I2C driver user guide

TRAVEO™ T2G family

About this document

Scope and purpose

This guide describes the architecture, configuration, and use of the inter-integrated circuit (I2C) driver. This document explains the functionality of the driver and provides a reference to the driver's API.

The installation, the build process, and general information on the use of EB tresos are not within the scope of this document.

Intended audience

This document is intended for anyone who uses the I2C driver of the TRAVEO™ T2G family.

Document structure

Chapter 1 General overview gives a brief introduction to the I2C driver, explains the embedding in the AUTOSAR environment, and describes the supported hardware and development environment.

Chapter 2 Using the I2C driver details the steps on how to use the I2C driver in your application.

Chapter 3 Structure and dependencies describes the file structure and the dependencies for the I2C driver.

Chapter 4 EB tresos Studio configuration interface describes the driver's configuration.

Chapter 5 Functional description gives a functional description of all services offered by the I2C driver.

Chapter 6 Hardware resources gives a description of all hardware resources used.

Appendix A and Appendix B provide a complete API reference and access register table.

Abbreviations and definitions

Table 1 Abbreviations

Abbreviation	Description
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASIL	Automotive Safety Integrity Level
AUTOSAR	Automotive Open System Architecture
DEM	Diagnostic Event Manager
DET	Default Error Tracer
DMA	Direct Memory Access
EB tresos Studio	Elektrobit Automotive configuration framework
GCE	Generic Configuration Editor
HW	Hardware
SW	Software

Abbreviation	Description
ISR	Interrupt Service Routine
LSb	Least Significant Bit
MCAL	Microcontroller Abstraction Layer
MPU	Memory Protection Unit
MSb	Most Significant Bit
OVS	Oversampling
PCLK	Peripheral Clock
µC	Microcontroller

Related documents

AUTOSAR requirements and specifications

Bibliography

[1] General specification of basic software modules, AUTOSAR release 4.2.2.

[2] Complex driver design and integration guideline, AUTOSAR release 4.2.2.

[3] Specification of standard types, AUTOSAR release 4.2.2.

[4] Specification of default error tracer, AUTOSAR release 4.2.2.

Elektrobit Automotive documentation

Bibliography

[5] EB tresos Studio for ACG9 user's guide.

Hardware documentation

The hardware documents are listed in the delivery notes.

Related standards and norms

Bibliography

[6] Layered software architecture, AUTOSAR release 4.2.2.

Table of contents

About this document **1**
Table of contents **3**
1 General overview **7**
1.1 Introduction of I2C driver 7
1.2 User profile 7
1.3 Embedding in the AUTOSAR environment 7
1.4 Supported hardware 8
1.5 Development environment 8
1.6 Character set and encoding 8
2 Using the I2C driver **9**
2.1 Installation and prerequisites 9
2.2 Configuring the I2C driver 9
2.2.1 Configuration outline 9
2.3 Adapting your application 11
2.4 Starting the build process 12
2.5 Measuring the stack consumption 12
2.6 Memory mapping 12
2.6.1 Memory allocation keywords 13
2.6.2 Memory allocation and constraints 13
3 Structure and dependencies **14**
3.1 Static files 14
3.2 Configuration files 14
3.3 Generated files 14
3.4 Dependencies 15
3.4.1 PORT driver 15
3.4.2 MCU driver 15
3.4.3 AUTOSAR OS 15
3.4.4 BSW scheduler 15
3.4.5 DET 15
3.4.6 DEM 15
3.4.7 Error callout handler 16
3.4.8 DMA 16
4 EB tresos Studio configuration interface **17**
4.1 General configuration 17
4.2 I2C configuration 18
4.2.1 I2C trigger level setting 20
4.2.2 I2C use DMA channel info 20
4.2.3 I2C channel OVS config 20
4.3 Other modules 21
4.3.1 PORT driver 21
4.3.2 MCU driver 21
4.3.3 DET 21
4.3.4 DEM 21
4.3.5 AUTOSAR OS 22
4.3.6 BSW scheduler 22
5 Functional description **23**
5.1 I2C driver functionality 23

5.1.1 Initialize and prepare the buffer for the I2C driver 23
5.1.1.1 Initialize the I2C driver 23
5.1.1.2 Prepare the external buffer 23
5.1.2 Master write operation 24
5.1.2.1 Using interrupt 25
5.1.2.2 Using polling 25
5.1.2.3 Using DMA 26
5.1.3 Master read operation 26
5.1.3.1 Using interrupt 27
5.1.3.2 Using polling 27
5.1.3.3 Using DMA 28
5.1.4 Slave mode operation 28
5.1.4.1 Slave write operation 29
5.1.4.2 Slave read operation 34
5.1.4.3 Auto acknowledge configuration 37
5.1.5 Confirm the I2C driver status 38
5.1.5.1 Driver status 38
5.1.5.2 Latest job result 38
5.1.5.3 Buffer status 39
5.1.5.4 Confirm Tx Transaction 40
5.1.6 Cancel the operation 40
5.1.7 Change I2C driver settings 40
5.1.7.1 OVS settings 41
5.1.7.2 Accept slave address / slave address mask 41
5.1.7.3 Repeated Start mode 42
5.1.8 Disabling the I2C driver 42
5.1.9 De-initialize and Re-initialize at HW unit 42
5.2 What is included 42
5.3 Initialization 43
5.4 Runtime reconfiguration 43
5.5 API parameter checking 43
5.5.1 Vendor-specific development errors 43
5.6 Production errors 44
5.6.1 Recoverable failure 44
5.6.2 Unrecoverable failure 45
5.7 Reentrancy 46
5.8 Sleep mode 46
5.9 Debugging support 46
5.10 Execution-time dependencies 46
5.11 Deviation from AUTOSAR 46
6 Hardware resources 47
6.1 Ports and pins 47
6.2 Timer 47
6.3 Interrupts 47
6.4 DMA 48
7 Appendix A 49
7.1 Include files 49
7.2 Data types 49
7.2.1 I2c_ChannelIdType 49

7.2.2 I2c_BufferType 49
7.2.3 I2c_BufferSizeType 49
7.2.4 I2c_OvsIdType 49
7.2.5 I2c_SlaveAddressType 50
7.2.6 I2c_ChannelStatusType 50
7.2.7 I2c_JobResultType 50
7.2.8 I2c_TransferDirectionType 51
7.2.9 I2c_AcknowledgeType 51
7.2.10 I2c_SlaveCompleteEventType 52
7.2.11 I2c_ConfigType 52
7.3 Constants 53
7.3.1 Error codes 53
7.3.2 Version information 54
7.3.3 Module information 54
7.3.4 API service IDs 54
7.4 Functions 55
7.4.1 I2c_Init 55
7.4.2 I2c_DeInit 56
7.4.3 I2c_GetStatus 56
7.4.4 I2c_GetJobResult 57
7.4.5 I2c_Cancel 58
7.4.6 I2c_MasterWrite 59
7.4.7 I2c_MasterRead 60
7.4.8 I2c_SlaveAwaitRequest 61
7.4.9 I2c_SetupEb 62
7.4.10 I2c_GetBufferStatus 64
7.4.11 I2c_ChangeOvs 66
7.4.12 I2c_ChangeSlaveAddress 67
7.4.13 I2c_GetVersionInfo 68
7.4.14 I2c_SetRepeatedStart 69
7.4.15 I2c_GetRepeatedStart 70
7.4.16 I2c_ConfirmTxTransaction 70
7.4.17 I2c_UpdateTxBuffer 71
7.4.18 I2c_SlaveStartTransfer 73
7.4.19 I2c_InitHwUnit 74
7.4.20 I2c_DeinitHwUnit 75
7.5 Scheduled functions 76
7.5.1 I2c_MainFunction_Handling 76
7.6 Interrupt service routine 77
7.6.1 I2c_Interrupt_SCB<n>_CatX 77
7.6.2 I2c_Interrupt_DMA_CH<m>_Isr_CatY 78
7.7 Required callback functions 79
7.7.1 I2C notification functions 79
7.7.1.1 I2c_MasterTxNotification 79
7.7.1.2 I2c_MasterRxNotification 80
7.7.1.3 I2c_SlaveTxNotification 81
7.7.1.4 I2c_SlaveRxNotification 82
7.7.1.5 I2c_MasterTxErrorNotification 83
7.7.1.6 I2c_MasterRxErrorNotification 84
7.7.1.7 I2c_SlaveTxErrorNotification 85

7.7.1.8 I2c_SlaveRxErrorNotification 86
7.7.1.9 I2c_MasterComReqNotification 87
7.7.1.10 I2c_SlaveSrNotification 88
7.7.1.11 I2c_SlaveCompleteNotification 89
7.7.1.12 I2c_SlaveAddressMatchNotification 93
7.7.2 DET 95
7.7.2.1 Det_ReportError 95
7.7.3 DEM 95
7.7.3.1 Dem_ReportErrorStatus 95
7.7.4 Error callout functions 96
7.7.4.1 Error callout API 96
8 Appendix B - Access register table 97
8.1 SCB 97
8.2 DMA (DW) 104
Revision history 107
Disclaimer 112

1 General overview

1.1 Introduction of I2C driver

The I2C driver is a complex driver, which enables you to support I2C communication on special output pins of the CPU.

The I2C driver provides services for reading from and writing to devices connected via I2C buses. The I2C driver provides access to I2C communication for multiple peripherals (e.g., EEPROM, watchdog, and I/O ASICs). Master mode and slave mode are supported.

The I2C driver provides FIFO access by the following methods:

- Interrupt: By using interrupts.
- Polling: By using periodical calls to `I2c_MainFunction_Handling()`.
- DMA: By using DMA and interrupts.

The I2C driver is not responsible for initializing or configuring hardware ports. This is done by the PORT driver.

The I2C driver conforms to the AUTOSAR standard and is implemented according to the AUTOSAR complex driver design and integration guideline [2].

1.2 User profile

This guide is intended for users with a basic knowledge of the following domains:

- Embedded systems
- C programming language
- AUTOSAR standard
- Target hardware architecture

1.3 Embedding in the AUTOSAR environment

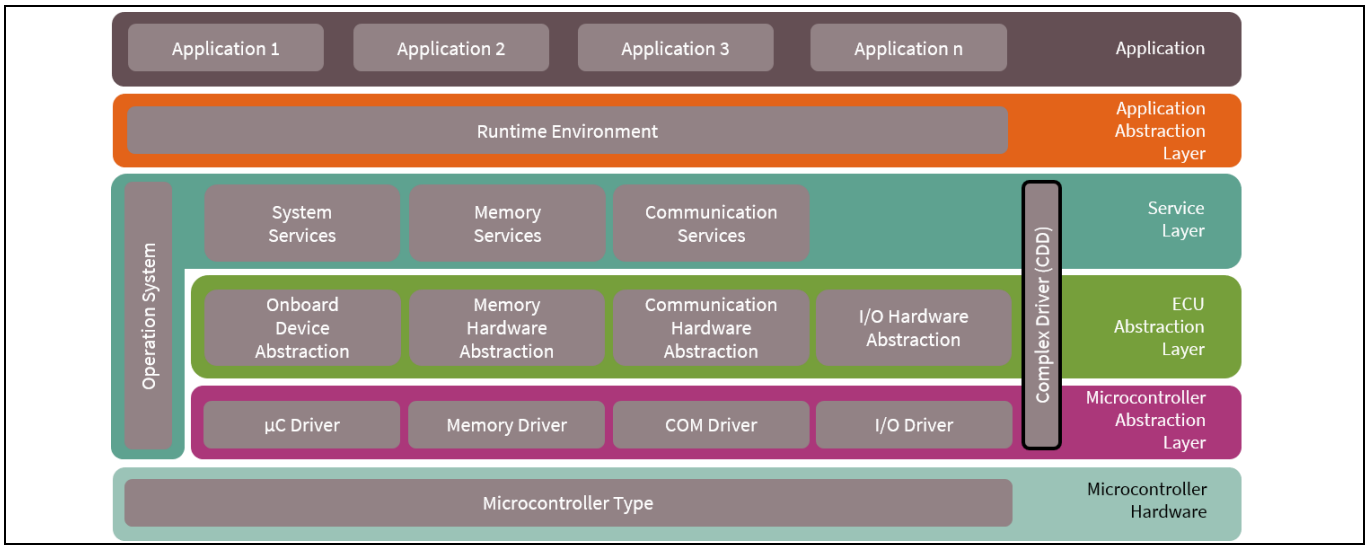

Figure 1 Overview of AUTOSAR software layers

Figure 1 shows the layered AUTOSAR software architecture. There can be multiple complex device drivers (CDD) and the I2C driver (Figure 2) is one of them. The I2C driver has similar functionality as the microcontroller abstraction layer (MCAL).

For an exact overview of the AUTOSAR layered software architecture, see *Layered software architecture* [6].

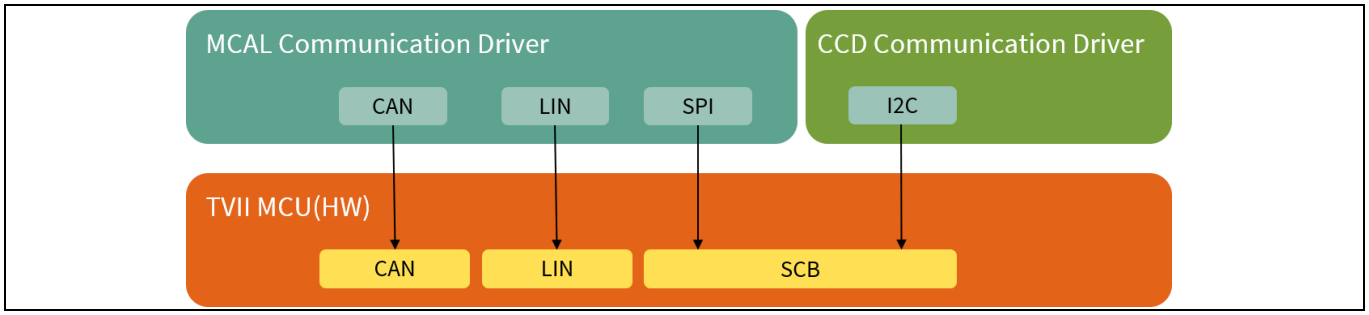

Figure 2 I2C driver in CDD

1.4 Supported hardware

This version of the I2C driver supports the TRAVEO™ T2G family. No special external hardware devices are required.

The supported derivatives are listed in the release notes.

1.5 Development environment

The development environment corresponds to AUTOSAR release 4.2.2. The Base, Make, Mcu, Port, and Resource modules are needed for the proper functionality of the I2C driver.

1.6 Character set and encoding

All source code files of the I2C driver are restricted to the ASCII character set. The files are encoded in UTF-8 format, with only the 7-bit subset (values 0x00 … 0x7F) being used.

2 Using the I2C driver

This chapter describes all necessary steps to incorporate the I2C driver into your application.

2.1 Installation and prerequisites

Before you start, see the *EB tresos Studio for ACG9 user's guide* **[5]** for the following information.

1. How to install EB tresos ECU AUTOSAR components.
2. How to use EB tresos Studio.
3. How to use the EB tresos ECU AUTOSAR build environment (includes an explanation of how to set up and integrate your application within the EB tresos ECU AUTOSAR build environment).

The installation of the I2C driver corresponds to the general installation procedure of EB tresos ECU AUTOSAR components given in the documents mentioned above. If the driver is successfully installed, it will appear in the module list of the EB tresos Studio (see *EB tresos Studio for ACG9 user's guide* **[5]**).

In the following section, it is assumed that the project is properly set up and is using the application template as described in the *EB tresos Studio for ACG9 user's guide* **[5]**. This template provides the necessary folder structure, project and Makefiles needed to configure and compile your application within the build environment. You need to be familiar with the usage of the command shell.

2.2 Configuring the I2C driver

The I2C driver can be configured with any AUTOSAR-compliant GCE tool. Save the configuration in a separate file, for example, *I2c.epc*. For more information about the I2C driver configuration, see Chapter 4 EB tresos Studio configuration interface.

2.2.1 Configuration outline

Table 2 Containers and parameters

Container	Description
`I2cDemEventParameterRefs`	Turns the DEM feature used in the I2C driver ON/OFF
`I2C_DEM_RECOVERABLE_FAILURE`	Specifies the DEM event for recoverable failures
`I2C_DEM_UNRECOVERABLE_FAILURE`	Specifies the DEM event for unrecoverable failures
`I2cGeneral`	Turns the optional APIs and features of the I2C driver ON/OFF
`I2cDevErrorDetect`	Specifies whether development error detection is used
`I2cVersionInfoApi`	Specifies whether `I2c_VersionInfo` is used
`I2cChangeOvsApi`	Specifies whether the `I2c_ChangeOvs` is used
`I2cChangeSlaveAddressApi`	Specifies whether `I2c_ChangeSlaveAddress` is used
`I2cErrorCalloutFunction`	Specifies the name of the callout function, which is called when an error occurs
`I2cOsCounterRef`	Specifies the OS counter which is used by the I2C driver
`I2cIncludeFiles`	Used for including external declaration files into the I2C driver. Specifies the header files which should be included in the I2C driver.
`I2cConfigSet`	Used for setting each I2C channel configuration
`I2cChannelConfig`	Specifies the container name for a channel configuration

Container	Description
I2cChannelId	Specifies the index number of an I2C channel
I2cScbChannelNumber	Specifies the SCB resource used for an I2C channel
I2cDefaultSlaveAddress	Specifies the default slave address for this I2C channel
I2cDefaultSlaveAddressMask	Specifies the default slave address mask for this I2C channel
I2cMasterWriteProcessing	Select interrupt or polling for master write transactions of this I2C channel
I2cMasterReadProcessing	Select interrupt or polling for master read transactions of this I2C channel
I2cSlaveProcessing	Select interrupt or polling for slave read and slave write transactions of this I2C channel
I2cUseDmaMasterTx	Specifies whether DMA is used for master write transactions
I2cUseDmaMasterRx	Specifies whether DMA is used for master read transactions
I2cUseDmaSlaveTx	Specifies whether DMA is used for slave write transactions
I2cUseDmaSlaveRx	Specifies whether DMA is used for slave read transactions
I2cChannelDefaultOvs	Specifies the default OVS settings ID
I2cBusIdleCheck	Specifies whether the bus idle check feature is used before starting a master transaction
I2cHwAutoAckSlaveAddress	Specifies whether to send an acknowledgment by HW/SW when the slave address matches
I2cHwAutoAckSlaveRxData	Specifies whether to send acknowledgment by HW/SW when receiving data in slave mode
I2cChannelOvsConfig	Contains the filter and OVS settings
I2cOvsId	Specifies the OVS settings index
I2cClockRef	Specifies the SCB input clock reference point in the MCU configuration
I2cClockRefInfo	Specifies the SCB input clock speed (Hz)
I2cDataRateMode	Specifies the SCB bus speed mode
I2cGlitcFiltering	Specifies the filter (digital, analog) used
I2cOVS	Specifies the OVS total value (low phase + high phase)
I2cLowPhaseOVS	Specifies the low phase OVS value
I2cHighPhaseOVS	Specifies the high phase OVS value
I2cBusFrequencyInfo	Specifies the I2C bus clock speed (Hz) calculated by other parameters
I2cTriggerLevelSetting	Contains the trigger level settings
I2cTxTriggerLevelMaster	Specifies the trigger level for master write transactions
I2cRxTriggerLevelMaster	Specifies the trigger level for master read transactions
I2cTxTriggerLevelSlave	Specifies the trigger level for slave write transactions
I2cRxTriggerLevelSlave	Specifies the trigger level for slave read transactions
I2cUseDmaChannelInfo	Contains the DMA channel settings
I2cDmaTxChannel	Specifies the DMA resource used for this channel's write transactions

Container	Description
I2cDmaRxChannel	Specifies the DMA resource used for this channel's read transactions

2.3 Adapting your application

To use the I2C driver in your application, include the header files of I2C, MCU and PORT driver by adding the following lines of code to your source file:

```
#include "Mcu.h"  /* AUTOSAR MCU Driver */
#include "Port.h" /* AUTOSAR PORT Driver */
#include "I2c.h"  /* I2C Driver */
```

This makes all required functions, data types, and symbolic names known to the application.

To use the I2C driver, you must configure appropriate port pins, SCB clock settings, and I2C interrupts in the PORT driver, MCU driver, and OS. For detailed information, see Chapter 6 Hardware resources.

You must initialize the MCU, PORT, and I2C driver in the following order:

```
Mcu_Init(&Mcu_Config[0]);
Port_Init(&Port_Config[0]);
I2c_Init(&I2c_Config[0]);
```

The function `Port_Init()` is called with a pointer to a structure of type `Port_ConfigType`, which is exported by the PORT driver itself.

If "interrupt" or "DMA" is used for periodic processes, an interrupt service routine must be configured in the AUTOSAR OS for each I2C peripheral, as described in 6.3 Interrupts.

If "polling" is used for periodic processes, you must call the `I2c_MainFunction_Handling` function cyclically. This can either be done by configuring the BSW scheduler accordingly, or by calling the `I2c_MainFunction_Handling` function from any other cyclic task.

All required input clocks for the configured hardware units (SCB) must be activated before initializing the I2C driver. See 3.4.2 MCU driver.

Your application must provide notification functions and their declarations. The file containing the declarations must be included using the `I2cIncludeFile` parameter. See the following example of function declarations:

```
extern void I2c_MasterTxNotification(uint8 Channel);
extern void I2c_MasterRxNotification(uint8 Channel);
extern void I2c_SlaveTxNotification(uint8 Channel);
extern void I2c_SlaveRxNotification(uint8 Channel);
extern void I2c_MasterTxErrorNotification(uint8 Channel);
extern void I2c_MasterRxErrorNotification(uint8 Channel);
extern void I2c_SlaveTxErrorNotification(uint8 Channel);
extern void I2c_SlaveRxErrorNotification(uint8 Channel);
extern void I2c_MasterComReqNotification(uint8 Channel);
extern void I2c_SlaveSrNotification(uint8 Channel);
```

```
extern void ErrorCalloutHandler(uint16 ModuleId,uint8 InstanceId,uint8
ApiId,uint8 ErrorId);
extern void I2c_SlaveCompleteNotification(uint8 Channel,
I2c_SlaveCompleteEventType Event,uint32 TransferCount);
extern I2c_AcknowledgeType I2c_SlaveAddressMatchNotification(uint8
Channel,uint8 SlaveAddress,I2c_TransferDirectionType Direction);
```

The notification functions are called from an interrupt or polling context.

2.4 Starting the build process

Do the following to build your application.

Note: *For a clean build, use the build command with target `clean_all` before executing "`make clean_all`".*

1. In the command shell, type the following command. This will generate the necessary configuration-dependent files. See 3.3 Generated files.

   ```
   > make generate
   ```

2. Type the following command to generate file dependency lists:

   ```
   > make depend
   ```

3. Compile and link the application with the following command:

   ```
   > make (optional target: all)
   ```

The application is built now. All files are compiled and linked to a binary file which can be downloaded to the target hardware.

2.5 Measuring the stack consumption

Do the following to measure the stack consumption. The Base module is needed for a proper measurement.

