFrtOcuConfig.bOcsaIop5InterruptEnable         = FALSE;
  stcMftFrtOcuConfig.bRt0HighLevel                    = FALSE;
  stcMftFrtOcuConfig.bRt1HighLevel                    = TRUE;
  stcMftFrtOcuConfig.bRt2HighLevel                    = FALSE;
  stcMftFrtOcuConfig.bRt3HighLevel                    = TRUE;
  stcMftFrtOcuConfig.bRt4HighLevel                    = FALSE;
```

▼

```
                                                                                          ▲

  stcMftFrtOcuConfig.bCmod01                        = TRUE;
  stcMftFrtOcuConfig.bCmod23                        = TRUE;
  stcMftFrtOcuConfig.bCmod45                        = TRUE;
  stcMftFrtOcuConfig.bOccp0BufTransferPeakDetection = TRUE;
  stcMftFrtOcuConfig.bOccp1BufTransferPeakDetection = TRUE;
  stcMftFrtOcuConfig.bOccp2BufTransferPeakDetection = TRUE;
  stcMftFrtOcuConfig.bOccp3BufTransferPeakDetection = TRUE;
  stcMftFrtOcuConfig.bOccp4BufTransferPeakDetection = TRUE;
  stcMftFrtOcuConfig.bOccp5BufTransferPeakDetection = TRUE;
  stcMftFrtOcuConfig.bMod0                          = TRUE;
  stcMftFrtOcuConfig.bMod1                          = TRUE;
  stcMftFrtOcuConfig.bMod2                          = TRUE;
  stcMftFrtOcuConfig.bMod3                          = TRUE;
  stcMftFrtOcuConfig.bMod4                          = TRUE;
  stcMftFrtOcuConfig.bMod5                          = TRUE;
  stcMftFrtOcuConfig.u16Ocu0CompareValue            = 0x0400;
  stcMftFrtOcuConfig.u16Ocu1CompareValue            = 0x0500;
  stcMftFrtOcuConfig.u16Ocu2CompareValue            = 0x0600;
  stcMftFrtOcuConfig.u16Ocu3CompareValue            = 0x0700;
  stcMftFrtOcuConfig.u16Ocu4CompareValue            = 0x0800;
  stcMftFrtOcuConfig.u16Ocu5CompareValue            = 0x0900;
  stcMftFrtOcuConfig.pfnFrt0PeakCallback            = NULL;
  stcMftFrtOcuConfig.pfnFrt1PeakCallback            = NULL;
  stcMftFrtOcuConfig.pfnFrt2PeakCallback            = NULL;
  stcMftFrtOcuConfig.pfnFrt0ZeroCallback            = NULL;
  stcMftFrtOcuConfig.pfnFrt1ZeroCallback            = NULL;
  stcMftFrtOcuConfig.pfnFrt2ZeroCallback            = NULL;
  stcMftFrtOcuConfig.pfnOcu0Callback                = NULL;
  stcMftFrtOcuConfig.pfnOcu1Callback                = NULL;
  stcMftFrtOcuConfig.pfnOcu2Callback                = NULL;
  stcMftFrtOcuConfig.pfnOcu3Callback                = NULL;
  stcMftFrtOcuConfig.pfnOcu4Callback                = NULL;
  stcMftFrtOcuConfig.pfnOcu5Callback                = NULL;

  if (Ok == Mft_Init((stc_mftn_t*)&MFT0,       // MFT instance
                     &stcMftFrtOcuConfig,      // FRT-OCU configuration
                     NULL,                     // WFG configuration
                     NULL,                     // NZCL configuration
                     NULL,                     // ICU configuration
                     NULL,                     // ADCMP configuration
                     NULL                      // PPG configuration
                    )
     )
  {
    pstcMft = (stc_mftn_t*)&MFT0;   // Setup pointer to MFT0 instance

    Mft_FrtStart(pstcMft,
                 TRUE,              // Start FRT0
                 TRUE,              // Start FRT1
                 TRUE);             // Start FRT2

    . . .

  }
}
```

5.21.28.2 FRT Operation with OCU Output, Dead Time Insertion, and PPG Modulation

This code shows how to initialize a WFG, OCU with WFG dead time insertion and PPG modulation. The RT0n ports show the same signals but with different PPG modulation and different WFG dead times.

```
#include "mft.h"

function
{
  stc_mft_frt_ocu_config_t  stcMftFrtOcuConfig; // MFT FRT/OCU conf.
  stc_mft_wfg_config_t      stcMftWfgConfig;    // MFT WFG configuration
  stc_mft_ppg_config_t      stcMftPpgConfig;    // MFT PPG configuration
  stc_mftn_t*               pstcMft = NULL;     // MFT instance

  L3_ZERO_STRUCT(stcMftFrtOcuConfig);
  L3_ZERO_STRUCT(stcMftWfgConfig);
  L3_ZERO_STRUCT(stcMftPpgConfig);

  // FRT settings
  stcMftFrtOcuConfig.enClock0                    = MftPclk2;
  stcMftFrtOcuConfig.enClock1                    = MftPclk2;
  stcMftFrtOcuConfig.enClock2                    = MftPclk2;
  stcMftFrtOcuConfig.enMode0                     = MftUpDownCount;
  stcMftFrtOcuConfig.enMode1                     = MftUpDownCount;
  stcMftFrtOcuConfig.enMode2                     = MftUpDownCount;
  stcMftFrtOcuConfig.bTccp0BufferEnable          = TRUE;
  stcMftFrtOcuConfig.bTccp1BufferEnable          = TRUE;
  stcMftFrtOcuConfig.bTccp2BufferEnable          = TRUE;
  stcMftFrtOcuConfig.bTcsa0IclrInterruptEnable   = FALSE;
  stcMftFrtOcuConfig.bTcsa1IclrInterruptEnable   = FALSE;
  stcMftFrtOcuConfig.bTcsa2IclrInterruptEnable   = FALSE;
  stcMftFrtOcuConfig.bTcsa0IrqzfInterruptEnable = FALSE;
  stcMftFrtOcuConfig.bTcsa1IrqzfInterruptEnable = FALSE;
  stcMftFrtOcuConfig.bTcsa2IrqzfInterruptEnable = FALSE;
  stcMftFrtOcuConfig.bExternalClock0Enable       = FALSE;
  stcMftFrtOcuConfig.bExternalClock1Enable       = FALSE;
  stcMftFrtOcuConfig.bExternalClock2Enable       = FALSE;
  stcMftFrtOcuConfig.u16Frt0Cycle                = 0x1234;
  stcMftFrtOcuConfig.u16Frt1Cycle                = 0x1234;
  stcMftFrtOcuConfig.u16Frt2Cycle                = 0x1234;
  stcMftFrtOcuConfig.enFrtOcu0Connection         = MftFrt0Ocu;
  stcMftFrtOcuConfig.enFrtOcu1Connection         = MftFrt0Ocu;
  stcMftFrtOcuConfig.enFrtOcu2Connection         = MftFrt1Ocu;
  stcMftFrtOcuConfig.enFrtOcu3Connection         = MftFrt1Ocu;
  stcMftFrtOcuConfig.enFrtOcu4Connection         = MftFrt2Ocu;
  stcMftFrtOcuConfig.enFrtOcu5Connection         = MftFrt2Ocu;

  // OCU settings
  stcMftFrtOcuConfig.bOcu0Enable                     = TRUE;
  stcMftFrtOcuConfig.bOcu1Enable                     = TRUE;
  stcMftFrtOcuConfig.bOcu2Enable                     = TRUE;
  stcMftFrtOcuConfig.bOcu3Enable                     = TRUE;
  stcMftFrtOcuConfig.bOcu4Enable                     = TRUE;
  stcMftFrtOcuConfig.bOcu5Enable                     = TRUE;
  stcMftFrtOcuConfig.bOccp0BufferEnable              = TRUE;
  stcMftFrtOcuConfig.bOccp1BufferEnable              = TRUE;
  stcMftFrtOcuConfig.bOccp2BufferEnable              = TRUE;
  stcMftFrtOcuConfig.bOccp3BufferEnable              = TRUE;
  stcMftFrtOcuConfig.bOccp4BufferEnable              = TRUE;
  stcMftFrtOcuConfig.bOccp5BufferEnable              = TRUE;
  stcMftFrtOcuConfig.bOcsaIop0InterruptEnable        = FALSE;
  stcMftFrtOcuConfig.bOcsaIop1InterruptEnable        = FALSE;
  stcMftFrtOcuConfig.bOcsaIop2InterruptEnable        = FALSE;
  stcMftFrtOcuConfig.bOcsaIop3InterruptEnable        = FALSE;
  stcMftFrtOcuConfig.bOcsaIop4InterruptEnable        = FALSE;
  stcMftFrtOcuConfig.bOcsaIop5InterruptEnable        = FALSE;
  stcMftFrtOcuConfig.bRt0HighLevel                   = FALSE;
  stcMftFrtOcuConfig.bRt1HighLevel                   = TRUE;

                                                                        ▼
```

```
    stcMftFrtOcuConfig.bRt2HighLevel                    = FALSE;
    stcMftFrtOcuConfig.bRt3HighLevel                    = TRUE;
    stcMftFrtOcuConfig.bRt4HighLevel                    = FALSE;
    stcMftFrtOcuConfig.bCmod01                          = TRUE;
    stcMftFrtOcuConfig.bCmod23                          = TRUE;
    stcMftFrtOcuConfig.bCmod45                          = TRUE;
    stcMftFrtOcuConfig.bOccp0BufTransferPeakDetection = TRUE;
    stcMftFrtOcuConfig.bOccp1BufTransferPeakDetection = TRUE;
    stcMftFrtOcuConfig.bOccp2BufTransferPeakDetection = TRUE;
    stcMftFrtOcuConfig.bOccp3BufTransferPeakDetection = TRUE;
    stcMftFrtOcuConfig.bOccp4BufTransferPeakDetection = TRUE;
    stcMftFrtOcuConfig.bOccp5BufTransferPeakDetection = TRUE;
    stcMftFrtOcuConfig.bMod0                            = TRUE;
    stcMftFrtOcuConfig.bMod1                            = TRUE;
    stcMftFrtOcuConfig.bMod2                            = TRUE;
    stcMftFrtOcuConfig.bMod3                            = TRUE;
    stcMftFrtOcuConfig.bMod4                            = TRUE;
    stcMftFrtOcuConfig.bMod5                            = TRUE;
    stcMftFrtOcuConfig.u16Ocu0CompareValue              = 0x0400;
    stcMftFrtOcuConfig.u16Ocu1CompareValue              = 0x0500;
    stcMftFrtOcuConfig.u16Ocu2CompareValue              = 0x0600;
    stcMftFrtOcuConfig.u16Ocu3CompareValue              = 0x0700;
    stcMftFrtOcuConfig.u16Ocu4CompareValue              = 0x0800;
    stcMftFrtOcuConfig.u16Ocu5CompareValue              = 0x0900;
    stcMftFrtOcuConfig.pfnFrt0PeakCallback              = NULL;
    stcMftFrtOcuConfig.pfnFrt1PeakCallback              = NULL;
    stcMftFrtOcuConfig.pfnFrt2PeakCallback              = NULL;
    stcMftFrtOcuConfig.pfnFrt0ZeroCallback              = NULL;
    stcMftFrtOcuConfig.pfnFrt1ZeroCallback              = NULL;
    stcMftFrtOcuConfig.pfnFrt2ZeroCallback              = NULL;
    stcMftFrtOcuConfig.pfnOcu0Callback                  = NULL;
    stcMftFrtOcuConfig.pfnOcu1Callback                  = NULL;
    stcMftFrtOcuConfig.pfnOcu2Callback                  = NULL;
    stcMftFrtOcuConfig.pfnOcu3Callback                  = NULL;
    stcMftFrtOcuConfig.pfnOcu4Callback                  = NULL;
    stcMftFrtOcuConfig.pfnOcu5Callback                  = NULL;