Note: *All files (including library files) should be rebuilt with the 'stack analysis' compiler option. The executable file built in this step must be used for stack consumption measurement only.*

1. Add the following compiler option to the makefile to enable stack consumption measurement.

   ```
   -DSTACK_ANALYSIS_ENABLE
   ```

2. Type the following command to clean library files.

   ```
   make clean_lib
   ```

3. Follow the build process described in 2.4 Starting the build process.
4. Follow the instructions in the release notes and measure the stack consumption.

2.6 Memory mapping

The *I2c_MemMap.h* file in the *$(TRESOS_BASE)/plugins/I2c_MemmapSmaple* directory is a sample. This file is replaced by the file generated by the MEMMAP module. The input to the MEMMAP module is described in the *I2c_Bswmd.arxml file* in the *$(PROJECT_ROOT)/ output/generate_swcd/swcd* directory of your project folder.

2.6.1 Memory allocation keywords

- `I2C_START_SEC_CODE_ASIL_B / I2C_STOP_SEC_CODE_ASIL_B`
  Memory section type is CODE. All executable code is allocated in this section.
- `I2C_START_SEC_CONST_ASIL_B_UNSPECIFIED / I2C_STOP_SEC_CONST_ASIL_B_UNSPECIFIED`
  Memory section type is CONST. All configuration data is allocated in this section.
- `I2C_START_SEC_VAR_NO_INIT_ASIL_B_UNSPECIFIED / I2C_STOP_SEC_VAR_NO_INIT_ASIL_B_UNSPECIFIED`
  Memory section type is VAR. All non-initialized variables without alignment restrictions are allocated in this section.
- `I2C_START_SEC_VAR_SLOW_NO_INIT_ASIL_B_UNSPECIFIED / I2C_STOP_SEC_VAR_SLOW_NO_INIT_ASIL_B_UNSPECIFIED`
  Memory section type is VAR. DMA related variables are allocated in this section.
  This section has restrictions on the allocated memory region. See 2.6.2 Memory allocation and constraints for details.
- `I2C_START_SEC_VAR_INIT_ASIL_B_UNSPECIFIED / I2C_STOP_SEC_VAR_INIT_ASIL_B_UNSPECIFIED`
  Memory section type is VAR. All initialized variables without alignment restrictions are allocated in this section.

2.6.2 Memory allocation and constraints

The CPU has a private cache that is not shared with the DMA bus master. Therefore, you must ensure that the data accessed by DMA are in uncached memory regions. The I2C driver does not support the memory allocation of DMA-related memory and data buffer to the CPU's tightly coupled memories (TCMs) and internal video RAM (VRAM).

- The section that contains the external buffers (EB) used for read transactions:
  When using DMA for read transactions, the section must be allocated to a user-specific memory region configured by the CPU's memory protection unit (MPU) as non-cacheable.
  There are no restrictions when not using DMA for read transactions.
- The section that contains the external buffers (EB) used for write transactions:
  When using DMA for write transactions, the section must be allocated to a user-specific memory region configured by the MPU as write-through or non-cacheable.
  There are no restrictions when DMA is not used for write transactions.
- The section surrounded by `I2C_START_SEC_VAR_SLOW_NO_INIT_ASIL_B_UNSPECIFIED / I2C_STOP_SEC_VAR_SLOW_NO_INIT_ASIL_B_UNSPECIFIED:`
  When using DMA, this section must be allocated to a user-specific memory region configured by the MPU as write-through or non-cacheable.
  There are no restrictions when DMA is not used.

Note: These restrictions are applied only to the Cortex®-M7 CPU because it includes TCMs, VRAM and cache. These restrictions do not apply when using the Cortex®-M4 CPU.
The areas mentioned above must be accessible through DMA and require 4-byte alignment.

Note: A CONST(rodata) data may also be generated for the CODE memory sections. An example is the jump address output by the compiler. Therefore, it is recommended to specify a CONST(rodata) data allocation keyword to the CODE memory section too. If you do not specify it in the CODE memory section, the generated CONST(rodata) data in the CODE memory section is placed in the default memory section.

3 Structure and dependencies

The I2C driver consists of static, configuration, and generated files.

3.1 Static files

Table 3 Static files

Folder	Description
*$(PLUGIN_PATH)=$(TRESOS_BASE)/plugins/I2c_TS_**	Path to the I2C driver plugin.
$(PLUGIN_PATH)/lib_src	Contains all static source files of the I2C driver. These files contain the functionality of the driver that does not depend on the current configuration. The files are used to build a static library.
$(PLUGIN_PATH)/src	Comprises configuration-dependent source files or derivative-specific files. Each file will be rebuilt when the configuration is changed. All necessary source files will automatically be compiled and linked during the build process and all include paths will be set if the I2C driver is enabled.
$(PLUGIN_PATH)/include	Basic public include directory needed by the user to include *I2c.h*.
$(PLUGIN_PATH)/autosar	Contains the AUTOSAR ECU parameter definition with vendor-, architecture-, and derivative-specific adaptations to create a correctly matching parameter configuration for the I2C driver.

3.2 Configuration files

The configuration of the I2C driver is done via EB tresos Studio. The file containing the I2C driver's configuration is named *I2c.xdm* and is in the *$(PROJECT_ROOT)/config* directory. This file serves as the input to generate configuration-dependent source and header files during the build process.

3.3 Generated files

During the build process, the following files are generated based on the current configuration description. They are in the *output/generated* subfolder of your project folder.

Table 4 Generated files

File	Description
include/I2c_Cfg.h	Contains all symbolic names for the configured I2C channels.
include/I2c_PBcfg.h	Contains the configured constants for the I2C driver.
include/I2c_ExternalInclude.h	Contains the include directives for user-configured external header files.
include/I2c_Irq.h	Contains the declaration of ISR functions.
src/I2c_PBcfg.c	Contains the configured constants for the I2C driver.
src/I2c_Irq.c	Contains the definition of ISR functions.
swcd/I2c_Bswmd.arxml	Contains BswModuleDescription.

Note: Generated source files need not to be added to your application make file. These files will be compiled and linked automatically during the build process.

*Note: Additional steps are required to generate the BSW module description. In EB tresos Studio, select **Project** > **Build Project**, and click **generate_swcd**.*

3.4 Dependencies

3.4.1 PORT driver

Although the I2C driver can be successfully compiled and linked without an AUTOSAR-compliant PORT driver, the latter is required to configure and initialize all ports. Otherwise, the I2C driver will show undefined behavior. The PORT driver needs to be initialized before the I2C driver is initialized.

3.4.2 MCU driver

The MCU driver must be initialized, and all MCU clock reference points referenced by the hardware units (SCB) via the `I2cClockRef` configuration parameter must have been activated (via calls of MCU API functions) before initializing the I2C driver. See the *MCU driver's user guide* for details.

Note that the clock, pre-scaler, or PLL settings are controlled by the MCU driver. There are no resources shared with the I2C driver. Depending on the configuration, changes in the clock settings may affect the operation of the I2C driver.

3.4.3 AUTOSAR OS

The AUTOSAR operating system handles the interrupts used by the I2C driver. See 6.3 Interrupts for more information.

The counter provided by the operating system is used by the I2C driver in the bus idle check feature.

3.4.4 BSW scheduler

The BSW scheduler handles the critical sections that are used by the I2C driver.

3.4.5 DET

If default error detection is enabled in the I2C driver configuration, DET needs to be installed, configured, and integrated into the application as well.

This driver reports DET error codes as 'instance 0'.

3.4.6 DEM

If DEM event reporting is enabled in the I2C driver configuration, DEM needs to be installed, configured, and integrated into the application as well.

To enable DEM support in the I2C driver, the `I2C_DEM_RECOVERABLE_FAILURE` and `I2C_DEM_UNRECOVERABLE_FAILURE` production error needs to be defined in the DEM configuration in the `I2cDemEventParameterRefs` container.

3.4.7 Error callout handler

The error callout handler is called on every error that is detected, regardless of whether default error detection is enabled. The error callout handler is an ASIL safety extension that is not specified by AUTOSAR. It is configured via the `I2cErrorCalloutFunction` configuration parameter.

3.4.8 DMA

DMA is supported for some hardware instances (see the datasheet for details). If a hardware instance that does not support DMA is configured to use DMA, an error will be generated.

The I2C driver does not modify the global status of the DMA hardware. You must ensure that DMA is globally enabled before using the DMA feature of the I2C driver.

4 EB tresos Studio configuration interface

The GUI is not part of this delivery. For further information, see *EB tresos Studio for ACG9 user's guide* **[5]**.

4.1 General configuration

The module comes preconfigured with default settings. You must adapt these to your environment when necessary.

Table 5 General configuration

Parameter	Description
`I2cDemEventParameterRefs`	Enables or disables the DEM functionality for the I2C driver. If this parameter is disabled, both of the following DEM functionalities are disabled: `I2C_DEM_RECOVERABLE_FAILURE` enables or disables the DEM functionality for recoverable failures, categorized as follows: • Bus protocol error (NACK, ARB_LOST, unintended STOP from external master) • Rx FIFO handling error (OVER_FLOW) `I2C_DEM_UNRECOVERABLE_FAILURE` enables or disables the DEM functionality for unrecoverable failures, categorized as follows: • Bus error • Tx FIFO handling error (OVER_FLOW) • Rx FIFO handling error (UNDER_FLOW) • DMA error
`I2cDevErrorDetect`	Enables or disables the DET functionality for the I2C driver
`I2cVersionInfoApi`	Specifies whether the `I2c_GetVersionInfo` API function is available
`I2cChangeOvsApi`	Specifies whether the `I2c_ChangeOvs` API function is available
`I2cChangeSlaveAddressApi`	Specifies whether the `I2c_ChangeSlaveAddress` API function is available
`I2cErrorCalloutFunction`	Specifies the name of the error callout function, which is called whenever an error occurs.
`I2cOsCounterRef`	Specifies the reference to the OS counter which is used by the I2C driver. This parameter must be enabled if `I2cBusIdleCheck` is enabled.
`I2cIncludeFile`	Specifies the external include files used in the I2C driver. If using this feature, the notification function and callout function declaration must be included.
`I2c_ConfigSet`	Specifies the configuration set for the I2C driver and its name

4.2 I2C configuration

Table 6 I2C configuration

Parameter	Description
`I2cChannelConfig`	Specifies the container name for channel configuration
`I2cChannelId`	Specifies the ID for the channel used in the I2C driver. It is used as a parameter for API functions. *Note: The combination of this parameter and the `I2cChannelConfig` container name should be the same in all configuration sets.*
`I2cScbChannelNumber`	Specifies the SCB resource number *Note: This parameter should be unique within a configuration set.*
`I2cDefaultSlaveAddrress`	Specifies the default slave address. This value is used for accepting slave transactions. *Note: This value does not include the R/W bit. It should not set the "general call" value (zero).*
`I2cDefaultSlaveAddrressMask`	Specifies the default slave address mask. This value is used for accepting slave transactions. *Note: This value does not include the R/W bit.*
`I2cMasterWriteProcessing`	Specifies the periodic process for master write transactions: • INTERRUPT: Using HW interrupt • POLLING: Using `I2c_MainFunction_Handling`
`I2cMasterReadProcessing`	Specifies the periodic process for master read transactions: • INTERRUPT: Using HW interrupt • POLLING: Using `I2c_MainFunction_Handling`
`I2cSlaveProcessing`	Specifies the periodic process for slave write/read transactions: • INTERRUPT: Using HW interrupt • POLLING: Using `I2c_MainFunction_Handling`
`I2cUseDmaMasterTx`	Enables or disables the DMA feature for master write transactions. *Note: If enabled, `I2cMasterWriteProcessing` must be set to `INTERRUPT`.*

Parameter	Description
I2cUseDmaMasterRx	Enables or disables the DMA feature for master read transactions. *Note: If enabled, `I2cMasterReadProcessing` must be set to `INTERRUPT`.*
I2cUseDmaSlaveTx	Enables or disables the DMA feature for slave write transactions. *Note: If enabled, `I2cSlaveProcessing` must be set to `INTERRUPT`.*
I2cUseDmaSlaveRx	Enables or disables the DMA feature for slave read transactions. *Note: If enabled, `I2cSlaveProcessing` must be set to `INTERRUPT`.*
I2cChannelDefaultOvs	Specifies the default OVS settings
I2cBusIdleCheck	Enables or disables the bus idle check feature before sending the “START” bit. *Note: If enabled, each master mode API checks the bus idle state before sending the “START” bit to the bus. It is useful for multi-master buses. However, to check the bus idle state, the hardware must wait for the stabilization of the SCB unit. Thus, the API needs more execution time than without this check.*
I2cHwAutoAckSlaveAddress	Specifies whether to send an acknowledgment by HW/SW when the slave address matches. *Note: If you need strict control flow and/or cannot avoid the interrupt delay for I2C handling, recommend to disable.*
I2cHwAutoAckSlaveRxData	Specifies whether to send an acknowledgment by HW/SW when receiving data in slave mode. *Note: If you need strict control flow and/or cannot avoid the interrupt delay for I2C handling, recommend to disable.* *Note: When using DMA for receiving data in slave mode (`I2cUseDmaSlaveRx` is enabled), `I2cHwAutoAckSlaveRxData` must also be enabled*, as software acknowledgment *is not possible.* *Note: If disabled, `I2cRxTriggerLevelSlave` must be 0 and cannot be edited.*

4.2.1 I2C trigger level setting

Table 7 I2C trigger level setting

Parameter	Description
`I2cTxTriggerLevelMaster`	Specifies the trigger level for the master write operation. If the FIFO fill level falls below this value, the periodic process is triggered.
`I2cRxTriggerLevelMaster`	Specifies the trigger level for the master read operation. This parameter is fixed to 0.
`I2cTxTriggerLevelSlave`	Specifies the trigger level for the slave write operation. If the FIFO fill level falls below this value, the periodic process is triggered.
`I2cRxTriggerLevelSlave`	Specifies the trigger level for the slave read operation. If the FIFO rises above this value, the periodic process is triggered.

Note: If these values are set higher, the interrupt (periodic process) frequency will decrease; however, the process load in one interrupt will increase. In other words, the interrupt handler execution time will be longer. Therefore, you should select an appropriate value suitable for your application.

4.2.2 I2C use DMA channel info

Table 8 I2C use DMA channel info

Parameter	Description
`I2cDmaTxChannel`	Specifies the DMA resource number to use for the Tx periodic processes. This resource is used for both master and slave transactions.
`I2cDmaRxChannel`	Specifies the DMA resource number to use for the Rx periodic processes. This resource is used for both master and slave transactions.

Note: The runtime system is responsible for globally activating DMA before using the I2C driver, if DMA is used. The selectable range of DMA resources is limited by the SCB resource in use.

4.2.3 I2C channel OVS config

Table 9 I2C channel OVS config

Parameter	Description
`I2cOvsId`	Specifies the ID for the OVS configuration set. It is used as a parameter for `I2c_ChangeOvs`.
`I2cClockRef`	Reference to the clock source configuration, which is set in the MCU driver configuration. *Note: The runtime system is responsible for activating the selected clock before using the I2C driver.*
`I2cClockRefInfo`	Specifies the SCB resource input clock value in Hz, which is referenced as `I2cClockRef`.
`I2cDataRateMode`	Specifies the I2C bus speed mode, selected from the following:

Parameter	Description
	• I2C_STANDARD_MODE • I2C_FAST_MODE • I2C_FAST_MODE_PLUS
I2cGlitchFiltering	Specifies whether the glitch filter should use a digital filter or an analog filter. • I2C_DF_in: digital filter • I2C_AF_in: analog filter
I2cOVS	Specifies the divider value of the selected frequency in I2cClockRef. The frequency divided by this value is the I2C bus speed. *Note: This value must be the same as I2cLowPhaseOVS plus I2cHighPhaseOVS.*
I2cLowPhaseOVS	Specifies the divider value of the low phase part of the frequency selected by I2cClockRef.
I2cHighPhaseOVS	Specifies the divider value of the high phase part of the frequency selected by I2cClockRef.
I2cBusFrequencyInfo	Indicates the frequency specified by I2cClockRef divided by the value specified by I2cOVS. *Note: This value should match I2cDataRateMode.*

Note: The I2C bus speed is specified by above parameters. See the hardware TRM for details. You should select appropriate values for these parameters to ensure communication with external nodes.

4.3 Other modules

4.3.1 PORT driver

The pins given in 6.1 Ports and pins must be configured in the PORT driver. The trigger multiplexer given in 6.4 DMA must be configured in the PORT driver if the DMA is configured to use.

4.3.2 MCU driver

The SCB clock must be configured.

4.3.3 DET

DET must be configured if the DET functionality is activated.

4.3.4 DEM

DEM must be configured if the DEM functionality is activated.

4.3.5 AUTOSAR OS

The I2C driver's interrupts (listed in 6.3 Interrupts) must be configured in the AUTOSAR operating system. If DMA is used, the corresponding DMA interrupt must also to be configured. The counter used by the I2C driver must be configured if `I2cBusIdleCheck` is enabled.

4.3.6 BSW scheduler

The I2C driver uses the following services of the BSW scheduler (SchM) to enter and leave critical sections:

- `SchM_Enter_I2c_I2C_EXCLUSIVE_AREA_0(void)`
- `SchM_Exit_I2c_I2C_EXCLUSIVE_AREA_0(void)`

You must ensure that the BSW scheduler is properly configured and initialized before using I2C services. The critical sections must prevent any task or interrupt from calling any I2C API function or I2C interrupt service routine.

5 Functional description

The I2C driver supports master and slave transaction modes. These modes are specified with the API call. Each mode is processed with the preconfigured method (interrupt, polling, or interrupt with DMA).

This chapter describes the basic operation of the I2C driver.

5.1 I2C driver functionality

5.1.1 Initialize and prepare the buffer for the I2C driver

5.1.1.1 Initialize the I2C driver

Before using other APIs, you must initialize the I2C driver.

1. Call I2c_Init.

In this function, the I2C driver initializes the configured SCB resource and internal variables.

Code Listing 1 Example using the I2c_Init() function with the first configuration set

```
I2c_Init(&I2cConf_I2cConfigSet_I2cConfigSet_0);
```

5.1.1.2 Prepare the external buffer

Before starting a transaction, you must prepare the external buffer (EB). This buffer is used for both master and slave operations.

1. Define the external buffer area for transmit and receive.

Code Listing 2 Example definition of the external buffer area for transmit and receive

```
uint8 TxBuffer[ DATA_SIZE_OF_TRANSMIT ];       /* external buffer for
transmit */
uint8 RxBuffer[ DATA_SIZE _OF_RECEIVE ];        /* external buffer for
receive */
```

Note: This buffer size is restricted by the memory allocation. (See 2.6.2 Memory allocation and constraints.)

Note: If the I2C driver is in the `I2C_IDLE` status, you can access the external buffer (read/write); do not access the external buffer otherwise.

2. Store the transmit data in the prepared to transmit buffer.
3. Call I2c_SetupEb.

Code Listing 3 Example using I2c_SetupEb() with the defined external buffer

```
Ret = I2c_SetupEb(channelId, &TxBuffer[0], transmit_size, &RxBuffer[0],
receive_size);
```

Note: This API must be called in the IDLE state.

Note: The transmit data is read from the address given by the second parameter. The transmit length is specified by the third parameter.

Note: The received data is stored at the address given by the fourth parameter. The receive length is specified by the fifth parameter.

Note: After calling this function, the specified external buffer and length will be cyclically reused in every master or slave operation. To change the buffer address or length, you should call `I2c_SetupEb`.

5.1.2 Master write operation

A master write operation is started as follows:

1. Call I2c_MasterWrite.

Code Listing 4 Example using I2c_MasterWrite()

```
Ret = I2c_MasterWrite(channelId, target_slave_address);    /*
target_slave_address is not contain the read/write bit */
```

Note: The target slave address should be set as a 7-bit number, starting with the MSb. (LSb is don't care.)

Note: The Tx buffer must be prepared before starting the master operation. This API should be called in the idle state.

In this function, the prepared external buffer (EB) data is stored into the Tx FIFO, and the START bit is sent to the bus.

If `I2cBusIdleCheck` is set, the bus idle state is checked before the START bit is sent. If the bus is busy (another master is using the bus), this API is declined.

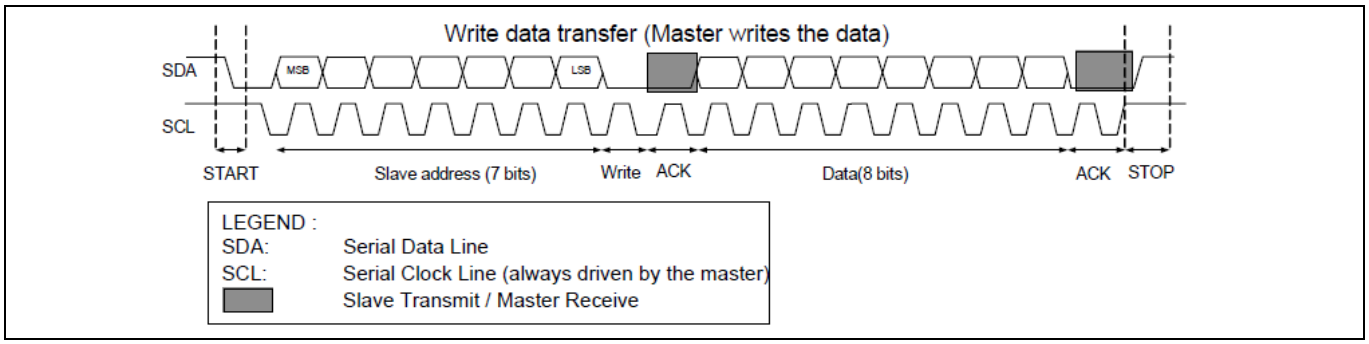

Figure 3 Master write transaction / (slave read transaction)

5.1.2.1 Using interrupt

The following operations are performed by the ISR.

If the transmit data length is greater than the Tx FIFO depth, the ISR stores the remaining data to the Tx FIFO.

If all transmit data has been sent, the ISR sends the STOP bit to the bus. After sending the STOP bit, the ISR calls the transmit complete notification function.

If repeated start mode is set, the driver calls the next communication request notification instead of sending the STOP bit to the bus. In this callback, you can call `I2c_SetupEb` or `I2c_MasterWrite` or `I2c_MasterRead` or `I2c_Cancel`. The repeated start bit is sent in the next API call (`I2c_MasterWrite` or `I2c_MasterRead`), and the STOP bit is sent in `I2c_Cancel`.

Note: Confirm that the Tx transaction ended by using the `I2c_ConfirmTxTransaction`, before you proceed to the next transaction with the repeated start bit.

Note: You can start the next transaction with repeated start (calling `I2c_SetupEb` or `I2c_MasterWrite` or `I2c_MasterRead`) or Stop request (calling `I2c_Cancel`), not only in the callback but also external of the callback. If you want to call these APIs external of the callback, take care to ensure that the corresponding situation is not judged only by status or job result. Therefore, ensure that the next communication request is notified by your own implementation (such as a set variable in callback). The relevant information is also described in 0

Note: I2c_MasterComReqNotification.