    // WFG settings
    stcMftWfgConfig.enWfgClock01                 = MftWfgPclk1;
    stcMftWfgConfig.enWfgClock23                 = MftWfgPclk1;
    stcMftWfgConfig.enWfgClock45                 = MftWfgPclk1;
    stcMftWfgConfig.enWfgMode01                  = MftWfgPpgDeadTimerMode;
    stcMftWfgConfig.enWfgMode23                  = MftWfgPpgDeadTimerMode;
    stcMftWfgConfig.enWfgMode45                  = MftWfgPpgDeadTimerMode;
    stcMftWfgConfig.enWfgGate01                  = MftWfgRt0Flag0;
    stcMftWfgConfig.enWfgGate23                  = MftWfgRt0Flag0;
    stcMftWfgConfig.enWfgGate45                  = MftWfgRt0Flag0;
    stcMftWfgConfig.enWfgPpgSel01                = MftWfgGatePpg0;
    stcMftWfgConfig.enWfgPpgSel23                = MftWfgGatePpg2;
    stcMftWfgConfig.enWfgPpgSel45                = MftWfgGatePpg4;
    stcMftWfgConfig.enWfgPpgEnable01             = MftWfgPpgPpg;
    stcMftWfgConfig.enWfgPpgEnable23             = MftWfgPpgPpg;
    stcMftWfgConfig.enWfgPpgEnable45             = MftWfgPpgPpg;
    stcMftWfgConfig.bNonOverlapHighPolarity0 = FALSE;
    stcMftWfgConfig.bNonOverlapHighPolarity1 = FALSE;
    stcMftWfgConfig.bNonOverlapHighPolarity2 = FALSE;
    stcMftWfgConfig.u16Wfg01TimerValue           = 0x40;
    stcMftWfgConfig.u16Wfg23TimerValue           = 0x80;
    stcMftWfgConfig.u16Wfg45TimerValue           = 0xC0;
```

```
                                                                        ▲

  stcMftWfgConfig.bWfgInterruptEnable      = FALSE;
  stcMftWfgConfig.pfnWfg01Callback         = NULL;
  stcMftWfgConfig.pfnWfg23Callback         = NULL;
  stcMftWfgConfig.pfnWfg45Callback         = NULL;

  // PPG settings
  stcMftPpgConfig.enPpg0Mode           = MftPpg8Bit;
  stcMftPpgConfig.enPpg2Mode           = MftPpg8Bit;
  stcMftPpgConfig.enPpg4Mode           = MftPpg8Bit;
  stcMftPpgConfig.enPpg0Clock          = MftPpgPclk8;
  stcMftPpgConfig.enPpg2Clock          = MftPpgPclk8;
  stcMftPpgConfig.enPpg4Clock          = MftPpgPclk8;
  stcMftPpgConfig.bPpg0InvertPolarity  = FALSE;
  stcMftPpgConfig.bPpg2InvertPolarity  = FALSE;
  stcMftPpgConfig.bPpg4InvertPolarity  = FALSE;
  stcMftPpgConfig.bPpg0InterruptEnable = FALSE;
  stcMftPpgConfig.bPpg2InterruptEnable = FALSE;
  stcMftPpgConfig.bPpg4InterruptEnable = FALSE;
  stcMftPpgConfig.bPpg0InterruptMode   = FALSE;
  stcMftPpgConfig.bPpg2InterruptMode   = FALSE;
  stcMftPpgConfig.bPpg4InterruptMode   = FALSE;
  stcMftPpgConfig.u8Ppg0Low            = 0x20;
  stcMftPpgConfig.u8Ppg0High           = 0x80;
  stcMftPpgConfig.u8Ppg2Low            = 0x30;
  stcMftPpgConfig.u8Ppg2High           = 0x90;
  stcMftPpgConfig.u8Ppg4Low            = 0x40;
  stcMftPpgConfig.u8Ppg4High           = 0xA0;
  stcMftPpgConfig.enPpg0Trigger        = MftPpgGateSignal;
  stcMftPpgConfig.enPpg2Trigger        = MftPpgGateSignal;
  stcMftPpgConfig.enPpg4Trigger        = MftPpgGateSignal;
  stcMftPpgConfig.bPpg0StartLowLevel   = FALSE;
  stcMftPpgConfig.bPpg2StartLowLevel   = FALSE;
  stcMftPpgConfig.bPpg4StartLowLevel   = FALSE;
  stcMftPpgConfig.enPpgCompareClock    = MftPpgPclk2;
  stcMftPpgConfig.u8Ppg0Compare        = 0;
  stcMftPpgConfig.u8Ppg2Compare        = 0;
  stcMftPpgConfig.u8Ppg4Compare        = 0;
  stcMftPpgConfig.pfnPpg0Callback      = NULL;
  stcMftPpgConfig.pfnPpg2Callback      = NULL;
  stcMftPpgConfig.pfnPpg4Callback      = NULL;

  if (Ok == Mft_Init((stc_mftn_t*)&MFT0,       // MFT instance
                     &stcMftFrtOcuConfig,      // FRT-OCU configuration
                     &stcMftWfgConfig,         // WFG configuration
                     NULL,                     // NZCL configuration
                     NULL,                     // ICU configuration
                     NULL,                     // ADCMP configuration
                     &stcMftPpgConfig          // PPG configuration
                    )
     )
  {
    pstcMft = (stc_mftn_t*)&MFT0;   // Setup pointer to MFT0 instance

    Mft_FrtStart(pstcMft,
                 TRUE,              // Start FRT0
                 TRUE,              // Start FRT1
                 TRUE);             // Start FRT2

    . . .
  }
}
```

5.21.28.3 PPG only Operation

This code shows how to initialize the MFT for PPG only operation and software start trigger.

```
#include "mft.h"

function
{
  stc_mft_frt_ocu_config_t  stcMftFrtOcuConfig;  // MFT FRT/OCU config.
  stc_mft_wfg_config_t      stcMftWfgConfig;     // MFT WFG configuration
  stc_mft_ppg_config_t      stcMftPpgConfig;     // MFT PPG configuration
  stc_mftn_t*               pstcMft = NULL;      // MFT instance

  L3_ZERO_STRUCT(stcMftFrtOcuConfig);
  L3_ZERO_STRUCT(stcMftWfgConfig);
  L3_ZERO_STRUCT(stcMftPpgConfig);

  // FRT outputs to high level for "anding" with PPG signals
  stcMftFrtOcuConfig.bRt0HighLevel = TRUE;
  stcMftFrtOcuConfig.bRt1HighLevel = TRUE;  // same as RT0
  stcMftFrtOcuConfig.bRt2HighLevel = TRUE;
  stcMftFrtOcuConfig.bRt3HighLevel = TRUE;  // same as RT2
  stcMftFrtOcuConfig.bRt4HighLevel = TRUE;
  stcMftFrtOcuConfig.bRt5HighLevel = TRUE;  // same as RT4

  // WFG settings
  stcMftWfgConfig.enWfgClock01              = MftWfgPclk1;
  stcMftWfgConfig.enWfgClock23              = MftWfgPclk1;
  stcMftWfgConfig.enWfgClock45              = MftWfgPclk1;
  stcMftWfgConfig.enWfgMode01               = MftWfgRtPpgMode;
  stcMftWfgConfig.enWfgMode23               = MftWfgRtPpgMode;
  stcMftWfgConfig.enWfgMode45               = MftWfgRtPpgMode;
  stcMftWfgConfig.enWfgGate01               = MftWfgRt10Flag10;
  stcMftWfgConfig.enWfgGate23               = MftWfgRt10Flag10;
  stcMftWfgConfig.enWfgGate45               = MftWfgRt10Flag10;
  stcMftWfgConfig.enWfgPpgSel01             = MftWfgGatePpg0;
  stcMftWfgConfig.enWfgPpgSel23             = MftWfgGatePpg2;
  stcMftWfgConfig.enWfgPpgSel45             = MftWfgGatePpg4;
  stcMftWfgConfig.enWfgPpgEnable01          = MftWfgPpgPpg;
  stcMftWfgConfig.enWfgPpgEnable23          = MftWfgPpgPpg;
  stcMftWfgConfig.enWfgPpgEnable45          = MftWfgPpgPpg;
  stcMftWfgConfig.bNonOverlapHighPolarity0 = FALSE;
  stcMftWfgConfig.bNonOverlapHighPolarity1 = FALSE;
  stcMftWfgConfig.bNonOverlapHighPolarity2 = FALSE;
  stcMftWfgConfig.u16Wfg01TimerValue        = 0;
  stcMftWfgConfig.u16Wfg23TimerValue        = 0;
  stcMftWfgConfig.u16Wfg45TimerValue        = 0;
  stcMftWfgConfig.bWfgInterruptEnable       = FALSE;
  stcMftWfgConfig.pfnWfg01Callback          = NULL;
  stcMftWfgConfig.pfnWfg23Callback          = NULL;
  stcMftWfgConfig.pfnWfg45Callback          = NULL;

  // PPG settings
  stcMftPpgConfig.enPpg0Mode            = MftPpg8Bit;
  stcMftPpgConfig.enPpg2Mode            = MftPpg8Bit;
  stcMftPpgConfig.enPpg4Mode            = MftPpg8Bit;
  stcMftPpgConfig.enPpg0Clock           = MftPpgPclk8;
  stcMftPpgConfig.enPpg2Clock           = MftPpgPclk8;
  stcMftPpgConfig.enPpg4Clock           = MftPpgPclk8;
  stcMftPpgConfig.bPpg0InvertPolarity   = FALSE;
  stcMftPpgConfig.bPpg2InvertPolarity   = FALSE;
  stcMftPpgConfig.bPpg4InvertPolarity   = FALSE;
  stcMftPpgConfig.bPpg0InterruptEnable = FALSE;
  stcMftPpgConfig.bPpg2InterruptEnable = FALSE;
  stcMftPpgConfig.bPpg4InterruptEnable = FALSE;
  stcMftPpgConfig.bPpg0InterruptMode   = FALSE;
  stcMftPpgConfig.bPpg2InterruptMode   = FALSE;
  stcMftPpgConfig.bPpg4InterruptMode   = FALSE;
```

▼

▲

```
  stcMftPpgConfig.u8Ppg0Low             = 0x20;
  stcMftPpgConfig.u8Ppg0High            = 0x80;
  stcMftPpgConfig.u8Ppg2Low             = 0x30;
  stcMftPpgConfig.u8Ppg2High            = 0x90;
  stcMftPpgConfig.u8Ppg4Low             = 0x40;
  stcMftPpgConfig.u8Ppg4High            = 0xA0;
  stcMftPpgConfig.enPpg0Trigger         = MftPpgTriggerRegister;
  stcMftPpgConfig.enPpg2Trigger         = MftPpgTriggerRegister;
  stcMftPpgConfig.enPpg4Trigger         = MftPpgTriggerRegister;
  stcMftPpgConfig.bPpg0StartLowLevel    = FALSE;
  stcMftPpgConfig.bPpg2StartLowLevel    = FALSE;
  stcMftPpgConfig.bPpg4StartLowLevel    = FALSE;
  stcMftPpgConfig.enPpgCompareClock     = MftPpgPclk2;
  stcMftPpgConfig.u8Ppg0Compare         = 0;
  stcMftPpgConfig.u8Ppg2Compare         = 0;
  stcMftPpgConfig.u8Ppg4Compare         = 0;
  stcMftPpgConfig.pfnPpg0Callback       = NULL;
  stcMftPpgConfig.pfnPpg2Callback       = NULL;
  stcMftPpgConfig.pfnPpg4Callback       = NULL;

  if (Ok == Mft_Init((stc_mftn_t*)&MFT0,      // MFT instance
                     &stcMftFrtOcuConfig,      // FRT-OCU configuration
                     &stcMftWfgConfig,         // WFG configuration
                     NULL,                     // NZCL configuration
                     NULL,                     // ICU configuration
                     NULL,                     // ADCMP configuration
                     &stcMftPpgConfig          // PPG configuration
                    )
      )
  {
    pstcMft = (stc_mftn_t*)&MFT0;   // Setup pointer to MFT0 instance

    Mft_FrtStart(pstcMft,
                 TRUE,              // Start FRT0
                 TRUE,              // Start FRT1
                 TRUE);             // Start FRT2

    // Start PPG0, PPG2, PPG4
    Mft_PpgSwTrigger(pstcMft, TRUE, TRUE, TRUE);

    // Stop PPGs consecutively after some time ...
    // Some wait delay here ...
    Mft_PpgSwStop(pstcMft, FALSE, FALSE, TRUE); // Stop PPG4

    // Some wait delay here ...
    Mft_PpgSwStop(pstcMft, FALSE, TRUE, FALSE); // Stop PPG2

    // Some wait delay here ...
    Mft_PpgSwStop(pstcMft, TRUE, FALSE, FALSE); // Stop PPG0

    . . .

  }
}
```

5.22 (QPRC) Quad Position and Revolution Counter

Type Definition	`stc_qprcn_t`
Configuration Type	`stc_qprc_config_t`
Address Operator	`QPRC`*n*

Init a QPRC instance with `Qprc_Init()`. Here the callback function pointers for each interrupt cause can be set in the `stc_qprc_config_t` configuration. The recent position count can be read after `Qprc_Init()` at any time by `Qprc_GetPositionCount()`. The same is valid for the revolution count and `Qprc_GetRevolutionCount()`.

After initialization the QPRC does not count immediately. Use `Qprc_Enable()` to enable the initialized instance and `Qprc_Disable()` to disable it temporarily.

To change the position and revolution count `Qprc_SetPositionCount()` and `Qprc_SetRevolutionCount()` can be used. Note that these functions temporarily disable the counter function for some CPU cycles. `Qprc_SetRevolutionCountCompare()`, `Qprc_SetPositionRevolutionCountCompare()`, and `Qprc_SetMaxPositionCount()` can set new values after initialization, if needed.

The functions `Qprc_GetRevolutionCountCompare()`, `Qprc_GetPositionRevolutionCountCompare()`, and `Qprc_GetMaxPositionCount()` return the values of the corresponding registers.

`Qprc_DeInit()` should be called, if the QPRC instance is no longer needed at runtime, or a new configuration should be initialized afterwards.

`Qprc_GetPositionRevolutionCount()` can only be used on ≥ Device Type 3. It reads out the Position and Revolution Count simultaneously and stores their values into a pointer structure.

Note:

The QPRC shares the interrupt vector with the Dual Timer. Therefore `stc_qprc_config_t::bTouchNVIC` determines whether to touch the NVIC registers or not. Take care, that in the 2nd case, if the Dual Timer is used, the NVIC initialization and de-initialization must be handled by the Dual Timer driver.

5.22.1 Configuration Structure

The QPRC module uses the following channel configuration structure of the type `stc_qprc_config_t`:

Type	Field	Possible Values	Description

`en_qprc_…`
`pcmode_t`
	`enPcMode`	`QprcPcMode0`
		`QprcPcMode1`
		`QprcPcMode2`
		`QprcPcMode3`
			Disable position counter
			Increm. AIN, decrem. BIN at active edges
			Phase different count mode
			Directional count mode

`en_qprc_…`
`rcmode_t`
	`enRcMode`	`QprcRcMode0`
		`QprcRcMode1`
		`QprcRcMode2`
		`QprcRcMode3`
			Disable revolution counter
			Up/Down count with ZIN active edge
			Up/Down count on over/underflow of PC
			Up/Down on over/underf. of PC and ZIN

`boolean_t`	`bSwapAin…` `Bin`	`TRUE` `FALSE`	Swap AIN and BIN inputs AIN and BIN inputs as there are
`boolean_t`	`bRegister…`	`TRUE`	Compare PC

	Function	FALSE	Compare RC
boolean_t	bGate… Function	TRUE FALSE	Count gate function Counter clear function
en_qprc_… compmode_t	enComapre… Mode	QprcComapre… WithPosition QprcComapre… With… Revolution	Compares the value of the QPRC RC and RC Compare Register (QPRCR) with that of the Position Counter. Compares the value of the QPRC RC and RC Compare Register (QPRCR) with that of the Revolution Counter.
en_qprc_… zinedge_t	enZinEdge	QprcZin… …Disable …FallingEdge …RisingEdge …BothEdges …LowLevel …HighLevel	ZIN functionality: Disable 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 detection ZIN active at high level detection
en_qprc_… binedge_t	enBinEdge	QprcBin… …Disable …FallingEdge …RisingEdge …BothEdges	BIN functionality: Disable edge and level detection BIN active at falling edge BIN active at rising edge BIN active at falling or rising edge
en_qprc_… binedge_t	enAinEdge	QprcAin… …Disable …FallingEdge …RisingEdge …BothEdges	AIN functionality: Disable edge and level detection AIN active at falling edge AIN active at rising edge AIN active at falling or rising edge
en_qprc_… zinmode_t	enZinMode	QprcZin… CounterClear QprcZinGate	ZIN: Counter clear function ZIN: Gate function
en_qprc_… pcreset… mask_t	enPcReset… Mask	QprcReset… …MaskDisable …Mask2Times …Mask4Times …Mask8Times	Reset Mask: No mask reset PC reset, RC U/C count masked until 2 PC changes PC reset, RC U/C count masked until 4 PC changes PC reset, RC U/C count masked until 8 PC changes
boolean_t	b8KValue	TRUE FALSE	Outrange mode from 0 to 0x7FFF Outrange mode from 0 to 0xFFFF
uint16_t	u16… Position… Count	-	Position Count register update, if QCR.PSTP == 1; not updated, QCR.PSTP == 0
uint16_t	u16… Revolution… Count	-	Revolution Count register update, if QCR.RCM == 2'b00
uint16_t	u16… Position… Counter… Compare	-	Position Counter Compare register value
uint16_t	u16… Position… Revolution… Counter… Compare	-	Position and Revolution Counter Compare register value

uint16_t	u16Maximum… Position… Count	–	Maximum position count register (increases revolution, if stepped beyond)
boolean_t	bEnable… Outrange… Interrupt	TRUE FALSE	QPRC outrange interrupt enabled IRQ disabled
boolean_t	bEnable… OvrfUndfZ… Idx… Interrupt	TRUE FALSE	Over-, underfl., or zero index IRQ enabled IRQ disabled
boolean_t	bEnable… RcMatch… Interrupt	TRUE FALSE	QPRC RC match IRQ enabled IRQ disabled
boolean_t	bEnablePc… MatchRc… Match… Interupt	TRUE FALSE	QPRC PC and RC match IRQ enabled IRQ disabled
boolean_t	bEnable… PcMatch… Interrupt	TRUE FALSE	QPRC PC match IRQ enabled IRQ disabled
boolean_t	bEnable… Count… Inversion… Interrupt	TRUE FALSE	Count inversion IRQ enabled IRQ disabled
func_ptr_t	pfnCount… Inversion… Callback	–	Pointer to count inversion interrupt callback function
func_ptr_t	pfnZero… Index… Callback	–	Pointer to zero inted interrupt callback function
func_ptr_t	pfn… Overflow… Callback	–	Pointer to overflow interrupt callback function
func_ptr_t	pfn… Underflow… Callback	–	Pointer to underflow interrupt callback function
func_ptr_t	pfnRc… Match… Callback	–	Pointer to RC match interrupt callback function
func_ptr_t	pfnPc… Match… Callback	–	Pointer to PC match interrupt callback function
func_ptr_t	pfnPcMatch… RcMatch… Callback	–	Pointer to PC match and RC match interrupt callback function
func_ptr_t	pfn… Outrange… Callback	–	Pointer to outrange interrupt callback function
boolean_t	bTouchNVIC	TRUE FALSE	Init NVIC registers Don’t touch NVIC registers

5.22.2 Qprc_init()

This functions initializes an QPRC instance. Depending on `sct_qprc_config_t::bTouchNVIC == TRUE` it sets also the NVIC registers.

Prototype	
`en_result_t Qprc_Init( stc_qprcn_t* pstcQprc, stc_qprc_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcQprc`	QPRC instance pointer
`[in] pstcConfig`	Pointer to QPRC configuration
Return Values	**Description**
`Ok`	Internal data has been setup
`ErrorInvalidParameter`	• `pstcQprc == NULL` • `pstcConfig == NULL` • `pstcQprcInternData == NULL` (Instance could not be resolved) • Other invalid settings or enumerator used

5.22.3 Qprc_Deinit()

This functions de-initializes an QPRC instance. Depending on `sct_qprc_config_t::bTouchNVIC == TRUE` it sets also the NVIC registers.

Prototype	
`en_result_t Qprc_Init( stc_qprcn_t* pstcQprc, stc_qprc_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcQprc`	QPRC instance pointer
`[in] pstcConfig`	Pointer to QPRC configuration
Return Values	**Description**
`Ok`	QPRC is de-initialized
`ErrorInvalidParameter`	• `pstcQprc == NULL` • `pstcConfig == NULL` • `pstcQprcInternData == NULL` (Instance could not be resolved) • Other invalid settings or enumerator used

5.22.4 Qprc_Enable()

This functions enables QPRC counting.

Prototype	
`void Qprc_Enable( stc_qprcn_t* pstcQprc )`	
Parameter Name	**Description**
`[in] pstcQprc`	QPRC instance pointer

5.22.5 Qprc_Disable()

This functions disables QPRC counting.

Prototype	
void Qprc_Disable(stc_qprcn_t* pstcQprc)	
Parameter Name	**Description**
[in] pstcQprc	QPRC instance pointer

5.22.6 Qprc_SetPositionCount()

Sets a new Position Count value.

Prototype	
void Qprc_SetPositionCount(stc_qprcn_t* pstcQprc, uint16_t u16PcValue)	
Parameter Name	**Description**
[in] pstcQprc	QPRC instance pointer
[in] u16PcValue	New position count value

5.22.7 Qprc_SetRevolutionCount()

Sets a new Revolution Count value.

Prototype	
void Qprc_SetRevolutionCount(stc_qprcn_t* pstcQprc, uint16_t u16RcValue)	
Parameter Name	**Description**
[in] pstcQprc	QPRC instance pointer
[in] u16RcValue	New revolution count value

5.22.8 Qprc_SetRevolutionCountCompare()

Sets a new Revolution Count Compare value.

Prototype	
void Qprc_SetRevolutionCountCompare(stc_qprcn_t* pstcQprc, uint16_t u16RcCompValue)	
Parameter Name	**Description**
[in] pstcQprc	QPRC instance pointer
[in] u16RcCompValue	New revolution count compare value

5.22.9 Qprc_SetPositionRevolutionCountCompare()

Sets a new Position and Revolution Count Compare value.

Prototype	
void Qprc_SetPositionRevolutionCountCompare(stc_qprcn_t* pstcQprc, uint16_t u16PcRcCompValue)	
Parameter Name	**Description**
[in] pstcQprc	QPRC instance pointer
[in] u16PcRcCompValue	New position and revolution count compare value

5.22.10 Qprc_SetMaxPositionCount()

Sets a new Maximum Position Counter Compare value.

Prototype	
`void Qprc_SetMaxPositionCount( stc_qprcn_t* pstcQprc, uint16_t u16PcMaxValue )`	
Parameter Name	**Description**
`[in] pstcQprc`	QPRC instance pointer
`[in] u16PcMaxValue`	New maximum position count value

5.22.11 Qprc_GetPositionCount()

Returns recent Position Count value.

Prototype	
`uint16_t Qprc_GetPositionCount( stc_qprcn_t* pstcQprc )`	
Parameter Name	**Description**
`[in] pstcQprc`	QPRC instance pointer
Return Value	**Description**
`uint16_t`	Recent position count value

5.22.12 Qprc_GetRevolutionCount()

Returns recent Revolution Count value.

Prototype	
`uint16_t Qprc_GetRevolutionCount( stc_qprcn_t* pstcQprc )`	
Parameter Name	**Description**
`[in] pstcQprc`	QPRC instance pointer
Return Value	**Description**
`uint16_t`	Recent revolution count value

5.22.13 Qprc_GetRevolutionCountCompare()

Returns recent Revolution Count Compare value.

Prototype	
`uint16_t Qprc_GetRevolutionCountCompare( stc_qprcn_t* pstcQprc )`	
Parameter Name	**Description**
`[in] pstcQprc`	QPRC instance pointer
Return Value	**Description**
`uint16_t`	Recent revolution count compare value

5.22.14 Qprc_GetPositionRevolutionCountCompare()

Returns recent Position and Revolution Count Compare value.

Prototype	
`uint16_t Qprc_GetPositionRevolutionCountCompare( stc_qprcn_t* pstcQprc )`	
Parameter Name	**Description**
`[in] pstcQprc`	QPRC instance pointer
Return Value	**Description**
`uint16_t`	Recent position and revolution count compare value

5.22.15 Qprc_GetMaxPositionCount()

Returns recent Maximum Position and Revolution Count value.

Prototype	
`uint16_t Qprc_GetMaxPositionCount( stc_qprcn_t* pstcQprc )`	
Parameter Name	**Description**
`[in] pstcQprc`	QPRC instance pointer
Return Value	**Description**
`uint16_t`	Recent Maximum Position and Revolution Count value

5.22.16 Qprc_GetPositionrevolutionCount()

Device Types ≥ 3 allow to read the Position and Revolution count at once (32-Bit access) can use this function. The result is stored in a structure of the type of `stc_qprc_pos_rev_count_t` as follows:

Structure Type		
`stc_qprc_pos_rev_count_t`		
Type	**Field**	**Description**
`uint16_t`	`u16PositionCount`	Recent Position Count
`uint16_t`	`u16RevolutionCount`	Recent Revolution Count

Prototype	
`void Qprc_GetPositionRevolutionCount( stc_qprcn_t* pstcQprc, stc_qprc_pos_rev_count_t* pstcPosRevCount )`	
Parameter Name	**Description**
`[in] pstcQprc`	QPRC instance pointer
`[out] pstcPosRevCount`	Pointer to recent Position and Revolution Counter values return structure as described above

5.22.17 CountInversionCallback()

This function is called, if `stc_qprc_config_t::pfnCountInversionCallback` is defined.

Callback Function
`void CountInversionCallback( void )`

5.22.18 ZeroIndexCallback()

This function is called, if `stc_qprc_config_t::pfnZeroIndexCallback` is defined..

Callback Function
`void ZeroIndexCallback( void )`

5.22.19 OverflowCallback()

This function is called, if `stc_qprc_config_t::pfnOverflowCallback` is defined..

Callback Function
`void OverflowCallback( void )`

5.22.20 UndeflowCallback()

This function is called, if `stc_qprc_config_t::pfnUnderflowCallback` is defined..

Callback Function
`void UnderflowCallback( void )`

5.22.21 RcMatchCallback()

This function is called, if `stc_qprc_config_t::pfnRcMatchCallback` is defined..

Callback Function
`void RcMatchCallback( void )`

5.22.22 PcMatchCallback()

This function is called, if `stc_qprc_config_t::pfnPcMatchCallback` is defined..

Callback Function
`void PcMatchCallback( void )`

5.22.23 PcMatchRcMatchCallback()

This function is called, if `stc_qprc_config_t::pfnPcMatchRcMatchCallback` is defined..

Callback Function
`void PcMatchRcMatchCallback( void )`

5.22.24 OutrangeCallback()

This function is called, if `stc_qprc_config_t::pfnOutrangeCallback` is defined..

Callback Function
`void OutrangeCallback( void )`

5.22.25 Example Code

The following example shows how to use the QPRC driver. The Position counter is set to Mode 3 and the Revolution Counter to 2, which means BIN acts as clock input and AIN as direction input. ZIN is unused. It counts the position count as one's digit up and down between 0 and 9. At any underflow the counter is set to 9 and at any overflow to 0. The revolution count is the ten's digit between 0 and 9.