5.1.2.2 Using polling

The following operations are performed by calling `I2c_MainFunction_Handling`.

If the transmit data length is longer than the Tx FIFO depth, `I2c_MainFunction_Handling` stores the remaining data to the Tx FIFO.

If all transmit data has been sent, `I2c_MainFunction_Handling` sends the STOP bit to the bus. After sending the STOP bit, `I2c_MainFunction_Handling` calls the transmit complete notification function.

If repeated start mode is set, the driver calls the next communication request notification instead of sending the STOP bit to the bus. In this callback, you can call `I2c_SetupEb` or `I2c_MasterWrite` or `I2c_MasterRead` or `I2c_Cancel`. The repeated start bit is sent in the next API call (`I2c_MasterWrite` or `I2c_MasterRead`), and the STOP bit is sent in `I2c_Cancel`.

Note: Confirm that the Tx transaction ended by using the `I2c_ConfirmTxTransaction`, before you proceed to the next transaction with repeated start bit.

Note: You can start the next transaction with repeated start (calling `I2c_SetupEb` or `I2c_MasterWrite` or `I2c_MasterRead`) or Stop request (calling `I2c_Cancel`), not only in the callback but also external of the callback. If you want to call these APIs external of the callback, take care to ensure that the corresponding situation is not judged only by status or job result. Therefore, ensure that the next communication request is notified by your own implementation (such as a set variable in callback). The relevant information is also described in 0

Note: I2c_MasterComReqNotification.

5.1.2.3 Using DMA

The following operations are performed by the ISR and DMA:

DMA stores the remaining transmit data to the Tx FIFO. If all transmit data is sent, the ISR sends the STOP bit to the bus. After sending the STOP bit, the ISR calls the transmit complete notification function.

If repeated start mode is set, the driver calls the next communication request notification instead of sending the STOP bit to the bus. In this callback, you can call `I2c_SetupEb` or `I2c_MasterWrite` or `I2c_MasterRead` or `I2c_Cancel`. The repeated start bit is sent in the next API call (`I2c_MasterWrite` or `I2c_MasterRead`), and the STOP bit is sent in `I2c_Cancel`.

Note: DMA transfer operates when the external buffer size is 2 bytes or more.

Note: Confirm that the Tx transaction ended by using the `I2c_ConfirmTxTransaction`, before you proceed to the next transaction with repeated start bit.

Note: You can start the next transaction with repeated start (calling `I2c_SetupEb` or `I2c_MasterWrite` or `I2c_MasterRead`) or Stop request (calling `I2c_Cancel`), not only in the callback but also external of the callback. If you want to call these APIs external of the callback, then take care to ensure that the corresponding situation is not judged only by status or job result. Therefore, ensure that the next communication request is notified is already done by your own implementation (such as a set variable in callback). The relevant information is also described in 0

Note: I2c_MasterComReqNotification.

5.1.3 Master read operation

The master read operation is started as follows:

1. Call I2c_MasterRead.

Code Listing 5 Example using I2c_MasterRead()

```
Ret = I2c_MasterRead(channelId, target_slave_address);
/* target_slave_address does not contain the read/write bit */
```

Note: The target slave address should be set as a 7-bit number, starting from MSb. (LSb is don't care.)

Note: The Rx buffer must be prepared before starting the master operation. This API should be called in the IDLE state.

In this function, the START bit is sent to the bus.

If `I2cBusIdleCheck` is set, the bus idle state is checked before the START bit is sent. If the bus is busy (another master is using the bus), this API is declined.

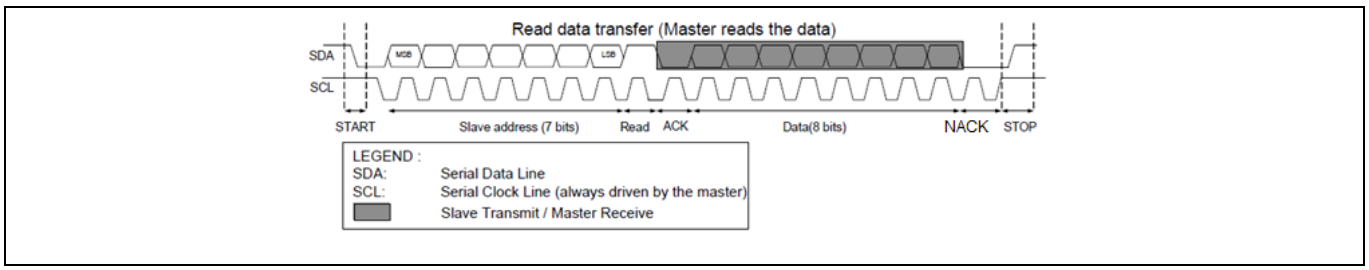

Figure 4 Master read transaction / (slave write transaction)

5.1.3.1 Using interrupt

The following operations are performed by the ISR.

When the data is received, the ISR copies the Rx FIFO data to the external buffer. If the expected amount of data has been copied, the ISR sends the NACK and STOP bits to the bus. After sending the STOP bit, the ISR calls the receive complete notification function.

If repeated start mode is set, the driver calls the next communication request notification instead of sending the STOP bit to the bus. In this callback, you can call `I2c_SetupEb` or `I2c_MasterWrite` or `I2c_MasterRead` or `I2c_Cancel`. The repeated start bit is sent in the next API call (`I2c_MasterWrite` or `I2c_MasterRead`), and the STOP bit is sent in `I2c_Cancel`.

Note: You can start the next transaction with repeated start (calling `I2c_SetupEb` or `I2c_MasterWrite` or `I2c_MasterRead`) or Stop request (calling `I2c_Cancel`), not only in the callback but also external of the callback. If you want to call these APIs external of the callback, take care to ensure that the corresponding situation is not judged only by status or job result. Therefore, ensure that the next communication request notification is already done by your own implementation (e.g. set variable in callback). The relevant information is also described in 0

Note: I2c_MasterComReqNotification.

5.1.3.2 Using polling

The following operations are performed by calling `I2c_MainFunction_Handling`.

When the data is received, `I2c_MainFunction_Handling` copies the Rx FIFO data to the external buffer. If the expected amount of data has been copied, `I2c_MainFunction_Handling` sends the NACK and STOP bits to the bus. After sending the STOP bit, `I2c_MainFunction_Handling` calls the receive complete notification function.

If repeated start mode is set, the driver calls the next communication request notification instead of sending the STOP bit to the bus. In this callback, you can call `I2c_SetupEb` or `I2c_MasterWrite` or `I2c_MasterRead` or `I2c_Cancel`. The repeated start bit is sent in the next API call (`I2c_MasterWrite` or `I2c_MasterRead`), and the STOP bit is sent in `I2c_Cancel`.

Note: You can start the next transaction with repeated start (calling `I2c_SetupEb` or `I2c_MasterWrite` or `I2c_MasterRead`) or Stop request (calling `I2c_Cancel`), not only in the callback but also external of the callback. If you want to call these APIs external of the callback, take care to ensure that the corresponding situation is not judged only by status or job result. Therefore, ensure that the next communication request is notified by your own implementation (such as a set variable in callback). The relevant information is also described in 0

Note: I2c_MasterComReqNotification.

5.1.3.3 Using DMA

The following operations are performed by the ISR and DMA.

When the data is received, DMA copies the Rx FIFO data to the external buffer. However, the remaining 128 bytes are copied by the ISR. If the expected amount of data has been copied, the ISR sends the NACK and STOP bits to the bus. After sending the STOP bit, the ISR calls the receive complete notification function.

If repeated start mode is set, the driver calls the next communication request notification instead of sending the STOP bit to the bus. In this callback, you can call `I2c_SetupEb` or `I2c_MasterWrite` or `I2c_MasterRead` or `I2c_Cancel`. The repeated start bit is sent in the next API call (`I2c_MasterWrite` or `I2c_MasterRead`), and the STOP bit is sent in `I2c_Cancel`.

Note: DMA transfer operates when the external buffer size is 129 bytes or more.

Note: You can start the next transaction with repeated start (calling `I2c_SetupEb` or `I2c_MasterWrite` or `I2c_MasterRead`) or Stop request (calling `I2c_Cancel`), not only in the callback but also external of the callback. If you want to call these APIs external of the callback, take care to ensure that the corresponding situation is not judged only by status or job result. Therefore, ensure that the next communication request is notified by your own implementation (such as a set variable in callback). The relevant information is also described in 0

Note: I2c_MasterComReqNotification.

5.1.4 Slave mode operation

A slave operation is started as follows:

1. Call I2c_SlaveAwaitRequest.

Code Listing 6 Example using I2c_SlaveAwaitRequest()

```
Ret = I2c_SlaveAwaitRequest(channelId);
```

Note: After the start of the slave operation, external bus master requests for the configured slave address are accepted.

Note: Both Tx and Rx buffers must be prepared before starting the slave operation. This API must be called in the IDLE state.

This function stores the prepared external buffer (EB) data into the Tx FIFO and waits for an external bus master request.

A slave write operation or slave read operation is performed when an external master request is received.

Note: Based on the different methods for sending an acknowledgment when slave address matching, there are two ways to confirm the actual data transfer length:

- When an acknowledgment is sent by hardware (`I2cHwAutoAckSlaveAddress` is enabled), confirm the actual transaction length by using `I2c_GetBufferStatus`. This API returns the remaining data length (the length not sent/received) as specified by `I2c_SetupEb`.
- When an acknowledgment is sent by software (`I2cHwAutoAckSlaveAddress` is disabled), confirm the actual transaction length by using `I2c_SlaveCompleteNotification`. The `TransferCount` parameter in this API represents the actual data transfer length, and the transfer completion event can be obtained through the `Event` parameter.

Note: For slave operation, if the actual transaction length is longer than the specified length, the driver handles the following transactions as follows:
In the receive operation, the data which over the specified length are read from Rx FIFO but not stored in the Rx buffer. (This means that the received data is ignored). And if further data is received, driver tries to send NACK to stop the current transaction. In the transmit operation, the default data (0xFF) is transmitted in repeatedly. Therefore, Infineon recommends use of the longer buffer (for example, max data length) for slave operation, to avoid these situations.-Even in such cases, each notification is called by receiving a STOP bit or Repeated start bit from the master.

5.1.4.1 Slave write operation

If a read request is received from the external bus master, the I2C driver starts the slave write transaction.

5.1.4.1.1 Using interrupt

The handling of slave write operations (using interrupt) differs depending on the method used to send an acknowledgment when slave address matching.

- When the slave address matches the acknowledgment is sent by hardware (`I2cHwAutoAckSlaveAddress` is enabled)

The following operations are performed by the ISR.

If the transmit data length is greater than the Tx FIFO depth, the ISR stores the remaining data in the Tx FIFO.

After receiving the STOP bit, the ISR calls the transmit complete notification function (`I2c_SlaveTxNotification`).

The driver also calls the repeated start notification (`I2c_SlaveSrNotification`), when detecting the Repeated Start (detection of the STOP bit and bus busy). In this callback, call `I2c_SetupEb` or `I2c_UpdateTxBuffer` or `I2c_SlaveAwaitRequest` to prepare the next transaction. To handle the repeated start, handle the ISR without delay. Infineon recommends setting a higher priority to the corresponding ISR.

- When the slave address matches the acknowledgment is sent by software (`I2cHwAutoAckSlaveAddress` is disabled)

The following operations are performed by the ISR.

If the slave address matching occurs, the ISR calls the slave address match notification (`I2c_SlaveAddressMatchNotification`), and the application must return the response as `I2C_ACK/I2C_HOLDACK/I2C_NACK` to this notification function when returning `I2C_ACK/I2C_NACK`, the ISR will send the corresponding acknowledge to the bus when returning `I2C_HOLDACK`, the ISR does nothing.

If the transmit data length is greater than the Tx FIFO depth, the ISR stores the remaining data in the Tx FIFO.

After receiving the STOP bit, the ISR calls the transmit complete notification function (`I2c_SlaveCompleteNotification`) and notifies the application layer of the `I2C_SLAVE_COMPLETE_STOP_TX_XX` event through the `Event` parameter.

The driver also calls the transmit complete notification function (`I2c_SlaveCompleteNotification`), when detecting the Repeated Start (detection of the STOP bit and bus busy) and notifies the application layer of the `I2C_SLAVE_COMPLETE_RESTART_TX_XX` event through the `Event` parameter. In this callback, call `I2c_SetupEb`, `I2c_UpdateTxBuffer`, or `I2c_SlaveAwaitRequest` to prepare the next transaction.

Note: If `I2cHwAutoAckSlaveAddress` and/or `I2cHwAutoAckSlaveRxData` is enabled, you can reduce the bus latency to acknowledge. However, if the driver's interrupt is disturbed by some reason (for example, critical section), the control flow to the bus may lost. You should carefully enable this configuration, depending on the use case. see 5.1.4.3 for more detail.

5.1.4.1.2 Using polling

The handling of the slave write operations (using polling) differs depending on the method used to send an acknowledgment when slave address matching.

- When the slave address matches the acknowledgment is sent by hardware (`I2cHwAutoAckSlaveAddress` is enabled)

The following operations are performed by calling `I2c_MainFunction_Handling`.

If the transmit data length is greater than the Tx FIFO depth, `I2c_MainFunction_Handling` stores the remaining data in the Tx FIFO.

After receiving the STOP bit, `I2c_MainFunction_Handling` calls the transmit complete notification function (`I2c_SlaveTxNotification`).

The driver also calls the repeated start notification (`I2c_SlaveSrNotification`), when detecting the Repeated Start (detection of the STOP bit and bus busy). In this callback, call `I2c_SetupEb` `I2c_UpdateTxBuffer`, or `I2c_SlaveAwaitRequest` to prepare the next transaction. However, Infineon does not recommend polling in the repeated start mode. Because the `I2c_MainFunction_Handling` should be called without any delay at the timing of Repeated Start receiving, to detect the next transaction start. If the `I2c_MainFunction_Handling` is delayed, the I2C driver shall not handle the next transaction as expected.

- When the slave address matches the acknowledgment is sent by software (`I2cHwAutoAckSlaveAddress` is disabled)

The following operations are performed by calling `I2c_MainFunction_Handling`.

If the slave address matching occurs, `I2c_MainFunction_Handling` calls the slave address match notification (`I2c_SlaveAddressMatchNotification`), and the application must return the response as `I2C_ACK/I2C_HOLDACK/I2C_NACK` to this notification function, when returning `I2C_ACK/I2C_NACK`, `I2c_MainFunction_Handling` will send the corresponding acknowledge to the bus when returning `I2C_HOLDACK`, `I2c_MainFunction_Handling` does nothing.

If the transmit data length is greater than the Tx FIFO depth, `I2c_MainFunction_Handling` stores the remaining data in the Tx FIFO.

After receiving the STOP bit, `I2c_MainFunction_Handling` calls the transmit complete notification function (`I2c_SlaveCompleteNotification`) and notifies the application layer of the `I2C_SLAVE_COMPLETE_STOP_TX_XX` event through the `Event` parameter.

The driver also calls the transmit complete notification function (`I2c_SlaveCompleteNotification`), when detecting the Repeated Start (detection of the STOP bit and bus busy) and notifies the application layer of the `I2C_SLAVE_COMPLETE_RESTART_TX_XX` event through the `Event` parameter. In this callback, call `I2c_SetupEb`, `I2c_UpdateTxBuffer`, or `I2c_SlaveAwaitRequest` to prepare the next transaction.

5.1.4.1.3 Using DMA

The handling of the slave write operations (using DMA) differs depending on the method used to send an acknowledgment when the slave address matches.

- When the slave address matches the acknowledgment is sent by hardware (`I2cHwAutoAckSlaveAddress` is enabled)

The following operations are performed by the ISR and DMA.

The DMA stores the remaining transmit data to Tx FIFO. After receiving the STOP bit, the ISR calls the transmit complete notification function (`I2c_SlaveTxNotification`).

The driver calls the repeated start notification (`I2c_SlaveSrNotification`) also when detecting the Repeated Start (detection of the STOP bit and bus busy). In this callback, call `I2c_SetupEb`, `I2c_UpdateTxBuffer`, or `I2c_SlaveAwaitRequest` to prepare the next transaction.

- When the slave address matches the acknowledgment is sent by software (`I2cHwAutoAckSlaveAddress` is disabled)

The following operations are performed by the ISR and DMA.

If slave address matching occurs, the ISR calls the slave address match notification (`I2c_SlaveAddressMatchNotification`), and the application must return the response as `I2C_ACK/I2C_HOLDACK/I2C_NACK` to this notification function when returning `I2C_ACK/I2C_NACK`, the ISR will send the corresponding acknowledge to the bus when returning `I2C_HOLDACK`, the ISR does nothing.

The DMA stores the remaining transmit data to Tx FIFO. After receiving the STOP bit, the ISR calls the transmit complete notification function (`I2c_SlaveCompleteNotification`) and notifies the application layer of the `I2C_SLAVE_COMPLETE_STOP_TX_XX` event through the `Event` parameter.

The driver also calls the transmit complete notification function (`I2c_SlaveCompleteNotification`), when detecting the Repeated Start (detection of the STOP bit and bus busy) and notifies the application layer of the `I2C_SLAVE_COMPLETE_RESTART_TX_XX` event through the `Event` parameter. In this callback, call `I2c_SetupEb`, `I2c_UpdateTxBuffer`, or `I2c_SlaveAwaitRequest` to prepare the next transaction.

Note: DMA transfer operates when the external buffer size is 2 bytes or more.

Note: If `I2cHwAutoAckSlaveAddress` and/or `I2cHwAutoAckSlaveRxData` is enabled, you can reduce the bus latency to acknowledge. However, if the driver's interrupt is disturbed by some reason (for example, a critical section), the control flow to the bus may lost. You should carefully enable this configuration, depending on the use case. see 5.1.4.3 for more detail.

5.1.4.1.4 Update Buffer

In the slave write operation, the buffer address and transmit length can be updated by calling `I2c_UpdateTxBuffer`.

Code Listing 7 Example using I2c_UpdateTxBuffer() with the defined external buffer

```
Ret = I2c_UpdateTxBuffer(channelId, &TxBuffer[0], transmit_size);
```

Note: Keep the external buffer area while the transmit operation is ongoing.

Note: Do not access the external buffer while the transmit operation is ongoing.

Note: The buffer must be set at least once via `I2c_SetupEb` before using this API.

Note: For external buffers, see 5.1.1.2 Prepare the external buffer

`I2c_SetupEb` can update the buffer information, but the `I2c_SetupEb` requires the driver state to be IDLE. After changing the buffer, the driver should be in the BUSY state to respond to the master by calling I2c_SlaveAwaitReqeust. Therefore, if you are using this sequence to update the buffer information, there is a period of IDLE state (this is a No response duration). The `I2c_UpdateTxBuffer` allows you to update buffer information without setting the driver state to IDLE.

`I2c_UpdateTxBuffer` is used when the driver is in the BUSY state or slave address match window. This API can call before the data transmission starts. If this API is called during data transmission, the request is declined to keep the transmit data consistency.

Note: The slave address match window occurs when the slave address matches the acknowledgment sent by software (`I2cHwAutoAckSlaveAddress` is disabled). If a slave address match occurs, the ISR will call the slave address match notification (`I2c_SlaveAddressMatchNotification`). If this notification returns `I2C_HOLDACK`, clock stretching will be applied to the bus, and the slave address match window will open. The window remains open until the application calls `I2c_SlaveStartTransfer` to start the transfer, after which the slave address match window will close.

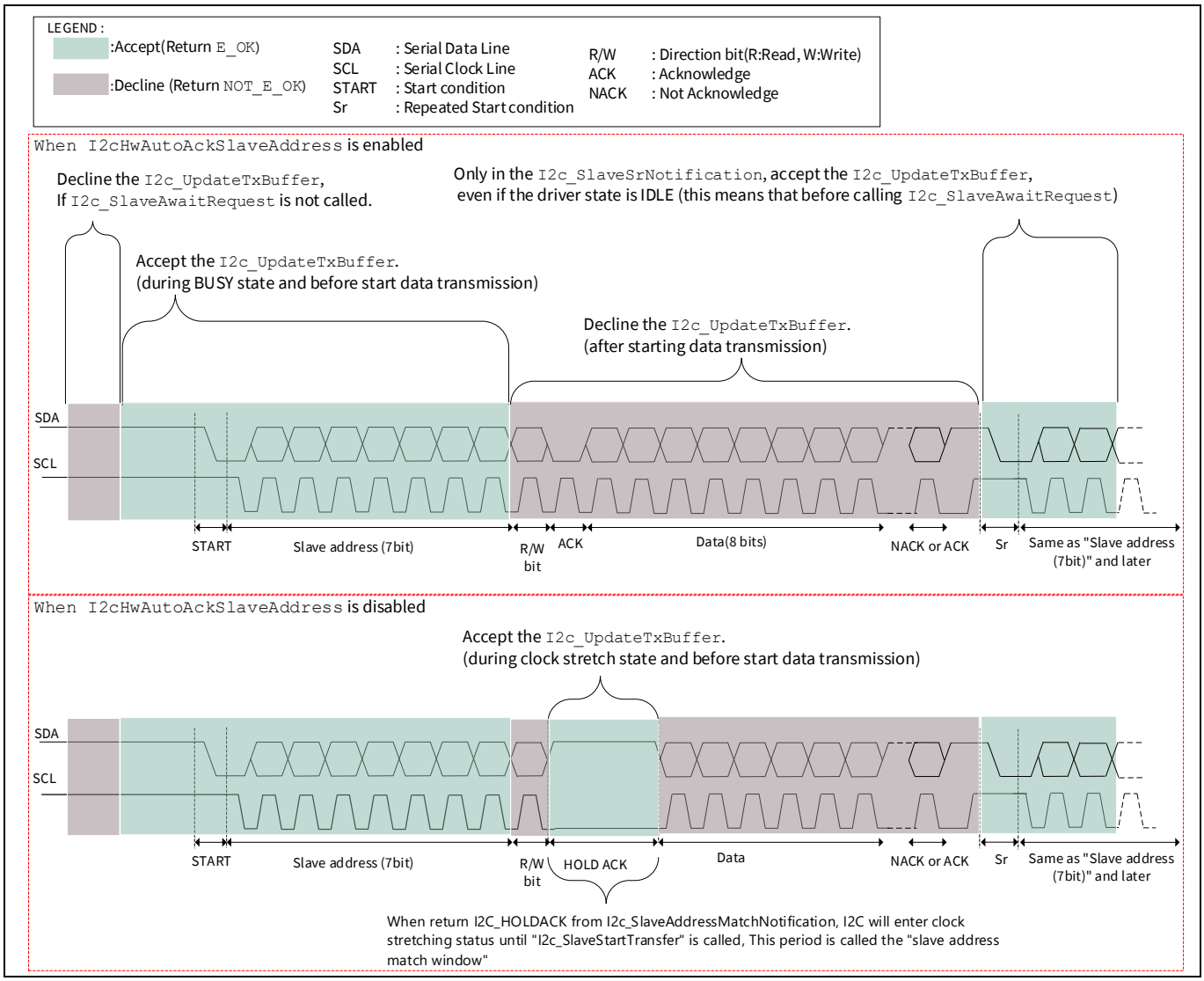

Figure 5 I2c_UpdateTxBuffer Accept / Decline

5.1.4.2 Slave read operation

When a write request is received from the external bus master, the I2C driver starts the slave read transaction.