```
#include "qprc.h"

stc_qprcn_t* pstcQprc == NULL ; // Globally defined because of callback usage

void OverflowCallback(void)
{
  if (Qprc_GetRevolutionCount(pstcQprc) == 10)
  {
    Qprc_SetRevolutionCount(pstcQprc, 0);
  }
}

void UnderflowCallback(void)
{
  if (Qprc_GetRevolutionCount(pstcQprc) == 0xFFFF)
  {
    Qprc_SetRevolutionCount(pstcQprc, 9);
  }
}

function
{
  stc_mft_qprc_config_t  stcQprcConfig;     // QPRC configuration
  uint8_t u8PosCount;
  uint8_t u8RevCount;

  L3_ZERO_STRUCT(stcQprcConfig);

  stcQprcConfig.enPcMode           = QprcPcMode3;
  stcQprcConfig.enRcMode           = QprcRcMode2;
  stcQprcConfig.bSwapAinBin        = FALSE;
  stcQprcConfig.bRegisterFunction = TRUE;
  stcQprcConfig.bGateFunction = FALSE;
  stcQprcConfig.enComapreMode = QprcComapreWithPosition;
  stcQprcConfig.enZinEdge     = QprcZinDisable;
  stcQprcConfig.enBinEdge     = QprcBinFallingEdge;
  stcQprcConfig.enAinEdge     = QprcAinFallingEdge;
  stcQprcConfig.enZinMode     = QprcZinCounterClear;  // switched off anyhow
  stcQprcConfig.enPcResetMask = QprcResetMaskDisable;
  stcQprcConfig.b8KValue                             = FALSE;
  stcQprcConfig.u16PositionCount                     = 1;
  stcQprcConfig.u16RevolutionCount                   = 0;
  stcQprcConfig.u16PositionCounterCompare            = 4;
  stcQprcConfig.u16PositionRevolutionCounterCompare = 5;
  stcQprcConfig.u16MaximumPositionCount              = 9;
  stcQprcConfig.bEnableOutrangeInterrupt       = TRUE;
  stcQprcConfig.bEnableOvrfUndfZIdxInterrupt   = TRUE;
  stcQprcConfig.bEnableRcMatchInterrupt        = TRUE;
  stcQprcConfig.bEnablePcMatchRcMatchInterupt  = TRUE;
  stcQprcConfig.bEnablePcMatchInterrupt        = TRUE;
  stcQprcConfig.bEnableCountInversionInterrupt = TRUE;
  stcQprcConfig.pfnCountInversionCallback = NULL;
  stcQprcConfig.pfnZeroIndexCallback      = NULL;
  stcQprcConfig.pfnOverflowCallback       = &OverflowCallback;
  stcQprcConfig.pfnUnderflowCallback      = &UnderflowCallback;
  stcQprcConfig.pfnRcMatchCallback        = NULL;
  stcQprcConfig.pfnPcMatchCallback        = NULL;
  stcQprcConfig.pfnPcMatchRcMatchCallback = NULL;
  stcQprcConfig.pfnOutrangeCallback       = NULL;
  stcQprcConfig.bTouchNVIC                = TRUE;  // Take care of DT!

  if (Ok == Qprc_Init((stc_qprcn_t*)&QPRC0, &stcQprcConfig))
  {
```

▼

▲

```
    pstcQprc = (stc_qprcn_t*)&QPRC0;

    Qprc_Enable(pstcQprc);

    while(1)
    {
      u8PosCount = Qprc_GetPositionCount(pstcQprc);
      u8RevCount = Qprc_GetRevolutionCount(pstcQprc);

      // Display u8PosCount as one's digit
      // Display u8RevCount as ten's digit
    }
  }
}
```

5.23 (RAMCODE) RAM Code

Type Definition	–
Configuration Type	–
Address Operator	–

This driver module does not support any API function for the user.

Nevertheless it is used by the FLASH driver module and thus contains the low-level Flash erase and programming routines. Because this routines use polling routines a similar hook function like in *l3.c* is called: `static void L3_RAMCODE_WAIT_LOOP_HOOK(void)`

```
/**
 ******************************************************************************
 ** \brief Hook function, which is called from polling loops
 **
 ** This functionality should be the same as in l3.c!
 ******************************************************************************/
#ifdef __ICCARM__
  __ramfunc
#elif  __CC_ARM
  __attribute__ ((section (".ramfunc")))
#else
  #error Please check compiler and linker settings for RAM code
#endif
static void L3_RAMCODE_WAIT_LOOP_HOOK(void)
{
    // Place code for triggering Watchdog counters here, if needed
}
```

This function may be filled with user code, if needed. It should represent the same functionality as described in chapter 4.2.

5.23.1 NMI Handling

Because NMI is by definition non-maskable, it may also occur, if RAMCODE is executed. In this case the switch-off Flash-Memory-Read would cause an bus error, because the NMI vector can not be fetched.

Therefore, the Flash erase and programming routines disable all other interrupts, patch the interrupt vector table to RAM location, and set at the correct offset a RAM vector for the NMI. This vector points to a temporary RAM NMI service routine:

```
/**
 ******************************************************************************
 ** \brief NMI RAMCODE handler
 **
 ******************************************************************************/
#ifdef __ICCARM__
  __ramfunc
#elif  __CC_ARM
  __attribute__ ((section (".ramfunc")))
#else
  #error Please check compiler and linker settings for RAM code
#endif
void NMI_Ramcode_Handler(void)
{
  Ramcode_bNmiOccurred = TRUE;  // Set NMI notification for caller

  FM3_EXTI->NMICL = 0;          // Clear NMI

  Ramcode_FlashReadReset();     // Stop all possible started Flash commands
} // NMI Ramcode Handler
```

The variable Ramcode_bNmiOccurred notifies the Flash (ROM) routines, that during RAM execution an NMI occurred. These Flash routines then call the user's NMI callback function.

The following flow diagrams illustrate the mechanisms. The ROM execution NMI and its callback handling is shown below:

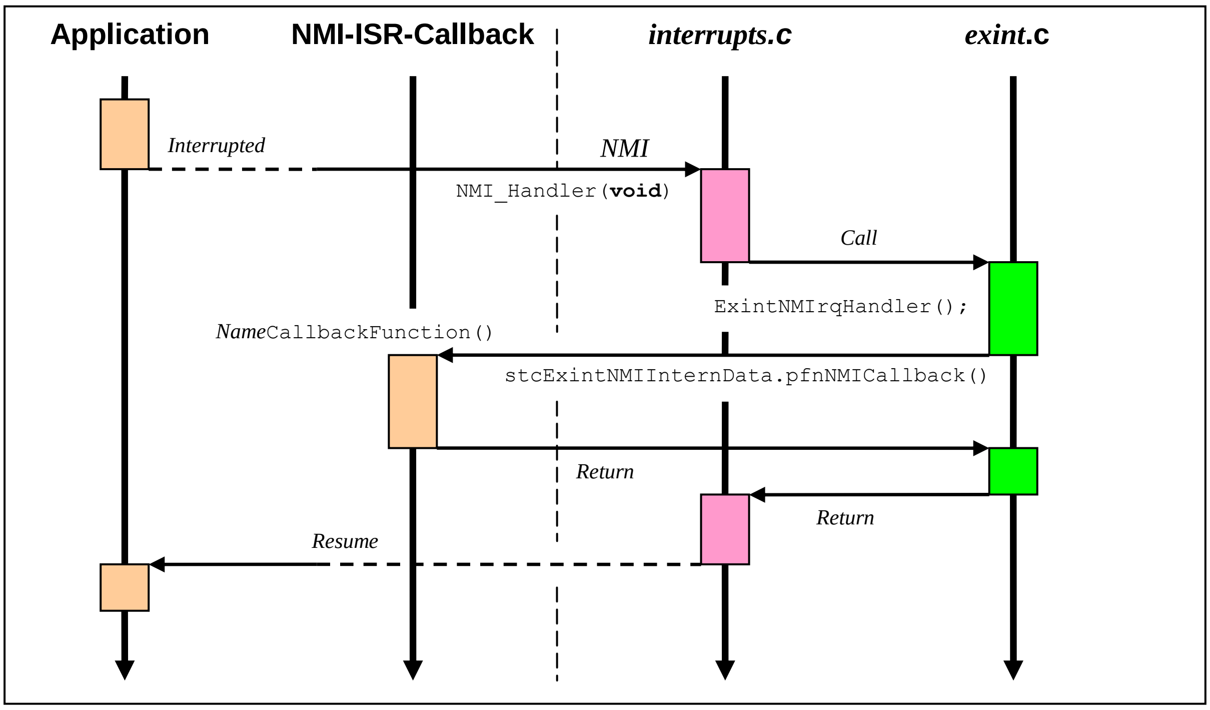


Figure 5-4: NMI and Callback Flow Diagram (ROM Execution)

The following flow shows the usual NMI service handling, if occurred during RAMCODE execution:

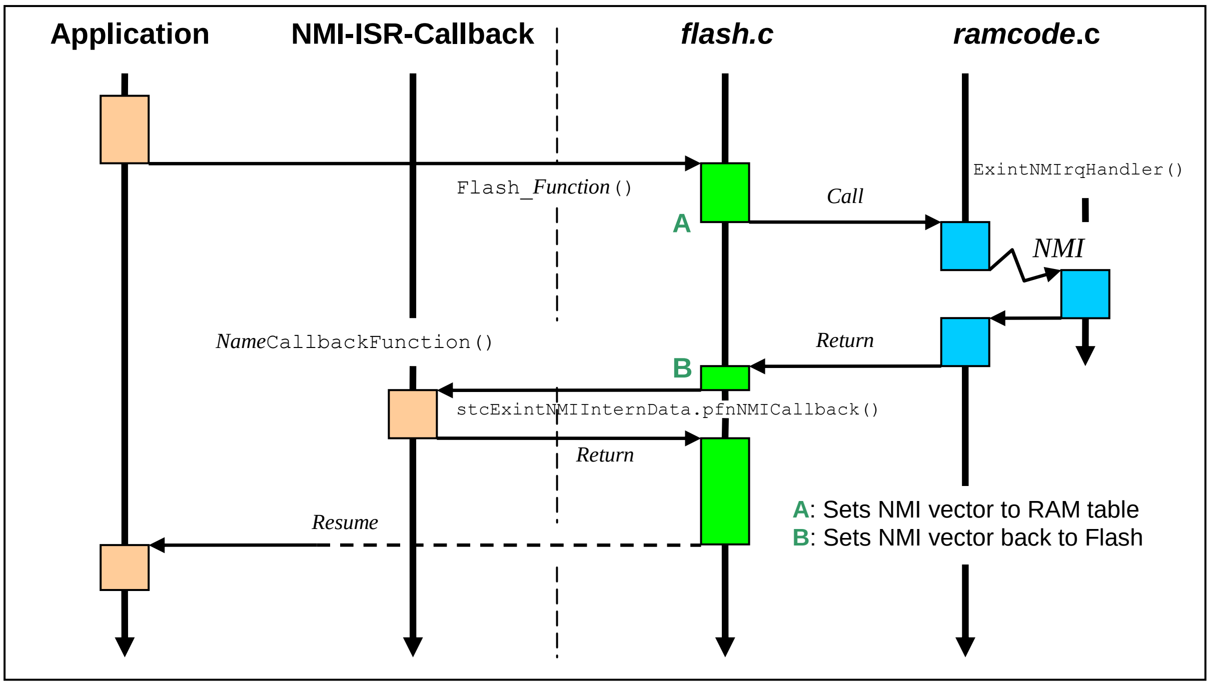


Figure 5-5: NMI and Callback Flow Diagram (RAM execution)

The necessary adjustments in *l3_user.h* are explained in chapter 4.4.7.7 and 4.4.7.8.

5.24 (RESET) Reset Cause

Type Definition	–
Configuration Type	–
Address Operator	–

This driver module does not need any configuration structure, but uses a result structure of the type of `stc_reset_result_t`, which has to provided by the caller.

5.24.1 Result Structure

The following structure is used for retrieving the Reset cause:

Structure Type		
`stc_reset_result_t`		
Type	**Field**	**Description**
`boolean_t`	`bPowerOn`	Power-on reset occurred
`boolean_t`	`bInitx`	INITX (external reset) occurred
`boolean_t`	`bLowVoltageDetection`	Low voltage reset occurred
`boolean_t`	`bSoftwareWatchdog`	Software watchdog reset occurred
`boolean_t`	`bHardwareWatchdog`	Hardware watchdog reset occurred
`boolean_t`	`bClockSupervisor`	Clock supervisor reset occurred
`boolean_t`	`bAnomalousFrequency`	Anomalous frequency detection reset occurred
`boolean_t`	`bSoftware`	A software reset was issued

5.24.2 Reset_GetCause()

This function returns the reset cause into a given structure of the type of `stc_reset_result_t` shown above.

Notes:

- The bitfields of the structure `pstcResult` must be all ‘`0`’ before calling this function.
- Calling this function clears all reset cause bits!

Prototype	
`en_result_t Reset_GetCause( stc_reset_result_t* pstcResult )`	
Parameter Name	**Description**
`[out] pstcResult`	Reset cause structure
Return Value	**Description**
`Ok`	Reset cause bitfield(s) set

5.24.3 Example Code

The following code example shows, how to call `Reset_GetCause()` correctly:

```
#include "reset.h"

function
{
  stc_reset_result_t  stcResetCause;      // Reset cause result structure

  . . .

  stcResetCause = {0};
  Reset_GetCause(stcResetCause);

  if (stcResetCause.bPowerOn == TRUE)
  {
    . . .
  }

  if (stcResetCause.bInitx == TRUE)
  {
    . . .
  }

  . . .

}
```

5.25 (RTC) Real Time Clock

Type Definition	`stc_rtc_t`
Configuration Type	`stc_rtc_config_t`
Address Operator	`RTC`*`n`*

`Rtc_Init()` initializes the RTC with the given time and date given in the RTC configuration. It also initialized the NVIC, if specidfied.

`Rtc_SetDateTime()` allows to set a new date and time.

`Rtc_SetAlarmDateTime()` allows to set a new alarm date and time.

`Rtc_EnableDisableInterrupts()` allows to change the interrupt configurations on runtime without touching the RTC work.

With `Rtc_EnableDisableAlarmRegisters()` the comparison to the recent time can be adjusted on runtime.

`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 Timer.

With `Rtc_ReadDateTimePolling()` the recent date and time can be stored in the pointered configuration structure while the Read Completion Interrupt is temporarily disabled.

`Rtc_ReadTimerStatus()` reads out the Timer status.

And finally `Rtc_DeInit()` allows to de-initialize all RTC functions and possible interrupts. Also here to touch the NVIC registers can be set.

`Rtc_GetRawTime()` is a helper function, which calculates the 'raw' time (UNIX time) from the RTC time structure `stc_rtc_time_t`.

`Rtc_SetDayOfWeek()` is a helper function, which sets the day of the week calculated from the date and time given in `stc_rtc_time_t`.

`Rtc_SetTime()` is a helper function which calculates the RTC time structure from the 'raw' time.

Note:

This driver module uses the standard C library *time.h*.

5.25.1 Configuration Structures

5.25.1.1 RTC Configuration

The RTC module uses the following channel configuration structure of the type `stc_rtc_config_t`:

Type	Field	Possible Values	Description
`uint32_t`	`u32RtcClock`	-	Used RTC input clock for sub second register
`boolean_t`	`bUseMainClock`	`TRUE` `FALSE`	Use Main Clock as input clock Use Sub Clock as input clock
`boolean_t`	`bUseFreqCorr`	`TRUE`	Use frequency correction value

		`FALSE`	Do not use frequency correction
`uint16_t`	`u16FreqCorr… Value`	–	Frequency correction value
`boolean_t`	`bUseDivider`	`TRUE` `FALSE`	Use divider for divider ration Do not use divider

`en_rtc_… divider_… ratio_t`	`enDivider… Ratio`	`RtcDivRatio`
		`…1`
		`…2`
		`…4`
		`…8`
		`…16`
		`…32`
		`…64`
		`…128`
		`…256`
		`…512`
			RIN clock division ratio :
			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

`uint8_t`	`u8FreqCorr… Cycle*`	–	Frequency correction cycle value*
`en_rtc_co_… div_t`	`enCoSignal… Div*`	`RtcCoDiv1*` `RtcCoDiv2*`	`CO` signal of RTC count part is output* `CO` signal of RTC divided by 2 is output*
`stc_rtc_… alarm_… register_t`	`stcAlarm… Register… Enable`	*see below*	*see below*

`uint16_t`
 	
`stc_rtc_… interrupts… _t`
	`u16All… Interrupts`
	
	`stcRtc… Interrupt… Enable`
		see below	*see below*

`func_ptr_… rtc_… arglist_t`	`pfnRead… Callback`	-	Callback function pointer for read completion interrupt
`func_ptr_t`	`pfnAlarm… Callback`	-	Callback function pointer for alarm interrupt
`func_ptr_t`	`pfnTimer… Callback`	-	Callback function pointer for timer interrupt
`func_ptr_t`	`pfnHalf… Second… Callback`	-	Callback function pointer for 0.5-second interrupt
`func_ptr_t`	`pfnOne… Second… Callback`	-	Callback function pointer for one-second interrupt
`func_ptr_t`	`pfnOne… Minute… Callback`	-	Callback function pointer for one-minute interrupt
`func_ptr_t`	`pfnOne… Hour… Callback`	-	Callback function pointer for one-hour interrupt

* Only available for Device Types 6 and 7

5.25.1.2 Alarm Register Enable Strucutre

The structure of the type of `stc_rtc_alarm_register_t` used by the configuration has the following 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

5.25.1.3 RTC Interrupts Enable Strucutre

The structure of the type of `stc_rtc_interrupts_t` used by the configuration has the following bit fields:

Field	Possible Values	Description
`ReadCompletionIrqEn`	0 1	Read Completion interrupt disabled Read Completion interrupt enabled
`TimeRewriteErrorIrqEn`	0 1	Timer rewrite error interrupt disabled Timer 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 enabled
`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

5.25.1.4 RTC Time structure

The structure of the type of stc_rtc_time_t has the following fields:

Type	Field	Possible Values	Description
`uint8_t`	`u8Second`	`1 ... 59`	RTC second
`uint8_t`	`u8Minute`	`1 ... 59`	RTC minute
`uint8_t`	`u8Hour`	`1 ... 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

5.25.1.5 RTC Alarm structure

The structure of the type of stc_rtc_alarm_t has the following fields:

Type	Field	Possible Values	Description
`uint8_t`	`u8AlarmMinute`	`1 ... 59`	Alarm minute
`uint8_t`	`u8AlarmHour`	`1 ... 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

5.25.1.6 RTC Timer Configuration

The RTC’s timer configuration of the type of stc_rtc_timer_config_t has the following structure:

Type	Field	Possible Values	Description
`uint32_t`	`u32TimerValue`	–	18-Bit value for RTC Timer

`boolean_t`	`bTimer…`
	`Interval…`
	`Enable`
		`TRUE`
		`FALSE`
			Timer set to count interval mode
			Timer set to one-shot mode

5.25.2 RTC Definitions

For the Month and Day of the Week values *rtc.h* provides defintions as shown 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)
 ******************************************************************************/
#define Rtc_January   1
#define Rtc_Febuary   2
#define Rtc_March     3
#define Rtc_April     4
#define Rtc_May       5
#define Rtc_June      6
#define Rtc_July      7
#define Rtc_August    8
#define Rtc_September 9
#define Rtc_October   10
#define Rtc_November  11
#define Rtc_December  12

/**
 ******************************************************************************
 ** \brief Day of week name definitions (not used in driver - to be used by
 **          user applciation)
 ******************************************************************************/
#define Rtc_Sunday    0
#define Rtc_Monday    1
#define Rtc_Tuesday   2
#define Rtc_Wednesday 3
#define Rtc_Thursday  4
#define Rtc_Friday    5
#define Rtc_Saturday  6
```

5.25.3 Rtc_Init()

This function initializes the RTC module and sets up the internal data structures. If the sub clock was not enabled before, the sub clock enable and stablilization is done in this function.

The user has to define the recent time in a structure of the type of `stc_rtc_time_t` explained in chapter 5.25.1.4.

Also a structure of the type of `stc_rtc_alarm_t` has to be defined (see chapter 5.25.1.5) even, if the alarm functionality is not used. The structure fields may contain `NULL` then.

Prototype	

`en_result_t Rtc_Init( volatile stc_rtcn_t* pstcRtc,`
`stc_rtc_config_t* pstcConfig,`
`stc_rtc_time_t* pstcTime,`
`stc_rtc_alarm_t* pstcAlarm,`
`boolean_t bTouchNvic )`	

Parameter Name	**Description**
`[in] pstcRtc`	RTC instance pointer
`[in] pstcConfig`	Pointer to RTC configuration
`[in] pstcTime`	Pointer to time structure
`[in] pstcAlarm`	Pointer to time alarm structure
`[in] bTouchNvic`	`TRUE`: Init NVIC registers `FALSE`: Do not touch NVIC registers
Return Value	**Description**
`Ok`	RTC successfully initialized
`ErrorInvalidParameter`	• `pstcRtc == NULL` • `pstcConfig == NULL` • `pstcTime == NULL` • `pstcAlarm == NULL` • `pstcMfsInternData == NULL` (Instance could not be resolved) • Other wrong setting or enumerator used

5.25.4 Rtc_DeInit()

This function de-initializes the RTC module.

Prototype	

`en_result_t Rtc_DeInit( volatile stc_rtcn_t* pstcRtc,`
`boolean_t bTouchNvic )`	

Parameter Name	**Description**
`[in] pstcRtc`	RTC instance pointer
`[in] bTouchNvic`	`TRUE`: De-Init NVIC registers `FALSE`: Do not touch NVIC registers
Return Value	**Description**
`Ok`	RTC successfully de-initialized
`ErrorInvalidParameter`	`pstcMft == NULL`

5.25.5 Rtc_SetDateTime()

This function builds two 32-bit words for minimum write access to the RTC's date and time registers.