5.1.4.2.1 Using interrupt

The handling of the slave read operations (using interrupt) differs depending on the method used to send an acknowledgment when the slave address matches and receives data.

- When slave address match acknowledgment is sent by hardware (`I2cHwAutoAckSlaveAddress` is enabled)

The following operations are performed by the ISR.

When the data is received, the ISR copies the Rx FIFO data to the external buffer and sends ACK/NACK to the master when receiving data acknowledgment is sent by software (`I2cHwAutoAckSlaveRxData` is disabled). After receiving the STOP bit, the ISR calls the receive complete notification function (`I2c_SlaveRxNotification`).

The driver also calls the repeated start notification (`I2c_SlaveSrNotification`), when detecting the Repeated Start (detection of the STOP bit and bus busy). In this callback, call `I2c_SetupEb` or `I2c_SlaveAwaitRequest` to prepare the next transaction. To handle the repeated start, handle the ISR without delay. Infineon recommends setting a higher priority to the corresponding ISR.

- When the acknowledgment is sent by software (`I2cHwAutoAckSlaveAddress` is disabled)

The following operations are performed by the ISR.

If slave address matching occurs, the ISR calls the slave address match notification (`I2c_SlaveAddressMatchNotification`), and the application must return the response as `I2C_ACK/I2C_HOLDACK/I2C_NACK` to this notification function when returning `I2C_ACK/I2C_NACK`, the ISR will send the corresponding acknowledge to the bus when returning `I2C_HOLDACK`, the ISR does nothing.

When the data is received, the ISR copies the Rx FIFO data to the external buffer and sends ACK/NACK to the master when receiving data acknowledgment is sent by software (`I2cHwAutoAckSlaveRxData` is disabled). After receiving the STOP bit, the ISR calls the receive complete notification function (`I2c_SlaveCompleteNotification`) and notifies the application layer of the `I2C_SLAVE_COMPLETE_STOP_RX_XX` event through the `Event` parameter.

The driver also calls the transmit complete notification function (`I2c_SlaveCompleteNotification`), when detecting the Repeated Start (detection of the STOP bit and bus busy) and notifies the application layer of the `I2C_SLAVE_COMPLETE_RESTART_RX_XX` event through the `Event` parameter. In this callback, call `I2c_SetupEb` or `I2c_SlaveAwaitRequest` to prepare the next transaction.

Note: If `I2cHwAutoAckSlaveAddress` and/or `I2cHwAutoAckSlaveRxData` is enabled, you can reduce the bus latency to acknowledge. However, if the driver's interrupt is disturbed by some reason (for example, critical section), the control flow to the bus may lost. Enable this configuration, depending on the use case. See 5.1.4.3 for more details.

5.1.4.2.2 Using polling

The handling of the slave read operations (using polling) differs depending on the method used to send an acknowledgment when the slave address matches and receives data.

- When the slave address matches the acknowledgment is sent by hardware (`I2cHwAutoAckSlaveAddress` is enabled)

The following operations are performed by calling `I2c_MainFunction_Handling`.

When the data is received, `I2c_MainFunction_Handling` copies the Rx FIFO data into the external buffer and sends ACK/NACK to master when receiving data acknowledgment is sent by software (`I2cHwAutoAckSlaveRxData` is disabled). After receiving the STOP bit, `I2c_MainFunction_Handling` calls the receive complete notification function (`I2c_SlaveRxNotification`).

The driver also calls the repeated start notification (`I2c_SlaveSrNotification`), when detecting the Repeated Start (detection of the STOP bit and bus busy). In this callback, call `I2c_SetupEb` or `I2c_SlaveAwaitRequest` to prepare the next transaction. However, Infineon does not recommend polling in repeated start mode. Call the `I2c_MainFunction_Handling` without any delay at the timing of Repeated Start receiving, to detect the next transaction start. If the `I2c_MainFunction_Handling` is delayed, the I2C driver shall not handle the next transaction as expected.

- When the acknowledgment is sent by software (`I2cHwAutoAckSlaveAddress` is disabled)

The following operations are performed by calling `I2c_MainFunction_Handling`.

If slave address matching occurs, `I2c_MainFunction_Handling` calls the slave address match notification (`I2c_SlaveAddressMatchNotification`), and the application must return the response as `I2C_ACK/I2C_HOLDACK/I2C_NACK` to this notification function when returning `I2C_ACK/I2C_NACK`, `I2c_MainFunction_Handling` will send the corresponding acknowledge to the bus when returning `I2C_HOLDACK`, `I2c_MainFunction_Handling` do nothing.

When the data is received, `I2c_MainFunction_Handling` copies the Rx FIFO data into the external buffer and sends ACK/NACK to master when receiving data acknowledgment is sent by software (`I2cHwAutoAckSlaveRxData` is disabled).

After receiving the STOP bit, `I2c_MainFunction_Handling` calls the transmit complete notification function (`I2c_SlaveCompleteNotification`) and notifies the application layer of the `I2C_SLAVE_COMPLETE_STOP_RX_XX` event through the `Event` parameter.

The driver also calls the transmit complete notification function (`I2c_SlaveCompleteNotification`), when detecting the Repeated Start (detection of the STOP bit and bus busy) and notifies the application layer of the `I2C_SLAVE_COMPLETE_RESTART_RX_XX` event through the `Event` parameter.

5.1.4.2.3 Using DMA

The handling of the slave read operations (using DMA) differs depending on the method used to send an acknowledgment when the slave address matches.

- When the slave address matches the acknowledgment is sent by hardware (`I2cHwAutoAckSlaveAddress` is enabled)

The following operations are performed by the ISR and DMA.

When the data is received, the DMA copies the Rx FIFO data to the external buffer. After receiving the STOP bit, the ISR calls the receive complete notification function (`I2c_SlaveRxNotification`).

The driver calls the repeated start notification (`I2c_SlaveSrNotification`) also when detecting the Repeated Start (detection of the STOP bit and bus busy). In this callback, call `I2c_SetupEb` or `I2c_SlaveAwaitRequest` to prepare the next transaction.

- When the acknowledgment is sent by software (`I2cHwAutoAckSlaveAddress` is disabled)

The following operations are performed by the ISR and DMA.

If slave address matching occurs, the ISR calls the slave address match notification (`I2c_SlaveAddressMatchNotification`), and the application must return the response as `I2C_ACK/I2C_HOLDACK/I2C_NACK` to this notification function when returning `I2C_ACK/I2C_NACK`, the ISR will send the corresponding acknowledge to the bus when returning `I2C_HOLDACK`, the ISR does nothing.

When the data is received, the DMA copies the Rx FIFO data to the external buffer. After receiving the STOP bit, the ISR calls the receive complete notification function (`I2c_SlaveCompleteNotification`) and notifies the application layer of the `I2C_SLAVE_COMPLETE_STOP_RX_XX` event through the `Event` parameter.

The driver also calls the transmit complete notification function (`I2c_SlaveCompleteNotification`), when detecting the Repeated Start (detection of the STOP bit and bus busy) and notifies the application layer of the `I2C_SLAVE_COMPLETE_RESTART_RX_XX` event through the `Event` parameter. In this callback, call `I2c_SetupEb` or `I2c_SlaveAwaitRequest` to prepare the next transaction.

Note: When using DMA for receiving data in slave mode, the configuration of `I2cHwAutoAckSlaveRxData` must be enabled. (Can not be acknowledged by software)

Note: When the following conditions are met, there is a limitation to using DMA feature for slave mode.
1. Master Write transaction is end with repeated start. Or next transaction is started immediately.
2. Master Write transaction length can be less than set length for driver.
3. Configuration, `I2cHwAutoAckSlaveAddress` is disabled.
4. Application cannot manage the ISR for Repeated Start bit received (transaction end ISR) before slave address match.
If all above conditions (1 to 4) are met, the DMA try to transfer received slave address to buffer without driver control. When the transaction end ISR is managed during this DMA transfer, driver will call the DEM error report. Also, if transaction end ISR is handled after DMA transfer for slave address is completed, received slave address is already send to buffer and missing for next transaction control. This means, I2c_SlaveAddressMatchNotificaion parameter (SlaveAddress) will be 0. This behavior could not be avoided by driver, hence if the application has possibility to face this situation, we strongly recommend to NOT use DMA feature for slave reception.

Note: If `I2cHwAutoAckSlaveAddress` and/or `I2cHwAutoAckSlaveRxData` is enabled, you can reduce the bus latency to acknowledge. However, if the driver's interrupt is disturbed by some reason (e.g. critical section), the control flow to bus may lost. You should carefully to enabling this configuration, depending on use case. see 5.1.4.3 for detail.

5.1.4.3 Auto acknowledge configuration

There are two configurations for controlling the method of sending acknowledgment in slave mode: `I2cHwAutoAckSlaveAddress` and `I2cHwAutoAckSlaveRxData`. below is a description of the functionality of these two configurations, as well as the trade-offs when they are enabled or disabled.

- `I2cHwAutoAckSlaveAddress` is enabled

Table 10 Description of the functionality when `I2cHwAutoAckSlaveAddress` is enabled

Functionality	Option
Send acknowledge method when matching the slave address	Hardware
`I2c_SlaveAddressMatchNotification`	Not called
Slave transfer complete notification when detecting the Stop condition	`I2c_SlaveTxNotification`/ `I2c_SlaveRxNotification`
Slave transfer complete notification when detecting the Repeated Start condition	`I2c_SlaveSrNotification`

- `I2cHwAutoAckSlaveAddress` is disabled

Table 11 Description of the functionality when `I2cHwAutoAckSlaveAddress` is disabled

Functionality	Option
Send acknowledge method when matching the slave address	Software
`I2c_SlaveAddressMatchNotification`	Called
Slave transfer complete notification when detecting the Stop condition	`I2c_SlaveCompleteNotification` with `I2C_SLAVE_COMPLETE_STOP_XX_XX` event
Slave transfer complete notification when detecting the Repeated Start condition	`I2c_SlaveCompleteNotification` with `I2C_SLAVE_COMPLETE_RESTART_XX_XX` event

Note: If the master performs continuous data transmission in the START -> STOP -> START -> STOP sequence and the delay between the intermediate STOP and START is very short, due to the HW spec, SW cannot distinguish between the repeated start and standard start bit. As a result, it will treat the second START as a repeated start condition and pass the `I2C_SLAVE_COMPLETE_RESTART_XX_XX` event through the `Event` parameter in the final transmission completion notification (`I2c_SlaveCompleteNotification`).

Note: If `I2cHwAutoAckSlaveAddress` and/or `I2cHwAutoAckSlaveRxData` is enabled, it can reduce the latency to bus acknowledge. In other words, you can reduce the clockstretch. However, if this is enabled and the driver's interrupt is disturbed by some reason (e.g. critical section), hardware acknowledges to the bus without software interaction. This may cause the application to lose the control flow to the bus. For example, there is a risk of transmitting the remaining data from a previous transaction to the next transaction in a repeated start scenario. Another example is incorrectly sending a NACK in response to a repeated start request. Such situations can occur when the driver ISR is not handled in time. Therefore, you should carefully select this configuration based on your system requirements.

- `I2cHwAutoAckSlaveRxData` is enabled

Table 12 Description of the functionality when `I2cHwAutoAckSlaveRxData` is enabled

Functionality	Option
Send acknowledge method when receiving data	Hardware

Note: The acknowledgment will be sent by hardware when receiving data under the expected length, once the received data length reaches the expected length, the I2C driver changes the HW setting to send the further acknowledgment from software, and a NACK will be sent.

- `I2cHwAutoAckSlaveRxData` is disabled

Table 13 Description of the functionality when `I2cHwAutoAckSlaveRxData` is disabled

Functionality	Option
Send acknowledge method when receiving data	Software

5.1.5 Confirm the I2C driver status

To confirm the driver progress or driver status, you can use the following features.

5.1.5.1 Driver status

1. Call I2c_GetStatus.

   This function returns one of the following statuses.
   `I2C_UNINIT`: Not yet initialized
   `I2C_IDLE`: No transaction request
   `I2C_BUSY_MASTERTX`: A master write operation is in progress.
   `I2C_BUSY_MASTERRX`: A master read operation is in progress.
   `I2C_BUSY_SLAVE`: Waiting for the slave operation to complete.
   `I2C_BUSY_SLAVETX`: A slave write operation is in progress.
   `I2C_BUSY_SLAVERX`: A slave read operation is in progress.
   `I2C_UNKNOWN_STATUS`: Cannot return the status (invalid channel ID)

Code Listing 8 Example using I2c_GetStatus()

```
Status = I2c_GetStatus(channelId);
```

5.1.5.2 Latest job result

A job means a transaction. Thus, the job result is the same as the result of the transaction.

1. Call I2c_GetJobresult.

   This function returns one of the following statuses.
   `I2C_NORESULT`: No job after initialization
   `I2C_PENDING`: A job is in progress.
   `I2C_CANCEL`: The previous job was canceled.
   `I2C_MASTER_TX_SUCCESS`: The previous master Tx job was successful.
   `I2C_MASTER_RX_SUCCESS`: The previous master Rx job was successful.
   `I2C_SLAVE_TX_SUCCESS`: The previous slave Tx job was successful.

`I2C_SLAVE_RX_SUCCESS`: The previous slave Rx job was successful.
`I2C_MASTER_TX_ERROR`: The previous master Tx job failed .
`I2C_MASTER_RX_ERROR`: The previous master Rx job failed .
`I2C_SLAVE_TX_ERROR`: The previous slave Tx job failed.
`I2C_SLAVE_RX_ERROR`: The previous slave Rx job failed.
`I2C_UNKNOWN_RESULT`: Cannot return a result (the channel ID was invalid or uninitialized)

Code Listing 9 Example using I2c_GetJobResult()

```
JobResult = I2c_GetJobResult(channelId);
```

5.1.5.3 Buffer status

You can check the progress of the transaction when it is underway.

1. Call I2c_GetBufferStatus.
   This function outputs the current Tx and Rx buffer pointer address and the length of the remaining data. If the pointer address or the length of the remaining data does not change (the bus transaction appears stalled) for reasons such as waiting for an external node's reaction, you can take action such as canceling the operation to prevent the system from freezing.

   This function can be used to confirm the actual transaction length in slave mode. If you use it for this purpose, you can call this function in each transaction's complete notification. If there is any difference between the specified length in `I2c_SetupEb` and the actual transaction length, the difference length can be found in this function parameter. In slave mode, when the default data sending (TX) or receiving (RX) is ignored by a longer data request from the Master, this function returns the last position and the length 0. This is because the external buffer data transaction is already complete.

   Note: This function returns the calculated buffer status (position/length). Thus, the returned value has limited accuracy. If DMA is used, the return value is a rough estimate.

   Note: When `I2cHwAutoAckSlaveAddress` is enabled, using `I2c_GetBufferStatus` to confirm the actual transaction length may become inaccurate in slave mode. Therefore, Infineon recommends using the transfer completion notification (`I2c_SlaveCompleteNotification`) to confirm the actual data transfer length.

Code Listing 10 Example using the I2c_GetBufferStatus()

```
I2c_BufferType * SrcAddressPtr;                  /* variable definition */
I2c_BufferType * DestAddressPtr;                 /* variable definition */
I2c_BufferSizeType SrcSize;                      /* variable definition */
I2c_BufferSizeType DestSize;                     /* variable definition */
Ret = I2c_GetBufferStatus(channelId,
                          &SrcAddressPtr,
                          &DestAddressPtr,
                          &SrcSize,
                          &DestSize );
```

5.1.5.4 Confirm Tx Transaction

After calling the MasterWrite in repeated start mode, make sure that all the Tx transactions are complete before you start the next transaction with repeated start.

1. Call I2c_ConfirmTxTransaction.

   This function confirms if the Tx transaction has ended or not.

   `E_OK`: Tx Transaction ended

   `E_NOT_OK`: Tx Transaction not ended

Code Listing 11 Example using I2c_ConfirmTxTransaction()

```
Ret = I2c_ConfirmTxTransaction(channelId);
```

Note: If you start the next Master transaction without confirming the Tx transaction ended, there is a possibility to broken the current transaction.

Note: This function is intended for use in the Master mode channel.

5.1.6 Cancel the operation

You can terminate an operation underway.

1. Call I2c_Cancel.

Code Listing 12 Example using I2c_Cancel()

```
Ret = I2c_Cancel(channelId);
```

Note: If this function is called, the I2C driver tries to stop the transaction (For example, if a master read operation is ongoing, it tries to send NACK and STOP). However, the communication partner may ignore this request. For example, if a slave write operation is underway, the driver clears the FIFO and disables the SCB, but the external master may continue to read the data. Such cases are difficult to avoid; therefore, you should pay extra attention when calling this function.

Note: If this function is called during slave write operation, the external master may detect a bus error.

This function terminates the currently ongoing operation. If the operation is terminated, the callback notification will not be called.

If repeated start mode is set and call this function in the next communication request notification, the driver sends the STOP bit in this function. In this case, the transaction complete callback notification will be called.

5.1.7 Change I2C driver settings

Even if not using reinitialization, the following settings can be changed.

5.1.7.1 OVS settings

The OVS settings can be changed as follows:

1. Call I2c_ChangeOvs.

Code Listing 13 Example using the I2c_ChangeOvs()

```
Ret = I2c_ChangeOvs(channelId, OvsId);
/* OvsId should be select from the configured one */
```

Note: This API should be called in the IDLE state. You should ensure that the clock setting is in sync with the input SCB clock setting.

This function changes the current OVS setting to another configured one, which is specified by the given parameter (OVS setting ID).

5.1.7.2 Accept slave address / slave address mask

The slave address / slave address mask value can be changed as follows:

1. Call I2c_ChangeSlaveAddress.

Code Listing 14 Example using the I2c_ChangeSlaveAddress()

```
Ret = I2c_ChangeSlaveAddress (channelId, Address, AddressMask);
```

Note: This API should be called in the IDLE state. The slave address should be set as a 7-bit number, starting with MSb.
You should avoid setting a slave address which only accepts the “general call (I2C protocol)” (address = 0, address mask = 0xFF).

5.1.7.3 Repeated Start mode

The repeated start mode can be changed as follows:

1. Call I2c_SetRepeatedStart.

Code Listing 15 Example using the I2c_SetRepeatedStart ()

```
Ret = I2c_SetRepeatedStart (channelId, RepeatedFlag);
/* parameter: RepeatedFlag */
/* TRUE: Repeated start mode */
/* FALSE: Normal mode */
```

Note: This API should be called in the IDLE state.

5.1.8 Disabling the I2C driver

To stop the I2C driver, use the following API:

1. Call I2c_DeInit.

Code Listing 16 Example using I2c_DeInit()

```
I2c_DeInit();
```

Note: In this function, the I2C driver resets the SCB settings to their reset values.

5.1.9 De-initialize and Re-initialize at HW unit

To De-initialize and Re-initialize an individual SCB channel, please follow the sequence outlined below:

Code Listing 17 Example De-initialize and Re-initialize an HW unit

```
1  I2c_Init(&I2cConf_I2cConfigSet_I2cConfigSet_0);

2  /* De-initializes and Re-initializes the SW and HW for the channelA */
3  I2c_DeinitHwUnit(channelA);
4  I2c_InitHwUnit(channelA);

5  /* De-initializes and Re-initializes the SW and HW for the channelB */
6  I2c_DeinitHwUnit(channelB);
7  I2c_InitHwUnit(channelB);

8  I2c_DeInit();
```

Note: The functions `I2c_DeinitHwUnit()` and `I2c_InitHwUnit()` shall only be invoked between the call to `I2c_Init()` and the call to `I2c_DeInit()`.

5.2 What is included

The *I2c.h* file includes all necessary external identifiers. Thus, your application only needs to include *I2c.h* to make all API functions and data types available.

5.3 Initialization

The I2C driver must be initialized before use by calling the `I2c_Init` API function.

Before using the I2C driver, the following is needed:

- The PORT and MCU module must be initialized
- If you use DMA, ensure that DMA is globally enabled before using the DMA feature of the I2C driver.
- Prepare other BSW modules (see 4.3 Other modules).

5.4 Runtime reconfiguration

To change the configuration set, disable the I2C driver (using `I2c_DeInit`). After this, initialize the I2C driver (using `I2c_Init`) with another configuration set.

To change a part of the configuration, see 5.1.7 Change I2C driver settings. This feature does not require disabling the I2C driver.

5.5 API parameter checking

The I2C driver's services perform regular error checks.

When an error occurs, the error hook routine (configured via `I2cErrorCalloutFunction`) is called with the error code, service ID, module ID, and instance ID as parameters.

If default error detection is enabled, all errors are also reported to DET, a central error hook function within the AUTOSAR environment. The checking itself cannot be deactivated for safety reasons.

See 7.4 Functions for a description of API functions and associated error codes.

5.5.1 Vendor-specific development errors

The I2C driver is not included in the AUTSOAR specification; therefore, all parameter error checks are vendor-specific.

Table 14 Vendor-specific development errors

Error	Description
`I2C_E_UNINIT`	An API (except the `I2c_Init`, `I2c_GetStatus`, `I2c_GetVersionInfo`) is called before the initialization of the I2C driver.
`I2C_E_ALREADY_INITIALIZED`	`I2c_Init` is called when the driver is already initialized, without calling `I2c_DeInit`.
`I2C_E_TRANSACTION`	An API is called when the driver is not in valid state. One of the following cases: • `I2c_SetupEb`, `I2c_MasterWrite`, `I2c_MasterRead`, `I2c_SlaveAwaitRequest`, `I2c_ChangeOvs`, `I2c_ChangeSlaveAddress` is called when the driver is not in IDLE state. • `I2c_InitHwUnit` is called when the driver is not in UNINIT state. • `I2c_DeinitHwUnit` is called when the driver is in UNINIT state.

Error	Description
I2C_E_OS_TIME_REFUSED	GetCounterValue or GetElapsedValue (OS reference functions) reports an error.
I2C_E_PARAM_CONFIG	I2c_Init is called with an invalid parameter (the configuration structure is not found in the configuration set).
I2C_E_PARAM_CHANNEL	An API is called with incorrect channel ID (channel ID not found in the configuration set).
I2C_E_PARAM_POINTER	An API is called with an invalid pointer. One of the following cases: • I2c_SetupEb and I2c_UpdateTxBuffer are called with NULL pointers. • I2c_GetBufferStatus and I2c_GetVersionInfo are called with a NULL pointer.
I2C_E_PARAM_LENGTH	An API (I2c_SetupEb, I2c_UpdateTxBuffer) is called with an invalid length (SrcSize or DstSize is 0 or greater than 65536u).
I2C_E_PARAM_OVSID	I2c_ChangeOvs is called with an invalid OVS ID (OVS ID not found in the configuration set).
I2C_E_PARAM_ADDRESS_MATCHING	I2c_ChangeSlaveAddress is called with an invalid combination of address and address mask (the value of address is equal to 0, and the address mask is equal to 0xFF; this combination would result in only accepting the “general call”).
I2C_E_PARAM_POINTER_AND_LENGTH	One of the following cases: I2c_SetupEb is called with an invalid combination of length and pointer. • Length (SrcSize/DstSize) is 0, but the pointer (SrcPtr/DstPtr) is a valid address. • Pointer is NULL, but the length is valid. I2c_MasterWrite is called with invalid parameters: • Previously set SrcSize is equal to 0 or SrcPtr is NULL. I2c_MasterRead is called with invalid parameters: • Previously set DstSize is equal to 0 or DstPtr is NULL. I2c_SlaveAwaitRequest is called with invalid parameters: • Previously set SrcSize or DstSize is equal to 0 or SrcPtr or DstPtr is NULL.