Note:

During update global interrupts are temporarily disabled. Consinder a possible maximum polling time of 0.5 seconds!

Prototype	
`en_result_t Rtc_SetDateTime(volatile stc_rtcn_t* pstcRtc, stc_rtc_time_t* pstcRtcTime, boolean_t bPollBusy )`	
Parameter Name	**Description**
`[in] pstcRtc`	RTC instance pointer
`[in] pstcRtcTime`	Pointer to time structure
`[in] bPollBusy`	`TRUE`: `WTCR1.ST == 1` polling (busy bit) `FALSE`: No polling
Return Value	**Description**
`Ok`	RTC successfully set with new date and time
`ErrorInvalidParameter`	• `pstcRtc == NULL` • Invalid date or time format
`ErrorTimeout`	Polling loop timed out
`ErrorOperationInProgress`	Update not possible

5.25.6 Rtc_SetAlarmDateTime()

This function updates the RTC alarm time.

Prototype	
`en_result_t Rtc_SetAlarmDateTime (volatile stc_rtcn_t* pstcRtc, stc_rtc_alarm_t* pstcRtcAlarm )`	
Parameter Name	**Description**
`[in] pstcRtc`	RTC instance pointer
`[in] pstcRtcAlarm`	Pointer to alarm structure
Return Value	**Description**
`Ok`	RTC successfully set with new alarm date and time
`ErrorInvalidParameter`	Invalid date or time format

5.25.7 Rtc_EnableDisableInterrupts()

With this function the interrupts en- and disabled by the RTC configuration can be changed.

Prototype	
`en_result_t Rtc_EnableDisableInterrupts( volatile stc_rtcn_t* pstcRtc, stc_rtc_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcRtc`	RTC instance pointer
`[in] pstcConfig`	Pointer to RTC configuration structure
Return Value	**Description**
`Ok`	RTC interrupts successfully en-/disabled
`ErrorInvalidParameter`	• `pstcRtc == NULL` • `pstcConfig == NULL`

5.25.8 Rtc_EnableDisableAlarmRegisters()

With this function the Alarm registers can be en- or disabled according the configuration.

Prototype	
`en_result_t Rtc_EnableDisableAlarmRegisters( volatile stc_rtcn_t* pstcRtc, stc_rtc_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcRtc`	RTC instance pointer
`[in] pstcConfig`	Pointer to RTC configuration structure
Return Value	**Description**
`Ok`	RTC alarm registers successfully en-/disabled
`ErrorInvalidParameter`	• `pstcRtc == NULL` • `pstcConfig == NULL`

5.25.9 Rtc_ReadDateTimePolling()

This function requests a copy of the recent time to the RTC registers and waits until finish. The recent time is transfered to the configuration pointer structure.

Note:

In this function a possible enabled Read Completion Interrupt (`INTCRI`) is temporarily disabled.

Prototype	
`en_result_t Rtc_EnableDisableAlarmRegisters( volatile stc_rtcn_t* pstcRtc, stc_rtc_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcRtc`	RTC instance pointer
`[in,out] pstcConfig`	Pointer to RTC configuration structure
Return Value	**Description**
`Ok`	Date and time updated in configuration
`ErrorInvalidParameter`	• `pstcRtc == NULL` • `pstcConfig == NULL`
`ErrorTimeout`	Polling loop has timed out

5.25.10 Rtc_RequestDateTime()

This function requests a copy of the recent time to the RTC registers. In the RTC ISR the callback function `stc_rtc_config_t::pfnReadCallback` with all its 7 arguments for date and time is called.

Note:

This function needs the `INTCRIE` bit (Read Completion Interrupt Enable) set to '`1`' by `Rtc_Init()` or `Rtc_EnableDisableInterrupts()`.

Prototype	
`en_result_t Rtc_RequestDateTime( volatile stc_rtcn_t* pstcRtc )`	
Parameter Name	**Description**
`[in] pstcRtc`	RTC instance pointer
Return Value	**Description**
`Ok`	Request started
`ErrorInvalidParameter`	`pstcRtc == NULL`
`ErrorNotReady`	A previous request was not finished

5.25.11 Rtc_StatusRead()

This function provides the enumerated status (`en_rtc_clock_status_t`) of the `WTCLKM` register.

Prototype	
`en_rtc_clock_status_t Rtc_StatusRead( volatile stc_rtcn_t* pstcRtc )`	
Parameter Name	**Description**
`[in] pstcRtc`	RTC instance pointer
Return Value	**Description**
`RtcRinClockNoOperation`	RIN clock not operating
`RtcSubClockSelected`	Sub Clock is selceted
`RtcMainClockSelected`	Main Clock is selceted
`RtcInvalidInstance`	`pstcRtc == NULL`

5.25.12 Rtc_TimerSet()

This function sets the mode and Timer value of the RTC Timer. The RTC has to be initialized before to use this function properly. Also use the Timer Interrupt enable by `Rtc_Init()`.

Prototype	
`en_result_t Rtc_TimerSet( volatile stc_rtcn_t* pstcRtc, stc_rtc_timer_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcRtc`	RTC instance pointer
`[in] pstcConfig`	Pointer to RTC configuration structure
Return Value	**Description**
`Ok`	Time mode successfully set
`ErrorInvalidParameter`	• `pstcRtc == NULL` • `pstcConfig == NULL`

5.25.13 Rtc_TimerStart()

This function starts the Timer of the RTC Timer. The RTC has to be initialized before to use this function properly. Also use the Timer Interrupt enable by `Rtc_Init()`.

Prototype	
`en_result_t Rtc_TimerStart( volatile stc_rtcn_t* pstcRtc )`	
Parameter Name	**Description**
`[in] pstcRtc`	RTC instance pointer
Return Value	**Description**
`Ok`	Time successfully started
`ErrorInvalidParameter`	`pstcRtc == NULL`

5.25.14 Rtc_TimerStop()

This function stops the Timer of the RTC Timer. The RTC has to be initialized before to use this function properly. Also use the Timer Interrupt enable by `Rtc_Init()`.

Prototype	
`en_result_t Rtc_TimerStart( volatile stc_rtcn_t* pstcRtc )`	
Parameter Name	**Description**
`[in] pstcRtc`	RTC instance pointer
Return Value	**Description**
`Ok`	Time successfully stopped
`ErrorInvalidParameter`	`pstcRtc == NULL`

5.25.15 Rtc_TimerStatusRead()

Read Timer Status of RTC This function provides the status of the `TMRUN` bit of the `WTCR2` register. It returns the type `en_rtc_timer_t` as described below.

Prototype	
`en_rtc_timer_status_t Rtc_TimerStop( volatile stc_rtcn_t* pstcRtc )`	
Parameter Name	**Description**
`[in] pstcRtc`	RTC instance pointer
Return Value	**Description**
`RtcTimerNoOperation`	Timer counter is not operating
`RtcTinerInOperation`	Timer counter is operating
`RtcInvalidInstance`	`pstcRtc == NULL`

5.25.16 Rtc_GetRawTime()

This function calculates the "raw" time ('UNIX time'). It uses `mktime()` of the *time.h* library.

Prototype	
`time_t Rtc_GetRawTime( stc_rtc_time_t* pstcRtcTime )`	
Parameter Name	**Description**
`[in] pstcRtcTime`	Pointer to RTC time structure
Return Value	**Description**
`time_t`	Calculated time or `-1` on error

5.25.17 Rtc_SetDayOfWeek()

This function calculates the 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,out] pstcRtcTime`	Pointer to RTC time structure
Return Value	**Description**
`Ok`	Day of the week successfully updated in the RTC time structure
`ErrorInvalidParameter`	• `pstcRtc == NULL` • `mktime()` failed

5.25.18 Rtc_SetTime()

This function calculates from the RTC time structure "raw" time ('UNIX time'). It uses `localtime()` of the *time.h* library.

Prototype	
`en_result_t  Rtc_SetTime( stc_rtc_time_t* pstcRtcTime, time_t tRawTime )`	
Parameter Name	**Description**
`[out] pstcRtcTime`	Pointer to RTC time structure
`[in]  tRawTime`	"raw" time
Return Value	**Description**
`Ok`	RTC time structure successfully updated
`ErrorInvalidParameter`	• `pstcRtc == NULL` • `localtime()` failed

5.25.19 ReadCallback()

This function is called, if `stc_rtc_config_t::pfnReadCallback` is defined and the according interrupt is enabled. The callback function must provide the following arguments:

Callback Function	
`void ReadCallback(uint8_t u8RcbSecond, uint8_t u8RcbMinute, uint8_t u8RcbHour, uint8_t u8RcbDay, uint8_t u8RcbDayOfWeek, uint8_t u8Month, uint8_t u8Year )`	
Parameter Name	**Description**
`[in] u8RcbSecond`	Current RTC second
`[in] u8RcbMinute`	Current RTC minute
`[in] u8RcbHour`	Current RTC hour
`[in] u8RcbDay`	Current RTC day (of month)
`[in] u8RcbDayOfWeek`	Current RTC day of week
`[in] u8Month`	Current RTC month
`[in] u8Year`	Current RTC year

5.25.20 AlarmCallback()

This function is called, if `stc_rtc_config_t::pfnAlarmCallback` is defined and the according interrupt is enabled.

Callback Function
`void AlarmCallback( void )`

5.25.21 TimerCallback()

This function is called, if `stc_rtc_config_t::pfnTimerCallback` is defined and the according interrupt is enabled.

Callback Function
`void TimerCallback( void )`

5.25.22 HalfSecondCallback()

This function is called, if `stc_rtc_config_t::pfnHalfSecondCallback` is defined and the according interrupt is enabled.

Callback Function
`void HalfSecondCallback( void )`

5.25.23 OneSecondCallback()

This function is called, if `stc_rtc_config_t::pfnOneSecondCallback` is defined and the according interrupt is enabled.

Callback Function
`void OneSecondCallback( void )`

5.25.24 OneMinuteCallback()

This function is called, if `stc_rtc_config_t::pfnOneMinuteCallback` is defined and the according interrupt is enabled.

Callback Function
`void OneMinuteCallback( void )`

5.25.25 OneHourCallback()

This function is called, if `stc_rtc_config_t::pfnOneHourCallback` is defined and the according interrupt is enabled.

Callback Function
`void OneHourCallback( void )`

5.25.26 Example Codes

5.25.26.1 RTC

The following code example shows, how to use the RTC driver.

```
#include "rtc.h"

stc_rtc_config_t  stcRtcConfig;    // recommended to be global
stc_rtc_time_t    stcRtcTime;      // recommended to be global
stc_rtc_alarm_t   stcRtcAlarm;     // recommended to be global

void Rtc_ReadCallback(uint8_t u8RcbSecond,
                      uint8_t u8RcbMinute,
                      uint8_t u8RcbHour,
                      uint8_t u8RcbDay,
                      uint8_t u8RcbDayOfWeek,
                      uint8_t u8Month,
                      uint8_t u8Year)
{
  // some action ...
}

function
{
  time_t      tRawTime;
  stc_rtcn_t pstcRtc;

  stcRtcConfig.u32RtcClock       = 999999;
  stcRtcConfig.bUseMainClock     = TRUE;
  stcRtcConfig.bUseFreqCorr      = FALSE;
  stcRtcConfig.u16FreqCorrValue = 1;
  stcRtcConfig.bUseDivider       = FALSE;
  stcRtcConfig.enDividerRatio    = RtcDivRatio1;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmYearEnable   = 0;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmMonthEnable  = 0;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmDayEnable    = 0;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmHourEnable   = 0;
  stcRtcConfig.stcAlarmRegisterEnable.AlarmMinuteEnable = 0;
  stcRtcConfig.stcRtcInterruptEnable.ReadCompletionIrqEn   = 1;
  stcRtcConfig.stcRtcInterruptEnable.TimeRewriteErrorIrqEn = 0;
  stcRtcConfig.stcRtcInterruptEnable.AlarmIrqEn            = 0;
  stcRtcConfig.stcRtcInterruptEnable.TimerIrqEn            = 0;
  stcRtcConfig.stcRtcInterruptEnable.OneHourIrqEn          = 0;
  stcRtcConfig.stcRtcInterruptEnable.OneMinuteIrqEn        = 0;
  stcRtcConfig.stcRtcInterruptEnable.OneSecondIrqEn        = 0;
  stcRtcConfig.stcRtcInterruptEnable.HalfSecondIrqEn       = 0;
  stcRtcConfig.pfnReadCallback       = &Rtc_ReadCallback;
  stcRtcConfig.pfnAlarmCallback      = NULL;
  stcRtcConfig.pfnTimerCallback      = NULL;
  stcRtcConfig.pfnHalfSecondCallback = NULL;
  stcRtcConfig.pfnOneSecondCallback  = NULL;
  stcRtcConfig.pfnOneMinuteCallback  = NULL;
  stcRtcConfig.pfnOneHourCallback    = NULL;

  stcRtcTime.u8Year       = Rtc_Year(2012);
  stcRtcTime.u8Month      = Rtc_December;
  stcRtcTime.u8Day        = 31;
  stcRtcTime.u8Second     = 58;
  stcRtcTime.u8Minute     = 58;   // <= 58, because of
                                  //  stcRtcAlarm.u8AlarmMinute =
                                  //  stcRtcTime.u8Minute + 1;
  stcRtcTime.u8Hour       = 23;

  Rtc_SetDayOfWeek(&stcRtcTime); // Set Day of the week in stcRtcTime
```

▼

▲
```
  tRawTime = Rtc_GetRawTime(&stcRtcTime);
  Rtc_SetTime(&stcRtcTime, tRawTime);

  stcRtcAlarm.u8AlarmYear     = 0;  // not used here
  stcRtcAlarm.u8AlarmMonth    = 0;  // not used here
  stcRtcAlarm.u8AlarmDay      = 0;  // not used here
  stcRtcAlarm.u8AlarmMinute   = 0;  // not used here
  stcRtcAlarm.u8AlarmHour     = 0;  // not used here

  if (Ok == Rtc_Init(((volatile stc_rtcn_t*)&RTC0),
                     &stcRtcConfig,
                     &stcRtcTime,
                     &stcRtcAlarm,
                     TRUE)
     )
  {
    pstcRtc = (stc_rtcn_t*)&RTC0;

    // wait some time ...

    Rtc_RequestDateTime(pstcRtc);  // Let the callback function do
                                   //    the rest ...

     . . .

  }
}
```

5.25.26.2 RTC Timer

The RTC Timer can be used as shown in the following code example.

```
#include "rtc.h"

stc_rtc_config_t  stcRtcConfig;    // recommended to be global
stc_rtc_time_t  stcRtcTime;      // recommended to be global
stc_rtc_alarm_t stcRtcAlarm;     // recommended to be global

uint8_t u8TimerRun;

void Rtc_TimerCallback(void)
{
   u8TimerRun = 0;
}

function
{
  stc_rtc_timer_config_t stcTimerConfig;
  time_t                 tRawTime;
  stc_rtcn_t             pstcRtc;

         Initialization of RTC like in the previous example here

  stcRtcConfig.pfnTimerCallback = &Rtc_TimerCallback;

  stcTimerConfig.u32TimerValue         = 1234;
  stcTimerConfig.bTimerIntervalEnable = FALSE;

  u8TimerRun = 1;
  Rtc_TimerSet(pstcRtc, &stcTimerConfig);
  Rtc_TimerStart(pstcRtc);

  while(1 == u8TimerRun);  // wait for timer finished via callback

  . . .

  }
}
```

5.26 (USB) USB Host and USB Device

For the USB functionality please refer to the following application notes.

- mcu-an-300122
  FM3 devices and Virtual COM Port usage
- mcu-an-300124
  Manual for the Fujitsu USB Assistant/USB Wizard
- mcu-an-300126
  Description of the Fujitsu USB Host Stack
- mcu-an-300131
  Description of the Fujitsu .NET Library FujitsuUsbLib
- mcu-an-300132
  Description of the Fujitsu USB Device Library
- mcu-an-300411
  USB Host Mass Storage Boot Loader application

5.27 (WC) Watch Counter

Type Definition	`stc_wcn_t`
Configuration Type	`stc_wc_config_t`
Address Operator	`WCn`

`Wc_Init()` has to be called after configuration (date, time, prescaler settings, etc.) was done. This driver module uses the time.h standard library.

`Wc_SetTime()` calculates from the date and time given in the configuration from the elements with prefix 'Set' the Unix time and sets the internal data `stc_wc_intern_data::u32RawTime`.

Wc_GetTime() calculates from `stc_wc_intern_data::u32RawTime` the date and time and sets the substructure `stc_wc_config::stcCalendar`.

Notes:

- Because sub clock enable/disable is used the sub clock oscillation is checked and is not touched by `Wc_Init()` or `Wc_DeInit()`, if enabled before. The WC driver works independently from the CLK driver.
- This driver uses the subclock. Connecting a subclock crystal to `XA0` and `XA1` is mandatory!
- Because the WC interrupt is shared with other interrupts (e.g. Clock Stabilization), `Wc_Init()` and `Wc_DeInit()` have an argument, whether to touch the NVIC registers or not to prevent interference with other driver parts.

5.27.1 Configuration Structure

The WC has the following configuration structure of the type of `stc_wc_config_t`:

Type	**Field**	**Possible Values**	**Description**
`boolean_t`	`bEnable`	`TRUE` `FALSE`	Enable WC Disable WC

`en_wc_…`
`prescaler_t`
	`enPrescaler…`
	`Div`
		`WcPrescalerDiv`
		`…1000ms`
		`…500ms`
		`…250ms`
		`…125ms`
			Prescaler Divisor:
			2^{15}, WCCK3
			2^{14}, WCCK2
			2^{13}, WCCK1
			2^{12}, WCCK0

`uint8_t`	`u8SetSecond`	`0 ... 59`	Seconds to be set
`uint8_t`	`u8SetMinute`	`0 ... 59`	Minutes to be set
`uint8_t`	`u8SetHour`	`0 ... 23`	Hour to be set
`uint8_t`	`u8SetDay`	`1 ... 31`	Day of month to be set
`uint8_t`	`u8SetMonth`	`0 ... 11`	Month to be set
`uint8_t`	`u8SetYear`	`0 ... 235`	Years since 1900 to be set
`int8_t`	`i8SetIsdst`	`-12 ... +12`	Time zone and/or daylight savings time correction to be set
`struct tm*`	`pstcCalendar`	*see time.h for details*	WC date/time structure
`wc_cb_…` `func_ptr_t`	`pfnCallback`	-	WC one-second callback function pointer

5.27.2 WC Defintions

For calculating the `tm*` year the macro `Wc_Year()` is provided in *wc.h*. Also the values for the months are defined here.

```
/**
 ******************************************************************************
 ** \brief Month name definitions (not used in driver - to be used by
 **        user applciation)
 ******************************************************************************/
#define Wc_January   0
#define Wc_Febuary   1
#define Wc_March     2
#define Wc_April     3
#define Wc_May       4
#define Wc_June      5
#define Wc_July      6
#define Wc_August    7
#define Wc_September 8
#define Wc_October   9
#define Wc_November  10
#define Wc_December  11

/**
 ******************************************************************************
 ** \brief Year calculation macro for adjusting time.h year
 ******************************************************************************/
#define Wc_Year(a) (a - 1900)
```

5.27.3 Wc_Init()

This function initializes the WC module and sets up the internal data structures.

If the sub clock was not enabled before, the sub clock enable and stablilization is done in this function.:

Prototype	
`en_result_t Wc_Init( volatile stc_wcn_t* pstcWc, stc_wc_config_t* pstcConfig, boolean_t bTouchNvic )`	
Parameter Name	**Description**
`[in] pstcWc`	WC instance pointer
`[in] pstcConfig`	Pointer to WC configuration
`[in] bTouchNvic`	`TRUE`: Init NVIC registers `FALSE`: Do not touch the NVIC registers
Return Values	**Description**
`Ok`	WC was set up successfully
`ErrorInvalidParameter`	• `pstcWc == NULL` • `pstcWcInternData == NULL` (Instance could not be resolved) • Wrong setting used
`ErrorInvalidMode`	Sub clock cannot be enabled or not existing.

5.27.4 Wc_DeInit()

De-initializes the WC.

Prototype	
`en_result_t Wc_DeInit( volatile stc_wcn_t* pstcWc, boolean_t bTouchNvic )`	
Parameter Name	**Description**
`[in] pstcWc`	WC instance pointer
`[in] bTouchNvic`	`TRUE`: Reset NVIC registers `FALSE`: Do not touch the NVIC registers
Return Values	**Description**
`Ok`	WC was de-initialized successfully
`ErrorInvalidParameter`	• `pstcWc == NULL` • `pstcWcInternData == NULL` (Instance could not be resolved)

5.27.5 Wc_GetTime()

This function calculates the time and date, based on the UNIX-Time. The time-base is generated from a regularly triggered variable. This variable is triggered from the watch-counter ISR. The Time-struct is stored in the internal data structure `Calender`.

This function uses `localtime()` of *time.h* library.

Prototype	
`en_result_t Wc_GetTime( volatile stc_wcn_t* pstcWc )`	
Parameter Name	**Description**
`[in] pstcWc`	WC instance pointer
Return Values	**Description**
`Ok`	Date and time in internal data updated
`ErrorInvalidParameter`	• `pstcWc == NULL` • `pstcWcInternData == NULL` (Instance could not be resolved)

5.27.6 Wc_SetTime()

This function calculates the time and date, based on the UNIX-Time. The time-base is generated from a regularly triggered variable. This variable is triggered from the watch-counter ISR. The Time-struct is stored in the global structure `stcCalendar`.

This function uses `localtime()` of *time.h* library.

Prototype	
`en_result_t Wc_SetTime(volatile stc_wcn_t* pstcWc, stc_wc_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcWc`	WC instance pointer
`[in] pstcConfig`	Pointer to WC configuration
Return Values	**Description**
`Ok`	Date and time successfully set
`ErrorInvalidParameter`	• `pstcWc == NULL` • `pstcWcInternData == NULL` (Instance could not be resolved)

5.27.7 WcCallback()

On a period of 1 second a callback function is called, if its pointer in the configuration is unequal to `NULL`.

Callback Function
`void WcCallback( void )`

5.27.8 Example Code

The following code excerpt gives an example of how to use the WC driver.

```
#include "wc.h"

stc_wc_config_t  stcWcConfig;
struct tm        stcCalendar;

void WcCallback(void)
{
  uint32_t u32CurrentSecond;

  Wc_GetTime((volatile stc_wcn_t*)&WC0);
  u32CurrentSecond = stcCalendar.tm_sec;

  // Do something ...
}

function
{
  stc_wcn_t* pstcWc = NULL;

  stcWcConfig.enPrescalerDiv = WcPrescalerDiv1000ms;
  stcWcConfig.bEnable        = TRUE;
  stcWcConfig.u8SetSecond    = 11;
  stcWcConfig.u8SetMinute    = 4
  stcWcConfig.u8SetHour      = 9;
  stcWcConfig.u8SetDay       = 15
  stcWcConfig.u8SetMonth     = Wc_September;
  stcWcConfig.u8SetYear      = Wc_Year(2011);
  stcWcConfig.i8SetIsdst     = 0;
  stcWcConfig.pfnCallback    = &WcCallback;
  stcWcConfig.pstcCalendar   = &stcCalendar;

  if (Ok == Wc_Init(((volatile stc_wcn_t*)&WC0), &stcWcConfig, TRUE))
  {
    pstcWc = (stc_wcn_t*)&WC0;

    Wc_SetTime(pstcWc, &stcWcConfig);

    // Do something ...
  }
}
```

5.28 (WDG) Software and Hardware Watch Dog

Type Definition	–
Configuration Type	`stc_wdg_config_t`
Address Operator	–

Both the Hardware and Software Watchdog use the same configuration structure. Both have dedicated interrupt callback functions, in which the user has to feed the corresponding Watchdog.

For the Software Watchdog `Wdg_InitSWdg()` is used for setting the interval time. `Wdg_FeedSwWdg()` resets the Watchdog timer by a function call. `Wdg_QuickFeedSwWdg()` does the same, but the code is inline expanded for e.g. time-critical polling loops.

`Wdg_InitHwWdg()` sets the Hardware Watchdog interval. `Wdg_DisableHwWdg()` disables the Watchdog, if 2 magic words as parameters are used. `Wdg_FeedHwWdg()` and `Wdg_QuickFeedHwWdg()` 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.