5.6 Production errors

There are two types of production errors: recoverable failure and unrecoverable failure.

These errors are reported to the DEM module with the category name, and to the error hook (configured via I2cErrorCalloutFunction) with the detailed error code.

5.6.1 Recoverable failure

These are temporary errors, which are cleared when the operation is retried.

- Bus protocol error (NACK, ARB_LOST, unintended STOP from external master)

- Rx FIFO handling error (OVER_FLOW)

`I2C_E_HW_NACK_ERROR`: NACK received from an external node.

Note: In the master mode (write direction) of the repeated start mode, if the last acknowledgement is delayed due to clock stretching on the slave node, reception of a NACK may not be detected. So, it is possible the driver does not report the NACK via the Error report feature. However, note that in any case, the corresponding last byte is already sent.
If the Tx data length is 1 byte (Master Write transaction) and depending on the interrupt handling timing, there is a possibility to ignore the NACK reception of a slave address. In this case, the driver does not report the NACK via the Error report feature.

Note: NACK may not be detected if the interrupt delay or the `I2c_MainFunction_Handling` call cycle is long in master mode.

`I2C_E_HW_ARB_LOST_ERROR`: The I2C driver lost bus arbitration.

`I2C_E_HW_RX_OVERFLOW_ERROR`: Rx FIFO overflow. This occurs when the periodic process is slower than the bus transaction speed.

5.6.2 Unrecoverable failure

These errors are typically caused by a hardware failure; if retried, the error may occur again.

- Bus error
- Tx FIFO handling error (OVER_FLOW)
- Rx FIFO handling error (UNDER_FLOW)
- DMA error

Table 15 Unrecoverable failure

Error	Description
`I2C_E_HW_BUS_ERROR`	SCB detected an I2C bus error
`I2C_E_HW_TX_OVERFLOW_ERROR`	Tx FIFO overflow
`I2C_E_HW_RX_UNDERFLOW_ERROR`	Rx FIFO underflow
`I2C_E_HW_DMA_SRC_BUS_ERROR`	DMA source bus error
`I2C_E_HW_DMA_DST_BUS_ERROR`	DMA destination bus error
`I2C_E_HW_DMA_SRC_MISAL_ERROR`	DMA source buffer misaligned
`I2C_E_HW_DMA_DST_MISAL_ERROR`	DMA destination buffer misaligned
`I2C_E_HW_DMA_CURR_PTR_NULL_ERROR`	Tried to activate DMA with NULL pointer
`I2C_E_HW_DMA_CH_DISABLED_ERROR`	DMA channel disabled
`I2C_E_HW_DMA_DESCR_BUS_ERROR`	DMA descriptor bus error

Note: If the slave write operation occurs an unrecoverable failure, the external master may detect a bus error.

5.7 Reentrancy

All services except `I2c_Init`, `I2c_DeInit`, `I2c_InitHwUnit`, `I2c_DeinitHwUnit` and `I2c_MainFunction_Handling` are reentrant if they are executed with different channel IDs (`I2c_GetVersionInfo` is always reentrant).

5.8 Sleep mode

The I2C driver does not provide a dedicated sleep mode.

Note: All I2C sequences must be completed or stopped before entering deep sleep mode. I2C operation in deep sleep mode is not guaranteed.

5.9 Debugging support

The I2C driver does not support debugging.

5.10 Execution-time dependencies

The execution time of the API functions depends on certain factors listed in Table 16.

Table 16 Execution-time dependencies

Affected function	Dependency
`I2c_Init()`, `I2c_DeInit()`	Number of configured hardware units
`I2c_MainFunction_Handling()`, (Interrupt)	Number of configured hardware units, trigger level setting, DMA usage, master or slave operation
`I2c_MasterWrite()`, `I2c_MasterRead()`	Trigger level setting, DMA usage, send/receive data length, and bus idle check
`I2c_SlaveAwaitRequest()`, `I2c_GetBufferStatus()`	Trigger level setting, DMA usage, send/receive data length
`I2c_Cancel()`	Length of remaining data and transaction status
`I2c_UpdateTxBuffer()`	Trigger-level setting, DMA usage, send data length, driver status, and bus status

5.11 Deviation from AUTOSAR

I2C is not defined in AUTOSAR. Thus, there is no specific requirement about where the I2C driver deviates from.

6 Hardware resources

6.1 Ports and pins

The I2C driver uses the SCB instances of the TRAVEO™ T2G family microcontrollers. The pins listed in Table 17 are used. Make sure that the pins are correctly set in the PORT driver's configuration.

Table 17 Pins for I2C operation

Pin name	Direction	Description
SCB<n>_I2C_SDA	Output	SCB channel <*n*> I2C data pin
SCB<n>_I2C_SCL	Output	SCB channel <*n*> I2C clock pin

6.2 Timer

The I2C driver does not use any hardware timers directly (An OS timer is referenced).

6.3 Interrupts

The interrupt services listed in Table 18 must be configured correctly for peripherals used by the I2C driver. If a peripheral is not used, the corresponding interrupt service must not be present in the configuration.

Table 18 IRQ vectors and ISR names

IRQ vector	ISR name Cat1	ISR name Cat2
SCB<n> interrupt request	I2c_Interrupt_SCB<*n*>_Cat1	I2c_Interrupt_SCB<*n*>_Cat2
DMA completion interrupt request ch.<i> for TX	I2c_Interrupt_DMA_CH<*i*>_Isr_Cat1	I2c_Interrupt_DMA_CH<*i*>_Isr_Cat2
DMA completion interrupt request ch.<j> for RX	I2c_Interrupt_DMA_CH<*j*>_Isr_Cat1	I2c_Interrupt_DMA_CH<*j*>_Isr_Cat2

Note: The OS must associate the named ISRs with the corresponding SCB interrupt. For example, if the hardware unit SCB ch.2 is configured, `I2c_Interrupt_SCB2_Cat2()` must be called from the (OS-)interrupt service routine of the SCB ch.2 interrupt. For category1 usage, the address of `I2c_Interrupt_SCB2_Cat1()` must be the entry for the SCB ch.2 interrupt in the (OS) interrupt vector table. DMA completion ISRs are generated only if the given DMA channel is used by an SCB channel configured to I2C. If an SCB channel uses DMA, the interrupt handlers for SCB are required in addition to the DMS completion ISRs.

Note: The DMA interrupt priority must be higher than the SCB interrupt priority.

Note: If the I2C interrupt priority is too low, FIFO access may be inhibited by other interrupts. This may cause FIFO overflow or underflow, and unintended behavior (especially during the repeated start operation). Thus, you should ensure an appropriate priority of the I2C interrupts.

Note: On the Arm® Cortex®-M4 CPU, priority inversion of interrupts may occur under specific timing conditions in the integrated system with I2C. For more details, see the following errata notice.

Arm® Cortex®-M4 Software Developers Errata Notice - 838869:

“Store immediate overlapping exception return operation might vector to incorrect interrupt”

If the user application cannot tolerate the priority inversion, a DSB instruction should be added at the end of the interrupt function to avoid the priority inversion.

I2C interrupts are handled by an ISR wrapper (handler) in the integrated system. Thus, if necessary, the DSB instruction should be added just before the end of the handler by the integrator.

6.4 DMA

The I2C driver uses DMA channels. The DMA channels can be configured by the user and are enabled or disabled by the I2C driver as required. The DMA hardware itself must be enabled by the user before the I2C driver uses them for DMA transfers. The conditions under which DMA transfer operates vary depending on the specified external buffer size and each mode. See 5.1.2.3 Using DMA, 5.1.3.3 Using DMA and 5.1.4.1.3 Using DMA.

When using DMA, ensure that “one-to-one trigger multiplexer” is correctly configured in the PORT driver’s configuration. In addition, ensure proper memory allocation. See 2.6.2 Memory allocation and constraints.

7 Appendix A

7.1 Include files

The *I2c.h* file is the only file that needs to be included to use the functions of the I2C driver.

7.2 Data types

7.2.1 I2c_ChannelIdType

Type

`uint8`

Description

`I2c_ChannelIdType`: Channel ID

Range of values from 0 to <number of Channels-1>

7.2.2 I2c_BufferType

Type

`uint8`

Description

`I2c_BufferType`: Type of external buffer elements

7.2.3 I2c_BufferSizeType

Type

`uint32`

Description

`I2c_BufferSizeType`: Size of the external buffer as the number of data elements of the `I2c_BufferType` type

7.2.4 I2c_OvsIdType

Type

`uint8`

Description

`I2c_OvsIdType`: OVS settings ID

7.2.5 I2c_SlaveAddressType

Type

`uint8`

Description

This type is used for the slave address and slave address mask.

7.2.6 I2c_ChannelStatusType

Type

```
typedef enum
{
  I2C_UNKNOWN_STATUS   = 0,
  I2C_UNINIT           = 1,
  I2C_IDLE             = 2,
  I2C_TX               = 3,
  I2C_RX               = 4,
  I2C_BUSY_MASTER      = 16,
  I2C_BUSY_MASTERTX    = (I2C_TX + I2C_BUSY_MASTER),
  I2C_BUSY_MASTERRX    = (I2C_RX + I2C_BUSY_MASTER),
  I2C_BUSY_SLAVE       = 32,
  I2C_BUSY_SLAVETX     = (I2C_TX + I2C_BUSY_SLAVE),
  I2C_BUSY_SLAVERX     = (I2C_RX + I2C_BUSY_SLAVE)
} I2c_ChannelStatusType;
```

Description

`I2c_ChannelStatusType` : Status of the I2C driver. This datatype holds the I2C channel status and can be obtained by calling `I2c_GetStatus`.

7.2.7 I2c_JobResultType

Type

```
typedef enum
{
  I2C_UNKNOWN_RESULT,
  I2C_NORESULT,
  I2C_PENDING,
  I2C_CANCEL,
  I2C_MASTER_TX_SUCCESS,
  I2C_MASTER_RX_SUCCESS,
  I2C_SLAVE_TX_SUCCESS,
  I2C_SLAVE_RX_SUCCESS,
```

```
  I2C_MASTER_TX_ERROR,
  I2C_MASTER_RX_ERROR,
  I2C_SLAVE_TX_ERROR,
  I2C_SLAVE_RX_ERROR
} I2c_JobResultType;
```

Description

`I2c_JobResultType`: Job status of the I2C driver. This datatype holds the I2C job status and can be obtained by calling `I2c_GetJobResult`.

7.2.8 I2c_TransferDirectionType

Type

```
typedef enum
{
  I2C_WRITE,
  I2C_READ
} I2c_TransferDirectionType;
```

Description

`I2c_TransferDirectionType`: The kind of I2c transfer direction. This datatype is used to determine the I2C transfer direction in `I2c_SlaveAddressMatchNotification`.

Note: Since `I2c_SlaveAddressMatchNotification` is only called when `I2cHwAutoAckSlaveAddress` is disabled, this datatype can only be used when `I2cHwAutoAckSlaveAddress` is disabled.

7.2.9 I2c_AcknowledgeType

Type

```
typedef enum
{
  I2C_NACK,
  I2C_HOLDACK,
  I2C_ACK
} I2c_AcknowledgeType;
```

Description

`I2c_AcknowledgeType`: The kind of I2c transfer acknowledge. This datatype is used to determine the I2C transfer acknowledge in `I2c_SlaveAddressMatchNotification`.

Note: Since `I2c_SlaveAddressMatchNotification` is only called when `I2cHwAutoAckSlaveAddress` is disabled, this datatype can only be used when `I2cHwAutoAckSlaveAddress` is disabled.

7.2.10 I2c_SlaveCompleteEventType

Type

```
typedef enum
{
  /* STOP : Slave write greater than expected */
  I2C_SLAVE_COMPLETE_STOP_TX_GT,
  /* STOP : Slave write equal to expected */
  I2C_SLAVE_COMPLETE_STOP_TX_EQ,
  /* STOP : Slave write less than expected */
  I2C_SLAVE_COMPLETE_STOP_TX_LT,
  /* STOP : Slave read greater or equal expected */
  I2C_SLAVE_COMPLETE_STOP_RX_GE,
  /* STOP : Slave read less than expected */
  I2C_SLAVE_COMPLETE_STOP_RX_LT,
  /* RESTART : Slave write greater than expected */
  I2C_SLAVE_COMPLETE_RESTART_TX_GT,
  /* RESTART : Slave write equal to expected */
  I2C_SLAVE_COMPLETE_RESTART_TX_EQ,
  /* RESTART : Slave write less than expected */
  I2C_SLAVE_COMPLETE_RESTART_TX_LT,
  /* RESTART : Slave read greater or equal expected */
  I2C_SLAVE_COMPLETE_RESTART_RX_GE,
  /* RESTART : Slave read less than expected */
  I2C_SLAVE_COMPLETE_RESTART_RX_LT
} I2c_SlaveCompleteEventType;
```

Description

`I2c_SlaveCompleteEventType`: The kind of I2c transfer complete event. This datatype is used to determine the I2C transfer complete event in `I2c_SlaveCompleteNotification`.

Note: Since `I2c_SlaveCompleteNotification` is only called when `I2cHwAutoAckSlaveAddress` is disabled, this datatype can only be used when `I2cHwAutoAckSlaveAddress` is disabled.

7.2.11 I2c_ConfigType

Type

```
typedef struct
{
  P2CONST(I2c_ScbChannelConfigType, TYPEDEF, TYPEDEF)
I2c_ScbChannelConfigsPtr;
  P2CONST(I2c_ChannelIdType, TYPEDEF, TYPEDEF) I2c_ChannelIdListPtr;
```

```
  CONST(uint8, TYPEDEF) NumberOfChannel;
} I2c_ConfigType;
```

Description

The type of the external data structure containing the initialization data for the I2C driver.

7.3 Constants

7.3.1 Error codes

A service may return one of the error codes, listed in Table 19, if default error detection is enabled.

Table 19 Error codes

Name	Value	Description
I2C_E_UNINIT	0x0A	No initialization done
I2C_E_ALREADY_INITIALIZED	0x0B	Initialization is already done
I2C_E_TRANSACTION	0x0C	Not called in IDLE status
I2C_E_OS_TIME_REFUSED	0x0D	OS reference function returned an error code
I2C_E_PARAM_CONFIG	0x10	Configuration pointer out of range
I2C_E_PARAM_CHANNEL	0x11	Channel ID out of range
I2C_E_PARAM_POINTER	0x12	Pointer out of range
I2C_E_PARAM_LENGTH	0x13	Length out of range
I2C_E_PARAM_OVSID	0x14	OVS ID out of range
I2C_E_PARAM_ADDRESS_MATCHING	0x15	Address/address mask out of range
I2C_E_PARAM_POINTER_AND_LENGTH	0x16	Pointer and length combination invalid
I2C_E_HW_NACK_ERROR	0x20	NACK received
I2C_E_HW_ARB_LOST_ERROR	0x21	Arbitration lost
I2C_E_HW_BUS_ERROR	0x22	Bus error
I2C_E_HW_TX_OVERFLOW_ERROR	0x23	TX FIFO overflow
I2C_E_HW_RX_OVERFLOW_ERROR	0x25	RX FIFO overflow
I2C_E_HW_RX_UNDERFLOW_ERROR	0x26	RX FIFO underflow
I2C_E_HW_DMA_SRC_BUS_ERROR	0x29	Internal bus error in source DMA
I2C_E_HW_DMA_DST_BUS_ERROR	0x2A	Internal bus error in destination DMA
I2C_E_HW_DMA_SRC_MISAL_ERROR	0x2B	DMA source buffer misaligned
I2C_E_HW_DMA_DST_MISAL_ERROR	0x2C	DMA destination buffer misaligned
I2C_E_HW_DMA_CURR_PTR_NULL_ERROR	0x2D	Current DMA pointer is NULL
I2C_E_HW_DMA_CH_DISABLED_ERROR	0x2E	DMA channel disabled
I2C_E_HW_DMA_DESCR_BUS_ERROR	0x2F	A bus error occurred when loading the descriptor

7.3.2 Version information

Table 20 Version information

Name	Value	Description
I2C_SW_MAJOR_VERSION	See release notes	Vendor-specific major version number
I2C_SW_MINOR_VERSION	See release notes	Vendor-specific minor version number
I2C_SW_PATCH_VERSION	See release notes	Vendor-specific patch version number

7.3.3 Module information

Table 21 Module information

Name	Value	Description
I2C_MODULE_ID	255	Module ID
I2C_VENDOR_ID	66	Vendor ID

7.3.4 API service IDs

Table 22 lists the API service IDs used when reporting errors via DET or via the error callout function.

Table 22 API service IDs

Name	Value	API name
I2C_API_INIT	0x00	I2c_Init
I2C_API_DEINIT	0x01	I2c_DeInit
I2C_API_GET_STATUS	0x02	I2c_GetStatus
I2C_API_GET_JOB_RESULT	0x03	I2c_GetJobResult
I2C_API_CANCEL	0x04	I2c_Cancel
I2C_API_MASTER_WRITE	0x05	I2c_MasterWrite
I2C_API_MASTER_READ	0x06	I2c_MasterRead
I2C_API_SLAVE_AWAIT_REQUEST	0x07	I2c_SlaveAwaitRequest
I2C_API_SETUP_EB	0x08	I2c_SetupEb
I2C_API_MAINFUNCTION_HANDLING	0x09	I2c_MainFunction_Handling
I2C_API_GET_BUFFER_STATUS	0x0A	I2c_GetBufferStatus
I2C_API_CHANGE_OVS	0x0B	I2c_ChangeOvs
I2C_API_CHANGE_SLAVE_ADDRESS	0x0C	I2c_ChangeSlaveAddress
I2C_API_GET_VERSION_INFO	0x0D	I2c_GetVersionInfo
I2C_API_INTERRUPT_SCB	0x0E	I2c_Interrupt_SCB<n>_Cat1, I2c_Interrupt_SCB<n>_Cat2
I2C_API_INTERRUPT_DMA	0x0F	I2c_Interrupt_DMA_CH<m>_Isr_Cat1, I2c_Interrupt_DMA_CH<m>_Isr_Cat2
I2C_API_SET_REPEATEDSTART	0x10	I2c_SetRepeatedStart
I2C_API_GET_REPEATEDSTART	0x11	I2c_GetRepeatedStart
I2C_API_CONFIRM_TXTRANSACTION	0x12	I2c_ConfirmTxTransaction
I2C_API_UPDATE_TX_BUFFER	0x13	I2c_UpdateTxBuffer

Name	Value	API name
I2C_API_SLAVE_START_TRANSFER	0x14	I2c_SlaveStartTransfer
I2C_API_INIT_HW_UNIT	0x15	I2c_InitHwUnit
I2C_API_DEINIT_HW_UNIT	0x16	I2c_DeinitHwUnit

7.4 Functions

7.4.1 I2c_Init

Syntax

```
FUNC(void, I2C_CODE) I2c_Init
(
  P2CONST(I2c_ConfigType, AUTOMATIC, I2C_APPL_CONST) ConfigPtr
)
```

Service ID

0x00

Sync/Async

Sync

Reentrancy

Non-reentrant

Parameters (in)

- `ConfigPtr` – Pointer to a configuration

Parameters (out)

None

Return value

None

DET errors

- `I2C_E_ALREADY_INITIALIZED` – Driver already initialized
- `I2C_E_PARAM_CONFIG` – Invalid pointer

DEM errors

None

Description

This function initializes all local data for the configured channels. The driver state will be set to `I2C_IDLE`, and all job results will be set to `I2C_NORESULT`.

7.4.2 I2c_DeInit

Syntax

```
FUNC(void, I2C_CODE) I2c_DeInit(void)
```

Service ID

0x01

Sync/Async

Sync

Reentrancy

Non-reentrant

Parameters (in)

None

Parameters (out)

None

Return value

None

DET errors

- `I2C_E_UNINIT` – Driver uninitialized

DEM errors

None

Description

Initializes all local data and registers to reset values. The driver state will be set to `I2C_UNINIT`, and all job results will be set to `I2C_NORESULT`.

This API can be used for force initialize the HW regardless the SW state. If there is a case to hang the communication, you can recover the I2C HW to default state with this API.

7.4.3 I2c_GetStatus

Syntax

```
FUNC(I2c_ChannelStatusType, I2C_CODE) I2c_GetStatus
(
  const I2c_ChannelIdType ChannelId
)
```

Service ID

0x02

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID

Parameters (out)

None

Return value

- `I2C_UNINIT`: Not yet initialized
- `I2C_IDLE`: No transaction request
- `I2C_BUSY_MASTERTX`: Ongoing master write operation
- `I2C_BUSY_MASTERRX`: Ongoing master read operation
- `I2C_BUSY_SLAVE`: Waiting for external master request
- `I2C_BUSY_SLAVETX`: Ongoing slave write operation
- `I2C_BUSY_SLAVERX`: Ongoing slave read operation
- `I2C_UNKNOWN_STATUS`: Cannot return the status (invalid channel ID)

DET errors

- `I2C_E_PARAM_CHANNEL` – Invalid channel ID

DEM errors

None

Description

Returns the current driver/channel status

7.4.4 I2c_GetJobResult

Syntax

```
FUNC(I2c_JobResultType, I2C_CODE) I2c_GetJobResult
(
  const I2c_ChannelIdType ChannelId
)
```

Service ID

0x03

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID

Parameters (out)

None

Return value

- `I2C_NORESULT`: Initial status, there is no job yet.
- `I2C_PENDING`: Ongoing job
- `I2C_CANCEL`: Previous job cancelled
- `I2C_MASTER_TX_SUCCESS`: Previous master Tx job success
- `I2C_MASTER_RX_SUCCESS`: Previous master Rx job success
- `I2C_SLAVE_TX_SUCCESS`: Previous slave Tx job success
- `I2C_SLAVE_RX_SUCCESS`: Previous slave Rx job success
- `I2C_MASTER_TX_ERROR`: Previous master Tx job failed
- `I2C_MASTER_RX_ERROR`: Previous master Rx job failed
- `I2C_SLAVE_TX_ERROR`: Previous slave Tx job failed
- `I2C_SLAVE_RX_ERROR`: Previous slave Rx job failed
- `I2C_UNKNOWN_RESULT`: Cannot return the result (channel ID invalid or channel uninitialized)

DET errors

- `I2C_E_UNINIT` – The driver is uninitialized.
- `I2C_E_PARAM_CHANNEL` – An invalid channel ID was specified.