Note:

- The Hardware Watchdog shares its interrupt vector with the NMI. Therefore use the `stc_wdg_config_t::bTouchNVIC` configuration whether to touch the NMI-HWWdg-NVIC registers in `Wdg_InitHwWdg()`/`Wdg_DisableHwWdg()` or not.
- The Hardware Watchdog is also switched off in `System_Init()` in *system_mb9[ab]xyz.c*. Set the definition for `HWWD_DISABLE` to `0` in *system_mb9[ab]xyz.h* for letting the Hardware Watchdog running during start-up phase.

5.28.1 Configuration Structure

Both Wardware and Software Watchdog have the following configuration structure of the type of `stc_wdg_config_t`:

Type	Field	Possible Values	Description
`uint32_t`	`u32LoadValue`	–	Timer interval
`boolean_t`	`bResetEnable`	`TRUE` `FALSE`	Enables Watchdog Reset Disables Watchdog Reset
`boolean_t`	`bIrqEnable`	`TRUE` `FALSE`	Enables Watchdog Interrupt Disables Watchdog Interrupt
`boolean_t`	`bTouchNVIC*`	`TRUE` `FALSE`	Initialize/reset NVIC registers* Do not touch NVIC registers*
`func_ptr_t`	`pfnCallback`	–	Pointer to Watchdog callback function

* Only valid for Hardware Watchdog

5.28.2 Wdg_InitSwWdg()

This function sets up the Software Watchdog according its configuration. The to enable the Software watchdog `stc_wdg_sw_config_t::bIrqEnable` must be set to `TRUE`!

Prototype	
`en_result_t Wdg_InitSwWdg( stc_wdg_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to Watchdog configuration structure
Return Values	**Description**
`Ok`	Setup sucessful
`ErrorInvalidParameter`	`pstcConfig == NULL`

5.28.3 Wdg_FeedSwWdg()

Feeds the Software Watchdog and restarts its interval.

Prototype
`void Wdg_FeedSwWdg( void )`

5.28.4 Wdg_InitHwWdg()

This function sets up the Hardware Watchdog according its configuration.

The Hardware Watchdog shares its interrupt vector with the NMI. Therefore use the `stc_wdg_config_t::bTouchNVIC` configuration whether to touch the NMI-HWWdg-NVIC registers or not.

Prototype	
`en_result_t Wdg_InitHwWdg( stc_wdg_config_t* pstcConfig )`	
Parameter Name	**Description**
`[in] pstcConfig`	Pointer to Watchdog configuration structure
Return Values	**Description**
`Ok`	Setup sucessful
`ErrorInvalidParameter`	`pstcConfig == NULL`

5.28.5 Wdg_DisableHwWdg()

This function disables the Hardware Watchdog, when the first argument is `0xC72E51A3` and the second agrument `0x89DB2E43`. The magic words are together 64-bit wide and have a balanced entropy (32 zero and 32 one bits).

Prototype

```
en_result_t Wdg_DisableHwWdg(stc_wdg_config_t* pstcConfig,
                             uint32_t          u32MagicWord1,
                             uint32_t          u32MagicWord2 )
```

Parameter Name	Description
`[in] pstcConfig`	Pointer to Watchdog configuration structure
`[in] u32MagicWord1`	Must be `0xC72E51A3`
`[in] u32MagicWord2`	Must be `0x89DB2E43`
Return Values	**Description**
`Ok`	Disable sucessful
`ErrorInvalidParameter`	Wrong magic word(s) provided

5.28.6 Wdg_FeedHwWdg()

This function feeds the Hardware Watchdog with the unlock, feed, and lock sequence. Take care of the arbitrary values, because there are not checked for plausibility!

Prototype	
`void Wdg_FeedHwWdg( uint8_t u8ClearPattern1, uint8_t u8ClearPattern2 )`	
Parameter Name	**Description**
`[in] u8ClearPattern1`	Pattern of arbitrary value
`[in] u8ClearPattern2`	Inverted arbitrary value

5.28.7 Static Inline Functions

For speed reasons the watchdog driver also provides feed functions, which are defined as static inline in the *wdg.h* file.

5.28.7.1 Wdg_QuickFeedSwWgd()

This function does exactly the same as `Wdg_FeedSwWdg()` instead of being compiled as an inline function.

Prototype
`static __INLINE void Wdg_QuickFeedSwWdg( void )`

5.28.7.2 Wdg_QuickFeedWwWgd()

This function does exactly the same as `Wdg_FeedWwWdg()` instead of being compiled as an inline function.

Prototype	
`static __INLINE void Wdg_FeedHwWdg( uint8_t u8ClearPattern1, uint8_t u8ClearPattern2 )`	
Parameter Name	**Description**
`[in] u8ClearPattern1`	Pattern of arbitrary value
`[in] u8ClearPattern2`	Inverted arbitrary value

5.28.8 WdgSwCallback()

If `stc_wdg_config_t::bIrqEnable` is `TRUE` `stc_wdg_config_t::pfnCallback` **must** be a valid pointer to a callback function!

Callback Function
`void` *`WdgSwCallback`*`( void )`

5.28.9 WdgHwCallback()

If `stc_wdg_config_t::bIrqEnable` is `TRUE` `stc_wdg_config_t::pfnCallback` **must** be a valid pointer to a callback function!

Callback Function
`void` *`WdgHwCallback`*`( void )`

5.28.10 Example Code

This code excerpt shows how to use the WDG functions.

```
#include "wdg.h"

void WdgSwCallback(void)
{
  Wdg_FeedSwWdg();     // Only for example! Do not use this in your application!
  // Do something ...
}

void WdgHwCallback(void)
{
  Wdg_QucikFeedHwWdg(0x55, 0xAA);  // Only for example! Do not use this in your
                                   //   application!
  // Do something ...
}

function
{
  stc_wdg_config_t stcWdgSwConfig;
  stc_wdg_config_t stcWdgHwConfig;

  L3_ZERO_STRUCT(stcWdgSwConfig);
  L3_ZERO_STRUCT(stcWdgHwConfig);

  stcWdgSwConfig.u32LoadValue = 12345678;
  stcWdgSwConfig.bResetEnable = TRUE;
  stcWdgSwConfig.bIrqEnable   = TRUE;
  stcWdgSwConfig.pfnCallback  = &WdgSwCallback;

  Wdg_InitSwWdg(&stcWdgSwConfig);

  stcWdgHwConfig.u32LoadValue = 0x30000;
  stcWdgHwConfig.bResetEnable = TRUE;
  stcWdgHwConfig.bIrqEnable   = TRUE;
  stcWdgHwConfig.bTouchNVIC   = TRUE;
  stcWdgHwConfig.pfnCallback  = &WdgHwCallback;

  Wdg_InitHwWdg(&stcWdgHwConfig);

  // wait for interrupts ...
}
```

6 Restrictions

THIS CHAPTER DESCRIBES THE KNOWN RESTRICTIONS

6.1 Preface

Because the FM3 architecture allows almost infinite possible applications, it is impossible that a driver library can cover each use case. Therefore the LLL was designed to cover as most of the basic possible setting for each implemented resource as feasible. It might be possible that a combination of several driver parts may lead to malfunction, although it was attentively tested.

This chapter gives an overview of known restrictions of the LLL.

6.2 Resource Activation

Because of the wide portfolio of the FM3 family it would be too complex to build-in a preprocessor logic to determine a peripheral and a certain instance availability for each derivative. The user has to check by himself via the datasheet for existence of a peripheral and a certain instance for his device.

6.3 Interrupt Handler/Enumerator Names

It may happen, that some defined interrupt handler names in the *startup_mb9[ab]xyz.s* files are not the same as used in the *interrupts.c* file of the LLL. Please adjust the names in the startup files to those used in *interrupts.c*.

Example: Devices with CAN and without Ethernet have the handler name `CANn_IRQHandler` where devices with both resources share the interrupt and have the name `ETHERn_CANn_IRQHandler`.

In the IAR environment the adjustment of the example above has to be done at marked lines in the code excerpt below:

```
                 MODULE  ?cstartup

                 ;; Forward declaration of sections.
                 SECTION CSTACK:DATA:NOROOT(3)

                 SECTION .intvec:CODE:NOROOT(2)

                 EXTERN  __iar_program_start
                 EXTERN  SystemInit
                 PUBLIC  __vector_table

                 DATA
__vector_table   DCD     sfe(CSTACK)              ; Top of Stack
                 DCD     Reset_Handler            ; Reset

                            . . .

                 DCD     CAN0_IRQHandler          ; 32: CAN ch.0
                 DCD     CAN1_IRQHandler          ; 33: CAN ch.1

                            . . .
```

Figure 6-1: Interrupt Vector Table of IAR Startup Code

In the KEIL environment the adjustment of the example above has to be done at marked lines in the code excerpt below:

```
                            . . .

; Vector Table Mapped to Address 0 at Reset

                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler

                            . . .

                DCD     CAN0_IRQHandler           ; 32: CAN ch.0
                DCD     CAN1_IRQHandler           ; 33: CAN ch.1

                            . . .
```

Figure 6-2: Interrupt Vector Table of KEIL Startup Code

For other IDE and tool chains than IAR and KEIL adjust the names accordingly.

The same appears to the interrupt enumerators for the NVIC intrinsic functions, which are defined in the device's header file *mb9[ab]xyz.h*.

Example (with same situation from above): `CANn_IRQn` versus `ETHERn_CANn_IRQn`.

```
/******************************************************************************
 * Interrupt Number Definition
 ******************************************************************************/
typedef enum IRQn
{
    NMI_IRQn            = -14, /*  2 Non Maskable      */
    HardFault_IRQn      = -13, /*  3 Hard Fault        */

       . . .

    CAN0_IRQn           = 32,  /* CAN ch.0             */
    CAN1_IRQn           = 33,  /* CAN ch.1             */

       . . .

} IRQn_Type;
```

Figure 6-3: Interrupt NVIC Enumerators of Device's Header File (*mb9[ab]xyz.h*)

6.4 Interrupt Types

The L3 only supports interrupts of Type A. Additionally the `IRQCMODER` functionality is not supported.

6.5 MFS and DMA

Because of the extensive usage of internal buffers, user buffers, interrupts, and (if existing) FIFOs of the Multi Function Serial Interfaces, DMA capability was not implemented in the LLL.

Nevertheless the user can initialize an MFS instance and use DMA transfer together with the MFS instance but cannot reuse any of the provided API functions. An own buffer handling and DMA trigger has to be developed. `pstcMfs->u8SCR.TXE` and `pstcMfs->u8SCR.RXE` has to be implemented for start/stop control of the DMA.

6.6 I^{2}C Slave Mode

The Slave mode of I^{2}C is not implemented yet (Version 1.9). Future versions of the LLL will cover this functionality.

6.7 Alternative Interrupt Vector Tables

The FM3 family provides three different types of interrupt vector tables depending on the device. The interrupt vector tables are numbered by the letters A, B, and C.

The recent version of the LLL (1.9) only provides interrupt vector type A.

6.8 LCD

LCD is not provided in the recent LLL version (1.9).

6.9 HDMI-CEC/Remote Control Reception

The HDMI-CEC/Remote Control Reception is not implemented in the recent LLL version (1.9). Future versions of the LLL will cover this functionality.

6.10 Supported Peripheral Instances

The following table shows the numbers of available instances of supported peripherals. Single built-in resources (which are not real peripherals) like Clock Module, Deep Standby Modes, or Low Voltage Detection are not listed here.

Peripheral	Maximum Number of Instances	Address Pointers of Instances
Analog Digital Converter	3	`ADC0, ADC1, ADC2`
Base Timer	8	`BT0, BT1, BT2, BT3, BT4, BT5, BT6, BT7`
Controller Area Network	2	`CAN0, CAN1`
Cyclic Redundancy Check	1	`CRC0`
Digital Analog Converter	2	`DAC0, DAC1`
Dual Timer	1	`DT0`
Ethernet Macro	2	`EMAC0, EMAC1`
External Ethernet PHY	(32)	-
Multi Function Serial	8	`MFS0, MFS1, MFS2, MFS3, MFS4, MFS5, MFS6, MFS7`
Multi Function Timer	3	`MFT0, MFT1, MFT2`
Quadrature Position and Revolution Counter	2	`QPRC0, QPRC1`
Universal Serial Bus	2	`USB0, USB1`

7 Additional Information

The software project name of the L3 is:

mb9bfxxx_l3

List of Figures