DEM errors

None

Description

Returns the newest driver/channel job status

7.4.5 I2c_Cancel

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_Cancel
(
  const I2c_ChannelIdType ChannelId
)
```

Service ID

0x04

Sync/Async

Async

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID

Parameters (out)

None

Return value

`E_OK`: Accepted and completed

`E_NOT_OK`: Not completed or declined with error

DET errors

- `I2C_E_UNINIT` – Driver uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID

DEM errors

None

Description

Terminates the current operation.

If the bus state does not allow an immediate stop, this function returns `E_NOT_OK` (without DET error). In such cases, this call should be retried until `E_OK` is returned.

The channel state will be set to `I2C_IDLE`, and the job result will be set to `I2C_CANCEL`.

7.4.6 I2c_MasterWrite

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_MasterWrite
(
  const I2c_ChannelIdType ChannelId,
  const I2c_SlaveAddressType SlaveAddr
)
```

Service ID

0x05

Sync/Async

Async

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID

- `SlaveAddr` – Target slave address (bit 7–bit 1 is used, bit 0 is ignored)

Parameters (out)

None

Return value

- `E_OK`: Request accepted
- `E_NOT_OK`: Request declined

DET errors

- `I2C_E_UNINIT` – Driver uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID
- `I2C_E_TRANSACTION` – Driver/channel not in IDLE state
- `I2C_E_PARAM_POINTER_AND_LENGTH` – Invalid pointer or length set with `I2c_SetupEb`.

DEM errors

None

Description

Starts the master write operation.

This function sets the SCB registers and local data. If DMA was configured to perform a periodic process, the DMA channel is also set in this function. The external “Src" buffer is referenced, and the stored data is used for transmission.

If the `I2cBusIdleCheck` is enabled, and the bus state is busy, this function returns `E_NOT_OK` without DET error. In such cases, this call should be retried until `E_OK` is returned.

The channel state will be set to `I2C_BUSY_MASTERTX`, and the job result will be set to `I2C_PENDING`.

Note: This bus idle check is not applied on Repeated Start mode.

7.4.7 I2c_MasterRead

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_MasterRead
(
  const I2c_ChannelIdType ChannelId,
  const I2c_SlaveAddressType SlaveAddr
)
```

Service ID

0x06

Sync/Async

Async

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID
- `SlaveAddr` – Target slave address (bit 7–bit 1 is used, bit 0 is ignored)

Parameters (out)

None

Return value

- `E_OK`: Request accepted
- `E_NOT_OK`: Request declined

DET errors

- `I2C_E_UNINIT` – Driver uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID
- `I2C_E_TRANSACTION` – Driver/channel not in IDLE state
- `I2C_E_PARAM_POINTER_AND_LENGTH` – Invalid pointer or length set with I2c_SetupEb

DEM errors

None

Description

Starts the master read operation.

This function sets the SCB registers and local data. If DMA was configured for periodic processes, the DMA channel is also set in this function. The external “Dst” buffer is used to store received data.

If `I2cBusIdleCheck` is enabled, and the bus state is busy, this function returns `E_NOT_OK` without DET error. In such cases, you should retry until `E_OK` is returned. The channel state will be set to `I2C_BUSY_MASTERRX`, and the job result will be set to `I2C_PENDING`.

Note: This bus idle check is not applied on repeated start mode.

7.4.8 I2c_SlaveAwaitRequest

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_SlaveAwaitRequest
(
  const I2c_ChannelIdType ChannelId
)
```

Service ID

0x07

Sync/Async

Async

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID

Parameters (out)

None

Return value

`E_OK`: Request accepted

`E_NOT_OK`: Request declined

DET errors

- `I2C_E_UNINIT` – Driver uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID
- `I2C_E_TRANSACTION` – Driver/channel not in IDLE state
- `I2C_E_PARAM_POINTER_AND_LENGTH` – Invalid pointer or length was set with `I2c_SetupEb`.

DEM errors

None

Description

Starts the slave operation.

The operation to start (Tx or Rx) depends on the external master request. This function prepares the SCB for both operations (Tx and Rx). If DMA was configured to periodic process, the DMA channel is also set in this function. The external “Src” buffer is referenced, and the stored data is used for transmission. The external “Dst” buffer is used to store the received data.

The channel state will be set to `I2C_BUSY_SLAVE`, and the job result will be set to `I2C_PENDING`. The state will be changed to `I2C_BUSY_SLAVETX` or `I2C_BUSY_SLAVERX` in the periodic process.

7.4.9 I2c_SetupEb

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_SetupEb
(
  const I2c_ChannelIdType ChannelId,
  P2VAR(I2c_BufferType, AUTOMATIC, I2C_APPL_DATA) SrcPtr,
  const I2c_BufferSizeType SrcSize,
  P2VAR(I2c_BufferType, AUTOMATIC, I2C_APPL_DATA) DstPtr,
  const I2c_BufferSizeType DstSize
)
```

Service ID

0x08

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID
- `SrcPtr` – External buffer address for transmit
- `SrcSize` – Transmit data length in bytes (0 to 65536)
- `DstPtr` – External buffer address for receive
- `DstSize` – Receive data length in bytes (0 to 65536)

Parameters (out)

None

Return value

- `E_OK`: Request accepted
- `E_NOT_OK`: Request declined

DET errors

- `I2C_E_UNINIT` – Driver uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID
- `I2C_E_TRANSACTION` – Driver/channel not in IDLE state
- `I2C_E_PARAM_POINTER` – Invalid pointer
- `I2C_E_PARAM_LENGTH` – Invalid length
- `I2C_E_PARAM_POINTER_AND_LENGTH` – Invalid combination of pointer and length

DEM errors

None

Description

Saves the buffer information for subsequent operations.

The saved data will be used repeatedly in subsequent operations until this request is invoked again. `SrcPtr` and `SrcSize` are used for data transmission. The transmit data will be read from the `SrcPtr` buffer; the total transmit length is specified as `SrcSize`.

`DstPtr` and `DstSize` are used to receive the data. The received data will be stored into the `DstPtr` buffer; the total buffer length is specified as `DstSize`.

Note: If you want to use the same external buffer repeatedly, you can omit calling this function for each transaction. (only once is needed).
If you omit this call, the same buffer address (specified by `SrcPtr` and `DstPtr`) and size (specified by `SrcSize` and `DstSize`) are reused, both in the first and the following operations. It means that the

same data will be transmitted from the `SrcPtr` buffer and received data will be overwritten in the `DstPtr` buffer.

If only used for master mode, the unused parameters can be set as NULL and 0. (For example, if the driver is used only for master write operations, `DstPtr` can be NULL, `DstSize` can be 0.)

However, inconsistent data (for example, a NULL pointer and a valid size or a valid pointer and a size of 0) is not allowed. Because slave mode needs both buffers, these cannot be set to NULL or 0.

The channel state and job result will not be changed.

Note: Keep the external buffer area while the transmit/receive operation is ongoing. Do not access the external buffer while the transmit/receive operation is ongoing.

7.4.10 I2c_GetBufferStatus

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_GetBufferStatus
(
  const I2c_ChannelIdType ChannelId,
  P2VAR(P2VAR(I2c_BufferType, AUTOMATIC, I2C_APPL_DATA), AUTOMATIC,
  I2C_APPL_DATA) SrcPtrPtr,
  P2VAR(P2VAR(I2c_BufferType, AUTOMATIC, I2C_APPL_DATA), AUTOMATIC,
  I2C_APPL_DATA) DstPtrPtr,
  P2VAR(I2c_BufferSizeType, AUTOMATIC, I2C_APPL_DATA)
  SrcRemainingLengthPtr,
  P2VAR(I2c_BufferSizeType, AUTOMATIC, I2C_APPL_DATA)
  DstRemainingLengthPtr
)
```

Service ID

0x0A

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID

Parameters (out)

- `SrcPtrPtr` – Pointer to the location where the current `SrcPtr` address is written (next address using for Tx)

- `DstPtrPtr` – Pointer to the location where the current `DstPtr` address is written (next address using for Rx)
- `SrcRemainingLengthPtr` – Pointer to the location where the length of remaining data to transmit is written
- `DstRemainingLengthPtr` – Pointer to the location where the length of remaining data to receive is written

Return value

- `E_OK`: Request accepted
- `E_NOT_OK`: Request declined

DET errors

- `I2C_E_UNINIT` – Driver uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID
- `I2C_E_PARAM_POINTER` – Given pointer is pointed to NULL_PTR.

DEM errors

None

Description

Gives the current job's buffer status. (buffer address and length)

If the job is ongoing, the buffer status will be changed by the periodic process.

For example, if a master write operation is ongoing, the address of the referenced external buffer (`SrcPtrPtr`) and the remaining data length (`SrcRemainingLengthPtr`) for sending will be changed by each periodic process (such as interrupts). If the output values are not changed in a sufficiently long interval, the transaction is suspended by an external node or stopped because of a detected error. Thus, in addition to receiving error notifications, you can observe the transaction progress to determine the transaction status.

Note: The required interval depends on the type of periodic process and the SCB/Bus frequency. For example, if polling is selected, the output value will not change until the scheduled function is called. The channel state and job result will not be changed.

Note: Do not rely on this API value if it is not in the transaction direction (Tx or Rx) of the target job. For example, do not rely on the remaining DST data length of the slave Tx transaction. This is because, when the Tx transaction is in progress, the remaining length of the external SRC buffer can be calculate with accuracy, but the external DST buffer remaining length cannot be handled well.

Note: When an error is detected or a job is canceled, this API value may not be accurate.

7.4.11 I2c_ChangeOvs

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_ChangeOvs
(
  const I2c_ChannelIdType ChannelId,
  const I2c_OvsIdType OvsId
)
```

Service ID

0x0B

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID
- `OvsId` – Ovs settings ID

Parameters (out)

None

Return value

- `E_OK`: Request accepted
- `E_NOT_OK`: Request declined

DET errors

- `I2C_E_UNINIT` – Driver uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID
- `I2C_E_TRANSACTION` – Driver/channel not in IDLE state
- `I2C_E_PARAM_OVSID` – Invalid OVS settings ID

DEM errors

None

Description

This operation changes the current OVS settings.

The I2C driver uses the default OVS settings for initialization. The default OVS settings are specified with the configuration, these values are related to the MCU clock settings. However, the MCU clock settings can be changed by the MCU functionality at run time. In such cases, by using this service, you can continue to use the I2C driver even if the SCB clock was changed.

Note: You should ensure proper configuration of the OVS settings and select the proper OVS settings. There is no check for coherency between the SCB clock and the OVS settings. After reinitialization, the default settings will be used again. The channel state and the job result will not be changed.

7.4.12 I2c_ChangeSlaveAddress

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_ChangeSlaveAddress
(
  const I2c_ChannelIdType ChannelId,
  I2c_SlaveAddressType Address,
  I2c_SlaveAddressType Mask
)
```

Service ID

0x0C

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID
- `Address` – Slave address
- `Mask` – Slave address mask

Parameters (out)

None

Return value

- `E_OK`: Request accepted
- `E_NOT_OK`: Request declined

DET errors

- `I2C_E_UNINIT` – Driver is uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID
- `I2C_E_TRANSACTION` – Driver/channel not in IDLE state
- `I2C_E_PARAM_ADDRESS_MATCHING` - Invalid address and mask combination

DEM errors

None

Description

This service changes the slave address setting which is used to accept messages when the I2C driver is in slave mode.

The default slave address/slave address mask setting is specified by the configuration. After reinitialization, the default settings will be used again. The channel state and the job result will not be changed.

7.4.13 I2c_GetVersionInfo

Syntax

```
FUNC(void, I2C_CODE) I2c_GetVersionInfo
(
  P2VAR(Std_VersionInfoType, AUTOMATIC, I2C_APPL_DATA) VersionInfo
)
```

Service ID

0x0D

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

None

Parameters (out)

- `Versioninfo` – Pointer to the location where the version information will be written

Return value

None

DET errors

- `I2C_E_PARAM_POINTER` – Version info is NULL pointer.

DEM errors

None

Description

This function returns the version information of this module. This includes module ID, vendor ID, and vendor-specific version numbers.

7.4.14 I2c_SetRepeatedStart

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_SetRepeatedStart
(
  const I2c_ChannelIdType ChannelId,
  const boolean RepeatedFlag
)
```

Service ID

0x10

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID
- `RepeatedFlag` – Repeated Start Mode (`TRUE`: Repeated Start mode, `FALSE`: Normal mode)

Parameters (out)

None

Return value

- `E_OK`: Request accepted
- `E_NOT_OK`: Request declined

DET errors

- `I2C_E_UNINIT` – Driver is uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID
- `I2C_E_TRANSACTION` – Driver/channel not in IDLE state

DEM errors

None

Description

This function changes the repeated start mode of this module. In the repeated start mode, Master job will not send the STOP bit at the end of the transaction. Instead, the driver sends the repeated start bit instead of the START bit in the next API call. In the repeated start mode, Slave job will invoke the additional callback if it detects the repeated start bit. In this callback, set the buffer for the following transaction.

Once you set the repeated start mode, it will be continued until calling deInit/deinitHwUnit function or changing the mode by calling this API again.

7.4.15 I2c_GetRepeatedStart

Syntax

```
FUNC(boolean, I2C_CODE) I2c_GetRepeatedStart
(
  const I2c_ChannelIdType ChannelId
)
```

Service ID

0x11

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID

Parameters (out)

None

Return value

- `TRUE`: Driver is in Repeated Start mode
- `FALSE`: Driver is in Normal mode

DET errors

- `I2C_E_UNINIT` – Driver is uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID

DEM errors

None

Description

Returns the current driver mode (Repeated start mode or normal mode).

7.4.16 I2c_ConfirmTxTransaction

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_ConfirmTxTransaction
(
  const I2c_ChannelIdType ChannelId
)
```

Service ID

0x12

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID

Parameters (out)

None

Return value

- `E_OK`: Tx transaction is ended
- `E_NOT_OK`: Tx transaction is not ended

DET errors

- `I2C_E_UNINIT` – Driver is uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID

DEM errors

None

Description

Returns the Tx transaction is ended or not. This confirmation is required after the MasterWrite transaction in repeated start mode.

7.4.17 I2c_UpdateTxBuffer

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_UpdateTxBuffer
(
  const I2c_ChannelIdType ChannelId,
  P2VAR(I2c_BufferType, AUTOMATIC, I2C_APPL_DATA) SrcPtr,
  const I2c_BufferSizeType SrcSize
)
```

Service ID

0x13

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID
- `SrcPtr` – External buffer address for transmit
- `SrcSize` – Transmit data length in bytes (0 to 65536)

Parameters (out)

None

Return value

- `E_OK`: Request accepted
- `E_NOT_OK`: Request declined

DET errors

- `I2C_E_UNINIT` – Driver uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID
- `I2C_E_PARAM_POINTER` – Invalid pointer
- `I2C_E_PARAM_LENGTH` – Invalid length

DEM errors

None

Description

This API updates the Tx buffer set by `I2c_SetupEb`. This API is accepted in the following cases:

- `I2c_SlaveAwaitRequest` was called but did not start the I2c transaction.
- During the callback `I2c_SlaveSrNotification` on repeated start mode, and before calling the `I2c_SlaveAwaitRequest`.

This API declines the update request when the bus transaction is in progress.

> *Note: If `I2cHwAutoAckSlaveAddress` and/or `I2cHwAutoAckSlaveRxData` is enabled, you can reduce the bus latency to acknowledge. However, if the driver's interrupt is disturbed by some reason (for example, critical section), the control flow to the bus may lost. You should carefully enable this configuration, depending on the use case. see 5.1.4.3 for more detail.*

If the request is accepted (API returns `E_OK`), the Tx buffer stored in the driver is updated. If a new transaction starts, the data in the new Tx buffer is transmitted.

If the request is declined (API returns `E_NOT_OK`), the Tx buffer is not updated with a new one. If a new transaction starts, the data in the stored buffer (previous setup buffer) is transmitted.

> *Note: The constraints and usage of the buffer to be updated are the same as the buffer to be updated with `I2c_SetUpEb`.*

7.4.18 I2c_SlaveStartTransfer

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_SlaveStartTransfer
(
  const I2c_ChannelIdType ChannelId
);
```

Service ID

0x14

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `ChannelId` – Channel ID

Parameters (out)

None

Return value

- `E_OK`: Succeeded to start slave transfer
- `E_NOT_OK`: Failed to start slave transfer

DET errors

- `I2C_E_UNINIT` – Driver uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID
- `I2C_E_TRANSACTION` – Invalid configuration and call timing

DEM errors

None

Description

This API is used to restart the slave transfer when:

After the I2C bus enters the clock stretching state due to `I2c_SlaveAddressMatchNotification` returns `I2C_HOLDACK`, and ensure that `I2c_UpdateTxBuffer` has been called before this function is used.

7.4.19 I2c_InitHwUnit

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_InitHwUnit
(
  const I2c_ChannelIdType ChannelId
);
```

Service ID

0x15

Sync/Async

Sync

Reentrancy

Non-reentrant

Parameters (in)

- `ChannelId` – Channel ID

Parameters (out)

None

Return value

- `E_OK`: Succeeded to initializes the SW and HW for the specified SCB channel
- `E_NOT_OK`: Failed to initializes the SW and HW for the specified SCB channel

DET errors

- `I2C_E_UNINIT` – Driver uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID
- `I2C_E_TRANSACTION` – Invalid call timing

DEM errors

None

Description

This API is used to re-initializes the SW and HW for the specified SCB channel.

Note: After calling this API, HW/SW on the specified channel will be in the same state as after calling `I2c_Init()`. In other words, dynamically set values such as Repeated Start Mode will be initialized, so please reconfigure them before starting communication if necessary.

7.4.20 I2c_DeinitHwUnit

Syntax

```
FUNC(Std_ReturnType, I2C_CODE) I2c_DeinitHwUnit
(
  const I2c_ChannelIdType ChannelId
);
```

Service ID

0x16

Sync/Async

Sync

Reentrancy

Non-reentrant

Parameters (in)

- `ChannelId` – Channel ID

Parameters (out)

None

Return value

- `E_OK`: Succeeded to de-initializes the SW and HW for the specified SCB channel
- `E_NOT_OK`: Failed to de-initializes the SW and HW for the specified SCB channel

DET errors

- `I2C_E_UNINIT` – Driver uninitialized
- `I2C_E_PARAM_CHANNEL` – Invalid channel ID
- `I2C_E_TRANSACTION` – Invalid call timing

DEM errors

None

Description

This API is used to de-initializes the SW and HW for the specified SCB channel.

Note: When this API is called, the specified channel will be forcibly shut down even if communication is in progress, do not call any APIs other than `I2c_GetStatus()`, `I2c_GetJobResult()`, and `I2c_GetVersionInfo()` for the specified channel until `I2c_InitHwUnit()` is called.

7.5 Scheduled functions

7.5.1 I2c_MainFunction_Handling

Syntax

```
FUNC(void, I2C_CODE) I2c_MainFunction_Handling(void)
```

Service ID

0x09

Sync/Async

Async

Reentrancy

Non-reentrant

Parameters (in)

None

Parameters (out)

None

Return value

None

DET errors

`I2C_E_UNINIT` – Driver is uninitialized.

DEM errors

`I2C_DEM_RECOVERABLE_FAILURE`:

- `I2C_E_HW_NACK_ERROR`: “NACK” received from an external node
- `I2C_E_HW_ARB_LOST_ERROR`: I2C driver lost the arbitration for the bus.
- `I2C_E_HW_RX_OVERFLOW_ERROR`: Rx FIFO overflow

`I2C_DEM_UNRECOVERABLE_FAILURE`:

- `I2C_E_HW_BUS_ERROR`: SCB detected an I2C bus error.
- `I2C_E_HW_TX_OVERFLOW_ERROR`: Tx FIFO overflow
- `I2C_E_HW_RX_UNDERFLOW_ERROR`: Rx FIFO underflow

Description

This function is used for polling.

This function progresses the jobs and operation of all configured channels. If a job has ended, the corresponding notification function will be called. Then the channel state will be set to `I2C_IDLE`, and the job result will be set according to the job’s outcome.

7.6 Interrupt service routine

7.6.1 I2c_Interrupt_SCB<n>_CatX

Syntax

`ISR_NATIVE(I2c_Interrupt_SCB<n>_Cat1) or`
`ISR(I2c_Interrupt_SCB<n>_Cat2)`

Service ID

0x0E

Sync/Async

Sync

Reentrancy

Non-reentrant

Parameters (in)

None

Parameters (out)

None

Return value

None

DET errors

None

DEM errors

- `I2C_DEM_RECOVERABLE_FAILURE`:
  - `I2C_E_HW_NACK_ERROR`: “NACK” received from an external node
  - `I2C_E_HW_ARB_LOST_ERROR`: I2C driver lost the arbitration for the bus
  - `I2C_E_HW_RX_OVERFLOW_ERROR`: Rx FIFO overflow
- `I2C_DEM_UNRECOVERABLE_FAILURE`:
  - `I2C_E_HW_BUS_ERROR`: SCB detected an I2C bus error
  - `I2C_E_HW_TX_OVERFLOW_ERROR`: Tx FIFO overflow
  - `I2C_E_HW_RX_UNDERFLOW_ERROR`: Rx FIFO underflow
  - `I2C_E_HW_DMA_SRC_BUS_ERROR`[1]: Source DMA detected an error.
  - `I2C_E_HW_DMA_DST_BUS_ERROR`[1]: Destination DMA detected an error
  - `I2C_E_HW_DMA_SRC_MISAL_ERROR`[1]: Source DMA buffer is misaligned

[1] This error causes both an SCB interrupt and a DMA interrupt.

- I2C_E_HW_DMA_DST_MISAL_ERROR[1]: Destination DMA buffer is misaligned
- I2C_E_HW_DMA_CURR_PTR_NULL_ERROR[1]: Current DMA pointer is NULL
- I2C_E_HW_DMA_CH_DISABLED_ERROR[1]: DMA channel is disabled
- I2C_E_HW_DMA_DESCR_BUS_ERROR[1]: A bus error occurred when loading the descriptor

Description

This function is an ISR.

This function progresses the jobs and operation of the affected channels. If a job has ended, the corresponding notification function will be called. Then the channel state will be set to I2C_IDLE, and the job result will be set according to the job's outcome.

7.6.2 I2c_Interrupt_DMA_CH<m>_Isr_CatY

Syntax

```
ISR_NATIVE(I2c_Interrupt_DMA_CH<m>_Isr_Cat1) or
ISR(I2c_Interrupt_DMA_CH<m>_Isr_Cat2)
```

Service ID

0x0F

Sync/Async

Sync

Reentrancy

Non-reentrant

Parameters (in)

None

Parameters (out)

None

Return value

None

DET errors

None

DEM errors

- I2C_DEM_RECOVERABLE_FAILURE:
  - I2C_E_HW_NACK_ERROR[1]: "NACK" received from an external node
  - I2C_E_HW_ARB_LOST_ERROR[2]: I2C driver lost arbitration for the bus
  - I2C_E_HW_RX_OVERFLOW_ERROR[2]: Rx FIFO overflow

[1] This error causes both SCB and DMA interrupts.

- I2C_DEM_UNRECOVERABLE_FAILURE:
  - I2C_E_HW_BUS_ERROR[2]: SCB detected an I2C bus error
  - I2C_E_HW_TX_OVERFLOW_ERROR[2]: Tx FIFO overflow
  - I2C_E_HW_RX_UNDERFLOW_ERROR[2]: Rx FIFO underflow
  - I2C_E_HW_DMA_SRC_BUS_ERROR: Source DMA detected an error.
  - I2C_E_HW_DMA_DST_BUS_ERROR: Destination DMA detected an error
  - I2C_E_HW_DMA_SRC_MISAL_ERROR: Source DMA buffer is misaligned
  - I2C_E_HW_DMA_DST_MISAL_ERROR: Destination DMA buffer is misaligned
  - I2C_E_HW_DMA_CURR_PTR_NULL_ERROR: Current DMA pointer is NULL
  - I2C_E_HW_DMA_CH_DISABLED_ERROR: DMA channel is disabled.
  - I2C_E_HW_DMA_DESCR_BUS_ERROR: A bus error occurred when loading the descriptor

Description

This function is an ISR.

This function performs the jobs and operation of the affected channels.

7.7 Required callback functions

7.7.1 I2C notification functions

The I2C driver uses the following callback routines to inform other software modules about certain states or state changes. These other modules are required to handle the conditions indicated by the callback routines.

All notification functions must be reentrant. Basically, I2C driver API calls are not allowed from callback functions. Some exceptions are described in each notification function's description.

Note: If the job is finished by cancellation (`I2c_Cancel` was called), the notification function will not be called.

However, if the Master job finishes without the slave sending or receiving a single data, the slave notification function (`I2c_SlaveTxNotification` or `I2c_SlaveRxNotification`) will be called.

7.7.1.1 I2c_MasterTxNotification

Syntax

```
void I2c_MasterTxNotification(uint8 Channel)
```

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

This notification is a user-provided callback routine to notify that a job has been finished.

It will be called at the end of a master write job.

If the driver is in the repeated start mode, this callback is only called after sending the STOP bit. This means, that, this callback is called only once in the one continuous transaction. And if the last transaction was a MasterWrite operation, then this callback is called.

Code Example

Code Listing 18 Code example for I2c_MasterTxNotifiaction

```
1  void I2c_MasterTxNotification(uint8 Channel)
2  {
3    /* Confirm if want to stop transaction */
4    if (endTransaction == FALSE)
5    {
6      /* Setup the buffers for the next transaction */
7      I2c_SetupEb(Channel, &TxBuffer[0], transmit_size, &RxBuffer[0],
   receive_size);
8      /* Call I2c_MasterWrite to start the next transaction */
9      I2c_MasterWrite(Channel, target_slave_address);
10   }
11 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- *In cases where you want to continue with MasterTx after MasterTx in STOP condition .*

7.7.1.2 I2c_MasterRxNotification

Syntax

```
void I2c_MasterRxNotification(uint8 Channel)
```

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

This notification is a user-provided callback routine to notify that a job has been finished.

It will be called at the end of a master read job.

If the driver is in the repeated start mode, this callback is only called after sending the STOP bit. This means, that, this callback is called only once in the one continuous transaction. And if the last transaction was a MasterRead operation, then this callback is called.

Code Example

Code Listing 19 Code example for I2c_MasterRxNotification

```
1  void I2c_MasterRxNotification(uint8 Channel)
2  {
3    /* Confirm if want to stop transaction */
4    if (endTransaction == FALSE)
5    {
6      /* Setup the buffers for the next transaction */
7      I2c_SetupEb(Channel, &TxBuffer[0], transmit_size, &RxBuffer[0],
   receive_size);
8      /* Call I2c_MasterRead to start the next transaction */
9      I2c_MasterRead(Channel, target_slave_address);
10   }
11 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- *In cases where you want to continue with MasterRx after MasterRx in STOP condition .*

7.7.1.3 I2c_SlaveTxNotification

Syntax

```
void I2c_SlaveTxNotification(uint8 Channel)
```

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

This notification is a user-provided callback routine to notify that a job has been finished. It will be called at the end of a slave write job. You can call the `I2c_GetBufferStatus` to confirm the actual transaction length in this function.

Code Example

Code Listing 20 Code example for I2c_SlaveTxNotification

```
1  void I2c_SlaveTxNotification(uint8 Channel)
2  {
3    I2c_BufferType *RemainSrcPtr;
```

```
4    I2c_BufferType *RemainDstPtr;
5    I2c_BufferSizeType RemainSrcSize;
6    I2c_BufferSizeType RemainDstSize;
7    I2c_GetBufferStatus(Channel, &RemainSrcPtr, &RemainDstPtr,
   &RemainSrcSize, &RemainDstSize);
8
9    /* Check the remaining sending data size using RemainSrcSize */
10
11   /* Confirm if want to stop transaction */
12   if (endTransaction == FALSE)
13   {
14     /* Setup the buffers for the next transaction */
15     I2c_SetupEb(Channel, &TxBuffer[0], transmit_size, &RxBuffer[0],
   receive_size);
16     I2c_SlaveAwaitRequest(Channel);
17   }
18 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- `I2cHwAutoAckSlaveAddress` is enabled.
- In cases where you want to check the remaining data size and continue with SlaveTx/SlaveRx after SlaveTx in STOP condition.

7.7.1.4 I2c_SlaveRxNotification

Syntax

```
void I2c_SlaveRxNotification(uint8 Channel)
```

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

This notification is a user-provided callback routine to notify that a job has been finished. It will be called at the end of a slave read job. You can call the `I2c_GetBufferStatus` to confirm the actual transaction length in this function.

Code Example

Code Listing 21 Code example for I2c_SlaveRxNotification

```
1  void I2c_SlaveRxNotification(uint8 Channel)
2  {
3    I2c_BufferType *RemainSrcPtr;
4    I2c_BufferType *RemainDstPtr;
```

```
5    I2c_BufferSizeType RemainSrcSize;
6    I2c_BufferSizeType RemainDstSize;
7    I2c_GetBufferStatus(Channel, &RemainSrcPtr, &RemainDstPtr,
   &RemainSrcSize, &RemainDstSize);
8
9    /* Check the remaining receiving data size using RemainDstSize */
10
11   /* Confirm if want to stop transaction */
12   if (endTransaction == FALSE)
13   {
14     /* Setup the buffers for the next transaction */
15     I2c_SetupEb(Channel, &TxBuffer[0], transmit_size, &RxBuffer[0],
   receive_size);
16     I2c_SlaveAwaitRequest(Channel);
17   }
18 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- `I2cHwAutoAckSlaveAddress` is enabled.
- In cases where you want to check the remaining data size and continue with SlaveTx/SlaveRx after SlaveRx in STOP condition.

7.7.1.5 I2c_MasterTxErrorNotification

Syntax

```
void I2c_MasterTxErrorNotification(uint8 Channel)
```

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

This notification is a user-provided callback routine to notify that a job finished with errors. It will be called at the end of a master write job. It is also recommended that you call the `I2c_Cancel` API to completely terminate the transaction in which the error occurred before executing the next transaction.

Code Example

Code Listing 22 Code example for I2c_MasterTxErrorNotification

```
1  void I2c_MasterTxErrorNotification(uint8 Channel)
2  {
3    /* Call cancel to stop transaction */
4    if (E_OK == I2c_Cancel(Channel))
5    {
```

```
6       /* Confirm if want to stop transaction */
7       if (endTransaction == FALSE)
8       {
9         /* Setup the buffers for the next transaction */
10        I2c_SetupEb(Channel, &TxBuffer[0], transmit_size, &RxBuffer[0],
   receive_size);
11        /* Call I2c_MasterWrite to start the next transaction */
12        I2c_MasterWrite(Channel, target_slave_address);
13      }
14    }
15 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- *In cases where you want to continue with MasterTx after MasterTx in ERROR condition .*

7.7.1.6 I2c_MasterRxErrorNotification

Syntax

```
void I2c_MasterRxErrorNotification(uint8 Channel)
```

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

This notification is a user-provided callback routine to notify that a job has been finished with error. It will be called at the end of a master read job. It is also recommended that you call the `I2c_Cancel` API to completely terminate the transaction in which the error occurred before executing the next transaction.

Code Example

Code Listing 23 Code example for I2c_MasterRxErrorNotification

```
1  void I2c_MasterRxErrorNotification(uint8 Channel)
2  {
3    /* Call cancel to stop transaction */
4    if (E_OK == I2c_Cancel(Channel))
5    {
6      /* Confirm if want to stop transaction */
7      if (endTransaction == FALSE)
8      {
9        /* Setup the buffers for the next transaction */
10       I2c_SetupEb(Channel, &TxBuffer[0], transmit_size, &RxBuffer[0],
   receive_size);
11       /* Call I2c_MasterRead to start the next transaction */
```

```
12        I2c_MasterRead(Channel, target_slave_address);
13      }
14    }
15 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- *In cases where you want to continue with MasterRx after MasterRx in ERROR condition .*

7.7.1.7 I2c_SlaveTxErrorNotification

Syntax

```
void I2c_SlaveTxErrorNotification(uint8 Channel)
```

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

This notification is a user-provided callback routine to notify that a job has been finished with error. It will be called at the end of a slave write job. It is also recommended that you call the `I2c_Cancel` API to completely terminate the transaction in which the error occurred before executing the next transaction.

Code Example

Code Listing 24 Code example for I2c_SlaveTxErrorNotification

```
1  void I2c_SlaveTxErrorNotification(uint8 Channel)
2  {
3    /* Call cancel to stop transaction */
4    if (E_OK == I2c_Cancel(Channel))
5    {
6      /* Confirm if want to stop transaction */
7      if (endTransaction == FALSE)
8      {
9        /* Setup the buffers for the next transaction */
10       I2c_SetupEb(Channel, &TxBuffer[0], transmit_size, &RxBuffer[0],
   receive_size);
11       I2c_SlaveAwaitRequest(Channel);
12     }
13   }
14 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

– *In cases where you want to continue with SlaveTx/SlaveRx after SlaveTx in ERROR condition .*

7.7.1.8 I2c_SlaveRxErrorNotification

Syntax

```
void I2c_SlaveRxErrorNotification(uint8 Channel)
```

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

This notification is a user-provided callback routine to notify that a job has been finished with error. It will be called at the end of a slave read job. It is also recommended that you call the `I2c_Cancel` API to completely terminate the transaction in which the error occurred before executing the next transaction.

Code Example

Code Listing 25 Code example for I2c_SlaveRxErrorNotification

```
1  void I2c_SlaveRxErrorNotification(uint8 Channel)
2  {
3    /* Call cancel to stop transaction */
4    if (E_OK == I2c_Cancel(Channel))
5    {
6      /* Confirm if want to stop transaction */
7      if (endTransaction == FALSE)
8      {
9        /* Setup the buffers for the next transaction */
10       I2c_SetupEb(Channel, &TxBuffer[0], transmit_size, &RxBuffer[0],
   receive_size);
11       I2c_SlaveAwaitRequest(Channel);
12     }
13   }
14 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

– *In cases where you want to continue with SlaveTx/SlaveRx after SlaveRx in ERROR condition .*

7.7.1.9 I2c_MasterComReqNotification

Syntax

`void I2c_MasterComReqNotification(uint8 Channel)`

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

This notification is a user-provided callback routine to notify that a master job finished in repeated start mode. In this callback, you can call the next communication request (`I2c_MasterWrite` or `I2c_MasterRead` or `I2c_Cancel`), and also call `I2c_SetupEb` to change the buffer or length. Infineon recommends calling the next communication request in this callback, but if you do not call the next communication request in this callback, then the I2C driver starts to wait for the next communication request. In this case, you can call the corresponding APIs after this callback. However, note that until the next communication request, the bus is occupied by the previous transaction.

If the previous transaction is MasterWrite, then confirm that the Tx transaction has ended by calling `I2c_ConfirmTxTransaction`, before you call the next communication request.

You should not call both the stop request (`I2c_Cancel`) and the start transaction request (`I2c_MasterWrite`/`I2c_MasterRead`) at the same time in this callback.

Code Example

Code Listing 26 Code example for I2c_MasterComReqNotification(MasterTx->MasterTx)

```
1  void I2c_MasterComReqNotification(uint8 Channel)
2  {
3    /* If the previous transaction is MasterWrite, then confirm if the
   Tx transaction has finished */
4    if (I2c_ConfirmTxTransaction(Channel) == E_OK)
5    {
6      /* Confirm if want to stop transaction */
7      if (endTransaction == TRUE)
8      {
9        /* Call cancel to stop transaction */
10       I2c_Cancel(Channel);
11     }
12     else
13     {
14       /* Setup the buffers for the next transaction */
15       I2c_SetupEb(Channel, &TxBuffer[0], transmit_size, &RxBuffer[0],
   receive_size);
16       /* Call I2c_MasterWrite to re-start the next transaction */
17       I2c_MasterWrite(Channel, target_slave_address);
18     }
```

```
    }
}
```

Note: It is necessary to use `I2c_ConfirmTxTransaction()` to confirm whether the Tx transaction has ended after the MasterTx transaction in repeated start mode.

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- *If you want to continue with MasterTx after MasterTx in repeated start condition, please refer to the path where* `endTransaction = FALSE.`
- *If you want stop transaction in repeated start condition, please refer to the path where* `endTransaction = TRUE.`

Code Listing 27 Code example for I2c_MasterComReqNotification(MasterRx->MasterRx)

```
void I2c_MasterComReqNotification(uint8 Channel)
{
  if (endTransaction == TRUE)
  {
    /* Call cancel to stop transaction */
    I2c_Cancel(Channel);
  }
  else
  {
    /* Setup the buffers for the next transaction */
    I2c_SetupEb(Channel, &TxBuffer[0], transmit_size, &RxBuffer[0],
receive_size);
    /* Call I2c_MasterRead to re-start the next transaction */
    I2c_MasterRead(Channel, target_slave_address);
  }
}
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- *If you want to continue with MasterRx after MasterRx in repeated start condition, please refer to the path where* `endTransaction = FALSE.`
- *If you want stop transaction in repeated start condition, please refer to the path where* `endTransaction = TRUE.`

7.7.1.10 I2c_SlaveSrNotification

Syntax

```
void I2c_SlaveSrNotification (uint8 Channel)
```

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

This notification is a user-provided callback routine to notify that a slave job has been finished in repeated start mode. In this callback, call `I2c_SetupEb` or `I2c_UpdateTxBuffer` or `I2c_SlaveAwaitRequest` to prepare the next transaction.

Code Example

Code Listing 28 Code example for I2c_SlaveSrNotification(using I2c_SetupEb to prepare the next transaction)

```
1  void I2c_SlaveSrNotification(uint8 Channel)
2  {
3    /* Setup the buffers for the next transaction */
4    I2c_SetupEb(Channel, &TxBuffer[0], transmit_size, &RxBuffer[0],
   receive_size);
5    I2c_SlaveAwaitRequest(Channel);
6  }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- `I2cHwAutoAckSlaveAddress` is enabled.
- *In cases where you want to use* `I2c_StepupEb()` *to setup the buffers and continue with SlaveTx/SlaveRx after SlaveTx/SlaveRx in repeated start condition .*

Code Listing 29 Code example for I2c_SlaveSrNotification(using I2c_UpdateTxBuffer to prepare the next transaction)

```
1  void I2c_SlaveSrNotification(uint8 Channel)
2  {
3    /* Updated Tx buffer for the next transaction */
4    I2c_UpdateTxBuffer(Channel, &TxBuffer[0], transmit_size);
5    I2c_SlaveAwaitRequest(Channel);
6  }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- `I2cHwAutoAckSlaveAddress` is enabled.
- *In cases where you want to use* `I2c_UpdateTxBuffer()` *to setup the buffers and continue with SlaveTx after SlaveTx/SlaveRx in repeated start condition .*

7.7.1.11 I2c_SlaveCompleteNotification

Syntax

```
void I2c_SlaveCompleteNotification(uint8 Channel, I2c_SlaveCompleteEventType
Event, uint32 TransferCount)
```

Parameters (in)

- `Channel` – Channel ID
- `Event` - Slave transfer complete event. See I2c_SlaveCompleteEventType.
- `TransferCount` - Indicator of the Transferred count

Parameters (out)

None

Return value

None

Description

This notification is a user-provided callback routine to notify that the slave transaction has been finished. The transaction complete event kind (param:`Event`) and data length informations (param:`TransferCount`) are provided through this notification function. In this callback, call `I2c_SetupEb` or `I2c_UpdateTxBuffer` or `I2c_SlaveAwaitRequest` to prepare the next transaction.

Code Example

Code Listing 30 Code example for I2c_SlaveCompleteNotification(Slave Tx greater than expected)

```
1  /* TransferCount > transmit_size */
2  void I2c_SlaveCompleteNotification(uint8 Channel,
   I2c_SlaveCompleteEventType Event, uint32 TransferCount)
3  {
4    /* Confirm if want to stop transaction */
5    if (endTransaction == FALSE)
6    {
7      /* The sending size expected by the slave is smaller than the
   receiving size expected by the master. */
8      if ((Event == I2C_SLAVE_COMPLETE_STOP_TX_GT) || (Event ==
   I2C_SLAVE_COMPLETE_RESTART_TX_GT))
9      {
10       /* The slave adjusts its buffer size to send the data the
   master expects */
11       I2c_SetupEb(Channel, &TxBuffer[0], TransferCount -
   transmit_size, &RxBuffer[0], receive_size);
12       I2c_SlaveAwaitRequest(Channel);
13     }
14   }
15 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- `I2cHwAutoAckSlaveAddress` is disabled.
- *Slave Tx size is greater than expected* .
- In cases where you want to adjust the buffer size and continue with SlaveTx after SlaveTx.

Code Listing 31 Code example for I2c_SlaveCompleteNotification(Slave Tx equal to expected)

```
1  /* TransferCount = transmit_size */
2  void I2c_SlaveCompleteNotification(uint8 Channel,
   I2c_SlaveCompleteEventType Event, uint32 TransferCount)
3  {
4    /* Confirm if want to stop transaction */
5    if (endTransaction == FALSE)
6    {
7      /* The send size of the slave is the same as the receive size of
   the master */
8      if ((Event == I2C_SLAVE_COMPLETE_STOP_TX_EQ) || (Event ==
   I2C_SLAVE_COMPLETE_RESTART_TX_EQ))
9      {
10         /* Setup the buffers for the next transaction */
11         I2c_SetupEb(Channel, &TxBuffer[0], transmit_size,
   &RxBuffer[0], receive_size);
12         I2c_SlaveAwaitRequest(Channel);
13     }
14   }
15 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- `I2cHwAutoAckSlaveAddress` *is disabled.*
- *Slave Tx size is equal to expected.*
- *In cases where you want to continue with SlaveTx/SlaveRx after SlaveTx.*

Code Listing 32 Code example for I2c_SlaveCompleteNotification(Slave Tx less than expected)

```
1  /* TransferCount < transmit_size */
2  void I2c_SlaveCompleteNotification(uint8 Channel,
   I2c_SlaveCompleteEventType Event, uint32 TransferCount)
3  {
4    /* Confirm if want to stop transaction */
5    if (endTransaction == FALSE)
6    {
7      /* The sending size expected by the slave is greater than the
   receiving size expected by the master. */
8      if ((Event == I2C_SLAVE_COMPLETE_STOP_TX_LT) || (Event ==
   I2C_SLAVE_COMPLETE_RESTART_TX_LT))
9      {
10         /* Setup the buffers for the next transaction */
11         I2c_SetupEb(Channel, &TxBuffer[0], transmit_size,
   &RxBuffer[0], receive_size);
12         I2c_SlaveAwaitRequest(Channel);
13     }
14   }
15 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- `I2cHwAutoAckSlaveAddress` *is disabled.*
- *Slave Tx size is less than expected.*
- *In cases where you want to continue with SlaveTx/SlaveRx after SlaveTx.*

Code Listing 33 Code example for I2c_SlaveCompleteNotification(Slave Rx greater or equal than expected)

```
1  /* TransferCount >= receive_size */
2  void I2c_SlaveCompleteNotification(uint8 Channel,
   I2c_SlaveCompleteEventType Event, uint32 TransferCount)
3  {
4    /* Confirm if want to stop transaction */
5    if (endTransaction == FALSE)
6    {
7      /* The sending size expected by the slave is smaller than the
   receiving size expected by the master */
8      /* Or, the receive size of the slave is the same as the send size
   of the master */
9      if ((Event == I2C_SLAVE_COMPLETE_STOP_RX_GE) || (Event ==
   I2C_SLAVE_COMPLETE_RESTART_RX_GE))
10     {
11         if (TransferCount == receive_size)
12         {
13            /* Setup the buffers for the next transaction */
14            I2c_SetupEb(Channel, &TxBuffer[0], transmit_size,
   &RxBuffer[0], receive_size);
15         }
16         else
17         {
18            /* The slave adjusts its buffer size to receive the data
   the master expects */
19            I2c_SetupEb(Channel, &TxBuffer[0], transmit_size,
   &RxBuffer[0], TransferCount-receive_size);
20         }
21         I2c_SlaveAwaitRequest(Channel);
22     }
23   }
24 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- `I2cHwAutoAckSlaveAddress` *is disabled.*
- *Slave Rx is greater or equal than expected.*
- *If "Transfer Count > receive_size", you want to adjust the buffer size and continue with SlaveRx after SlaveRx.*
- *If "TransferCount = receive_size", you want to continue with SlaveTx/SlaveRx after SlaveRx.*

Code Listing 34 Code example for I2c_SlaveCompleteNotification(Slave Rx less than expected)

```
1  /* TransferCount < receive_size */
2  void I2c_SlaveCompleteNotification(uint8 Channel,
   I2c_SlaveCompleteEventType Event, uint32 TransferCount)
3  {
```

```
4    /* Confirm if want to stop transaction */
5    if (endTransaction == FALSE)
6    {
7      /* The receiving size expected by the slave is greater than the
   sending size expected by the master. */
8      if ((Event == I2C_SLAVE_COMPLETE_STOP_RX_LT) || (Event ==
   I2C_SLAVE_COMPLETE_RESTART_RX_LT))
9      {
10         I2c_SetupEb(Channel, &TxBuffer[0], transmit_size,
   &RxBuffer[0], receive_size);
11         I2c_SlaveAwaitRequest(Channel);
12     }
13   }
14 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- `I2cHwAutoAckSlaveAddress` *is disabled.*
- *Slave Rx is less than expected.*
- *In cases where you want to continue with SlaveTx/SlaveRx after SlaveRx.*

7.7.1.12 I2c_SlaveAddressMatchNotification

Syntax

```
I2c_AcknowledgeType I2c_SlaveAddressMatchNotification(uint8 Channel, uint8
SlaveAddress, I2c_TransferDirectionType Direction)
```

Parameters (in)

- `Channel` – Channel ID
- `SlaveAddress` – Slave matching address
- `Direction` – Indicator of the Transfer direction (Read/Write). See I2c_TransferDirectionType.

Parameters (out)

None

Return value

- `I2C_NACK`: I2C transfer non-acknowledge
- `I2C_HOLDACK`: I2C hold acknowledge
- `I2C_ACK`: I2C transfer acknowledge

Description

This notification is a user-provided callback routine to notify that a slave-matching address has been received. The corresponding slave address (param:`SlaveAddress`) and transfer direction (param:`Direction`) are provided through this notification function. In this callback, you can call `I2c_UpdateTxBuffer` to prepare the next transaction, and you must not call `I2c_SetupEb` or `I2c_SlaveAwaitRequest` in this notification.

Note: The application must return the response as `I2C_ACK/I2C_HOLDACK/I2C_NACK` to this notification function, when returning `I2C_ACK/I2C_NACK`, the I2C driver will send the corresponding

acknowledge to the bus. However, when returning `I2C_HOLDACK`, the I2C will enter clock stretching status until `I2c_SlaveStartTransfer` is called, during this notification function or after you return `I2C_HOLDACK` (means that before sent the acknowledge), clock stretch will be applied to the bus.

Code Example

Code Listing 35 Code example for I2c_SlaveAddressMatchNotification

```
1  I2c_AcknowledgeType I2c_SlaveAddressMatchNotification(uint8 Channel,
   I2c_SlaveAddressType SlaveAddress, I2c_TransferDirectionType
   Direction)
2  {
3    I2c_AcknowledgeType ret;
4
5    /* Confirm if slave address matching */
6    if (SlaveAddress != target_slave_address)
7    {
8      ret = I2C_NACK;
9    }
10   else
11   {
12     /* Confirm if next transaction is TX */
13     if (Direction == I2C_READ)
14     {
15       /* Setup the buffers for the next transaction */
16       I2c_UpdateTxBuffer(Channel, &TxBuffer[0], transmit_size);
17     }
18     ret = I2C_ACK;
19   }
20   return ret;
21 }
```

Note: depending on the configuration you should select the implementation of notification. This example is for following conditions:

- `I2cHwAutoAckSlaveAddress` *is disabled.*
- *If you want to perform slave address matching, please refer to the path where* `SlaveAddress != target_slave_address`.

  In cases where you want to continue with SlaveTx after SlaveAddrMatch, please refer to the path where `Direction = I2C_READ`.

7.7.2 DET

If default error detection is enabled, the I2C driver uses the following callback function provided by DET. If you do not use DET, you must implement this function within your application.

7.7.2.1 Det_ReportError

Syntax

```
Std_ReturnType Det_ReportError
(
   uint16 ModuleId,
   uint8 InstanceId,
   uint8 ApiId,
   uint8 ErrorId
)
```

Reentrancy

Reentrant

Parameters (in)

- `ModuleId` – Module ID of the calling module
- `InstanceId` – Instance ID of the calling module
- `ApiId` – ID of the API service that calls this function
- `ErrorId` – ID of the detected development error

Return value

Always returns `E_OK`.

Description

Service for reporting development errors.

7.7.3 DEM

If DEM notifications are enabled, the I2C driver uses the following callback function provided by DEM. If you do not use DEM, you must implement this function within your application.

7.7.3.1 Dem_ReportErrorStatus

Syntax

```
void Dem_ReportErrorStatus
(
   Dem_EventIdType EventId,
   Dem_EventStatusType EventStatus
)
```

Reentrancy

Reentrant

Parameters (in)

- `EventId` – Identification of an event by the assigned event ID
- `EventStatus` – Monitor the test result of the given event

Return value

None

Description

Service for reporting diagnostic events.

7.7.4 Error callout functions

7.7.4.1 Error callout API

The I2C driver requires an error callout handler. Every error is reported to this handler; error checking cannot be switched off. The name of the function to be called can be configured by the `I2cErrorCalloutFunction` parameter.

Syntax

```
void Error_Handler_Name
(
   uint16 ModuleId,
   uint8 InstanceId,
   uint8 ApiId,
   uint8 ErrorId
)
```

Reentrancy

Reentrant

Parameters (in)

- `ModuleId` – Module ID of the calling module
- `InstanceId` – Instance ID of the calling module
- `ApiId` – ID of the API service that calls this function
- `ErrorId` – ID of the detected error

Return value

None

Description

Service for reporting errors

8 Appendix B - Access register table

8.1 SCB

Table 23 SCB access register table

Register	Bit no.	Access size	Value	Description	Timing	Mask value	Monitoring value
CTRL	31:0	Word (32 bits)	0x00000000	Initialize the CTRL register	After calling `I2c_Init or I2c_InitHwUnit`	0x8300400F	0x00000000
			0x0300400F	Deinitialize the CTRL register	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0x8300400F	0x0300400F
			0x80000000	Set up the CTRL register	During master write / master read / slave mode operation	0x8300400F	0x80000000
I2C_CTRL	31:0	Word (32 bits)	0x00000000	Initialize the I2C_CTRL register	During calling `I2c_Init or I2c_InitHwUnit`	0xC000FBFF	0x00000000
			0x0000FB88	Deinitialize the I2C_CTRL register	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0xC000FBFF	0x0000FB88
			0x80000000 | OvsValue Depends on API and configuration	Set up the I2C_CTRL register.	During master write operation	0xC000FBFF	0x800000** * = Depends on API and configuration
			0x80000200 | 1 or 0 << 8 | OvsValue Depends on API and configuration	Set up the I2C_CTRL register.	During master read operation	0xC000FBFF	0x80000*** * = Depends on API and configuration
			0x4000C800 | 1 or 0 << 12 | 1 or 0 << 13	Set up the I2C_CTRL register	During slave mode operation	0xC000FBFF	0x4000*800 * = Depends on API and configuration

Register	Bit no.	Access size	Value	Description	Timing	Mask value	Monitoring value
			Depends on API and configuration				
TX_CTRL	31:0	Word (32 bits)	0x00010107	Initialize the TX_CTRL register	After calling `I2c_Init or I2c_InitHwUnit`	0x00010107	0x00010107
			0x00000107	Deinitialize the TX_CTRL register	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0x00010107	0x00000107
TX_FIFO_CTRL	31:0	Word (32 bits)	0x00000000	Deinitialize the TX_FIFO_CTRL register	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0x0001007F	0x00000000
			0x00000000 | Invalidate FIFO << 16 | Trigger level Depends on API and configuration	Set up the TX_FIFO_CTRL register	During master write operation	0x0001007F	0x000*00** bit[16]: clear FIFO (end of job:1, other:0) bit[6:0]: trigger level (configured trigger level or DMA usage:1)
			0x00000000 | Invalidate FIFO << 16 | Trigger level Depends on API and configuration	Set up the TX_FIFO_CTRL register	During slave mode operation	0x0001007F	0x000*00** bit[16]: clear FIFO (end of job:1, other:0) bit[6:0]: trigger level (configured trigger level or DMA usage: 1)
TX_FIFO_STATUS	31:0	Word (32 bits)	-	Read-only register	Always	0x00000000 (Monitoring not needed.)	0x00000000 (Monitoring not needed.)

Register	Bit no.	Access size	Value	Description	Timing	Mask value	Monitoring value
TX_FIFO_WR	31:0	Word (32 bits)	Transfer data	Transfer data	During transfer	-	Write-only register
RX_CTRL	31:0	Word (32 bits)	0x00000107	Initialize the RX_CTRL register	After calling `I2c_Init or I2c_InitHwUnit`	0x00000307	0x00000107
			0x00000107	Deinitialize the RX_CTRL register	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0x00000307	0x00000107
			0x00000107 Depends on API and configuration	Set up the RX_CTRL register	During master read / slave mode operation	0x00000307	0x00000*07. bit[9]: Depends on the glitch filter configuration
RX_FIFO_CTRL	31:0	Word (32 bits)	0x00000000	Deinitialize the RX_FIFO_CTRL register	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0x0001007F	0x00000000
			0x00000000 | Freeze FIFO << 17 | Invalidate FIFO << 16 | Trigger level Depends on API and configuration	Set up the RX_FIFO_CTRL register	During master read / slave mode operation	0x0003007F	0x0000*0** bit[17]: freeze FIFO(full of external RX buffer when sending acknowledge by HW:1, other:0) bit[16]: clear FIFO (end of job:1, other:0) bit[6:0]: trigger level (configured trigger level or DMA usage:1 or sending acknowledge by SW:1)

Register	Bit no.	Access size	Value	Description	Timing	Mask value	Monitoring value
RX_FIFO_STATUS	31:0	Word (32 bits)	-	Read-only register	Always (Depends on FIFO situation.)	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
RX_MATCH	31:0	Word (32 bits)	0x00000000	Initialize	After calling `I2c_Init or I2c_InitHwUnit`	0x00FF00FF	0x00000000
			0x00000000	Initialize	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0x00FF00FF	0x00000000
			Slave address | Slave address mask << 16 Depends on API and configuration	Set up the RX_MATCH register	During slave mode operation	0x00FF00FF	0x00**00** bit[23:16]:slave address mask bit[7:0]:slave address
RX_FIFO_RD	31:0	Word (32 bits)	-	Read-only register	When reading the received data	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
INTR_CAUSE	31:0	Word (32 bits)	0x00000000	Initialize	After calling `I2c_Init or I2c_InitHwUnit`	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
			0x00000000	Deinitialize	After calling `I2c_DeInit or I2c_DeinitHwUnit`		
			0x00000000 | RX interrupt << 3 | TX interrupt << 2 | Slave interrupt << 1 | Master interrupt Read only	Interrupt cause (Read-only)	During transfer		
INTR_M	31:0		0x00000000	Initialize	After calling `I2c_Init or I2c_InitHwUnit`	0x00000000	0x00000000

Register	Bit no.	Access size	Value	Description	Timing	Mask value	Monitoring value
		Word (32 bits)	0x00000000	Deinitialize	After calling `I2c_DeInit` or `I2c_DeinitHwUnit`	(Monitoring is not needed.)	(Monitoring is not needed.)
			0x00000000 | I2C_BusError << 8 | Stop << 4 | Ack << 2 | Nack << 1 | Arb_lost	Master mode interrupt cause (read and clear with write)	During transfer		
INTR_M_MASK	31:0	Word (32 bits)	0x00000000	Initialize	After calling `I2c_Init` or `I2c_InitHwUnit`	0x00000113	0x00000000
			0x00000000	Deinitialize	After calling `I2c_DeInit` or `I2c_DeinitHwUnit`	0x00000113	0x00000000
			0x00000000 | I2C_BusError << 8 | Stop << 4 | Nack << 1 | Arb_lost	Enable or disable the master mode interrupt.	During transfer with master operation (interrupt)	0x00000113	0x00000113
INTR_M_MASKED	31:0	Word (32 bits)	-	Read-only register	During transfer with master operation (interrupt)	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring is not needed.)
INTR_S	31:0	Word (32 bits)	0x00000000	Initialize	After calling `I2c_Init` or `I2c_InitHwUnit`	0x00000000 (Monitoring not needed.)	0x00000000 (Monitoring not needed.)
			0x00000000	Deinitialize	After calling `I2c_DeInit` or `I2c_DeinitHwUnit`		
			0x00000000 | I2C_BusError << 8 | Stop << 4 | Nack << 1 | Arb_lost | Addr_Match << 6	Slave mode interrupt cause (read and clear with write)	During transfer		
INTR_S_MASK	31:0		0x00000000	Initialize	After calling `I2c_Init` or `I2c_InitHwUnit`	0x000007FF	0x00000000

Register	Bit no.	Access size	Value	Description	Timing	Mask value	Monitoring value
		Word (32 bits)	0x00000000	Deinitialize	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0x000007FF	0x00000000
			0x00000000 | I2C_BusError << 8 | Stop << 4 | Nack << 1 | Arb_lost | Addr_Match << 6	Enable or disable the slave mode interrupt	During transfer with slave operation (interrupt)	0x000007FF	0x000001*3 * = Depends on configuration
INTR_S_MASKED	31:0	Word (32 bits)	-	Read-only register	During transfer with slave operation (interrupt)	0x00000000 (Monitoring not needed.)	0x00000000 (Monitoring not needed.)
INTR_TX	31:0	Word (32 bits)	0x00000000	Initialize	After calling `I2c_Init or I2c_InitHwUnit`	0x00000000 (Monitoring not needed.)	0x00000000 (Monitoring not needed.)
			0x00000000	Deinitialize	After calling `I2c_DeInit or I2c_DeinitHwUnit`		
			0x00000000 | Overflow << 5 | Empty << 4 | Trigger	TX interrupt cause (read and clear with write)	During Tx transaction (interrupt)		
INTR_TX_MASK	31:0	Word (32 bits)	0x00000000	Initialize	After calling `I2c_Init or I2c_InitHwUnit`	0x0000007F	0x00000000
			0x00000000	Deinitialize	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0x0000007F	0x00000000
			0x00000000 | Overflow << 5 | Empty << 4 | Trigger	Enable or disable the TX interrupt cause	During Tx transaction	0x0000007F	0x000000** Depends on the transfer status.
INTR_TX_MASKED	31:0	Word (32 bits)	-	Read-only register	During Tx transaction	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring not needed.)

Register	Bit no.	Access size	Value	Description	Timing	Mask value	Monitoring value
INTR_RX	31:0	Word (32 bits)	0x00000000	Initialize	After calling `I2c_Init or I2c_InitHwUnit`	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring not needed.)
			0x00000000	Deinitialize	After calling `I2c_DeInit or I2c_DeinitHwUnit`		
			0x00000000 | Underflow << 6 | Overflow << 5 | Full << 3 | Not empty << 2 | Trigger	Rx interrupt cause (read and clear with write)	During RX transaction (interrupt)		
INTR_RX_MASK	31:0	Word (32 bits)	0x00000000	Initialize	After calling `I2c_Init or I2c_InitHwUnit`	0x0000007F	0x00000000
			0x00000000	Deinitialize	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0x0000007F	0x00000000
			0x00000000 | Underflow << 6 | Overflow << 5 | Full << 3 | Not empty << 2 | Trigger	Rx interrupt cause (read and clear with write)	During Rx transaction	0x0000007F	0x000000** Depends on the transfer status.
INTR_RX_MASKED	31:0	Word (32 bits)	-	Read-only register	During Rx transaction	0x00000000 (Monitoring not needed.)	0x00000000 (Monitoring not needed.)

8.2 DMA (DW)

Table 24 DMA (DW) access register table

Register	Bit no.	Access size	Value	Description	Timing	Mask value	Monitoring value
CH_CTL	31:0	Word (32 bits)	0x00000002	Initialize the channel control register	After calling `I2c_Init or I2c_InitHwUnit`	0x80000BF4	0x00000000
			0x00000002	Deinitialize the channel control register	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0x80000BF4	0x00000000
			0x00000000 | DMA channel enable << 31	Start or stop DMA	During transfer with DMA	0x80000BF4	0x00000000 bit[31]:Set on transfer (DMA) start/Cleared on transfer (DMA) end
CH_STATUS	31:0	Word (32 bits)	-	Read-only register	Always	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring not needed.)
CH_IDX	31:0	Word (32 bits)	0x00000000	Initialize the channel's current indices	After calling `I2c_Init or I2c_InitHwUnit`	0x0000FFFF	0x00000000
			0x00000000	Deinitialize the channel's current indices	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0x0000FFFF	0x00000000
			0x00000000 | Y loop index << 8 | X loop index	Calculate the buffer position	During transfer with DMA	0x0000FFFF	0x00000000 bit[15:8] | bit[7:0] Changed during transfer

Register	Bit no.	Access size	Value	Description	Timing	Mask value	Monitoring value
CH_CURR_PTR	31:0	Word (32 bits)	0x00000000	Initialize the channel's current indices	After calling `I2c_Init or I2c_InitHwUnit`	0x0000FFFF	0x00000000
			0x00000000	Deinitialize the channel's current indices	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0x0000FFFF	0x00000000
			0x00000000 | Address of descriptor	Descriptor position	During transfer with DMA	0xFFFFFFFF	0x00000000 bit[31:2]:Set to current descriptor address on start of transfer
INTR	31:0	Word (32 bits)	0x00000000	Initialize the channel's current indices	After calling `I2c_Init or I2c_InitHwUnit`	0x00000000 (Monitoring is not needed.)	0x00000000 (Monitoring not needed.)
			0x00000000	Deinitialize the channel's current indices	After calling `I2c_DeInit or I2c_DeinitHwUnit`		
			0x00000001	Descriptor position	During transfer with DMA		
INTR_MASK	31:0	Word (32 bits)	0x00000000	Initialize the channel's current indices	After calling `I2c_Init or I2c_InitHwUnit`	0x00000001	0x00000000
			0x00000000	Deinitialize the channel's current indices	After calling `I2c_DeInit or I2c_DeinitHwUnit`	0x00000001	0x00000000
			0x00000000 | Enable interrupt	Descriptor position	During transfer with DMA	0x00000000	0x00000000 bit[0]:Set on DMA start/Cleared on DMA end
INTR_MASKED	31:0	Word (32 bits)	-	Read-only register	During transfer with DMA	0x00000000 (Monitoring not needed.)	0x00000000 (Monitoring not needed.)

Register	Bit no.	Access size	Value	Description	Timing	Mask value	Monitoring value
SRAM_DATA0	31:0	Word (32bit)	0x00000000	Initialize the current data	After calling `I2c_Init` or `I2c_InitHwUnit`	0xFFFFFFFF	0x00000000
SRAM_DATA1	31:0	Word (32bit)	0x00000000	Initialize the current data	After calling `I2c_Init` or `I2c_InitHwUnit`	0xFFFFFFFF	0x00000000

Revision history

Version	Date	Description
**	2020-09-24	Initial release.
*A	2021-01-18	4.2 I2C Configuration Changed “Note” in I2cChannelId 4.2.3 I2C Channel OVS config Deleted “Note” in I2cOvsId 5.1.2.3 Using DMA, 5.1.3.3 Using DMA, 5.1.4.1.3 Using DMA and 6.4 DMA Added conditions for DMA transfer to operate 5.1.6 Cancel the Operation Added “Note” about cancel in slave write mode operation. 5.6.2 Unrecoverable Failure Added “Note” in slave write operation 7.7.1 I2C Notification Functions Added note about notifications Migrated to Infineon template.
*B	2021-03-31	2.6.2 Memory allocation and constraints Add the restriction for VRAM
*C	2021-06-15	Added a note in 5.8 Sleep mode
*D	2021-08-18	Added a note in 6.3 Interrupts
*E	2021-12-21	Updated to the latest branding guidelines.
*F	2022-07-05	Deleted "Unintended interrupt cause" -Table 5 General configuration -5.6.1 Recoverable failure Deleted "I2C_E_HW_INVALID_INTERRUPT_ERROR" and the description or value related to this error code. -5.6.1 Recoverable failure -Table 15 Error codes -7.5.1 I2c_MainFunction_Handling -7.6.1 I2c_Interrupt_SCB<n>_CatX -7.6.2 I2c_Interrupt_DMA_CH<m>_Isr_CatY
*G	2023-03-08	Added description for repeated start mode in the following sections 5.1.2.1 Using interrupt 5.1.2.2 Using polling 5.1.2.3 Using DMA 5.1.3.1 Using interrupt 5.1.3.2 Using polling 5.1.3.3 Using DMA 5.1.4.1.1 Using interrupt 5.1.4.1.2 Using polling 5.1.4.1.3 Using DMA 5.1.4.2.1 Using interrupt

Version	Date	Description
		5.1.4.2.2 Using polling 5.1.4.2.3 Using DMA 5.1.6 Cancel the operation 7.4.6 I2c_MasterWrite (adding note) 7.4.7 I2c_MasterRead (adding note) Added Set repeated start mode API in the following sections 5.1.7.3 Repeated Start mode 7.3.4 API service IDs 7.4.14 I2c_SetRepeatedStart Added/Changed description due to repeated start in the following sections 7.7.1 I2C notification functions 7.7.1.1 I2c_MasterTxNotification 7.7.1.2 I2c_MasterRxNotification 7.7.1.3 I2c_SlaveTxNotification 7.7.1.4 I2c_SlaveRxNotification Added callback function for repeated start mode in the following sections 2.3 Adapting your application 7.7.1.9 I2c_MasterComReqNotification 7.7.1.10 I2c_SlaveSrNotification Added description for data length is shorter or longer case in the following section 5.1.4 Slave mode operation Added description for checking the actual transaction length in the following section 5.1.5.3 Buffer status Removed the data length error and tx underflow error in the following sections 5.6.1 Recoverable failure 5.6.2 Unrecoverable failure 7.3.1 Error codes 7.6.1 I2c_Interrupt_SCB<n>_CatX 7.6.2 I2c_Interrupt_DMA_CH<m>_Isr_CatY Added Get repeated start API in the following sections 7.3.4 API service IDs

Version	Date	Description
		7.4.15 I2c_GetRepeatedStart
*H	2023-04-21	Modified and Added the description regarding to repeated start callback and I2c_ConfirmTxTransaction. 5.1.2.1 Using interrupt 5.1.2.2 Using polling 5.1.2.3 Using DMA 5.1.3.1 Using interrupt 5.1.3.2 Using polling 5.1.3.3 Using DMA 5.1.5.4 Confirm Tx Transaction Table 18 API service IDs 7.4.16 I2c_ConfirmTxTransaction 7.7.1.9 I2c_MasterComReqNotification Removed the Tx underflow error in the following sections 4.1 General configuration
*I	2023-10-06	Table 7 I2C trigger level setting - Fixed typo 5.1.4 Slave mode operation: Updated the note description regarding unexpected length of data requested from master node. 8.2 DMA(DW): Updated the description regarding register access.
*J	2023-12-08	No content updates. Web release.
*K	2024-04-12	Added description about CONST generated during CODE section to the following section 2.6.2 Memory allocation and constraints Fixed typo 5.1.2.1 Using interrupt 5.1.2.2 Using polling 5.1.2.3 Using DMA 5.1.3.1 Using interrupt 5.1.3.2 Using polling 5.1.3.3 Using DMA 5.1.5.2 Latest job result 5.1.6 Cancel the operation 5.1.7.1 OVS settings 7.4.11 I2c_ChangeOvs Table 19 SCB access register table

Version	Date	Description
*K (Contd.)	2024-04-12	Improved the descriptions in the following sections 5.1.4 Slave mode operation 5.1.4.1.1 Using interrupt 5.1.4.1.2 Using polling 5.1.4.1.3 Using DMA 5.1.4.2.1 Using interrupt 5.1.4.2.2 Using polling 5.1.4.2.3 Using DMA 5.1.5.3 Buffer status 6.3 Interrupts 7.4.10 I2c_GetBufferStatus 7.7.1.9 I2c_MasterComReqNotification Modified and added the description regarding to I2c_TxUpdateBuffer 5.1.4.1.4 Update Buffer Table 12 Execution-time dependencies Table 18 API service IDs 7.4.17 I2c_UpdateTxBuffer Added notes descriptions on NACK receive in the following sections 5.6.1 Recoverable failure Added descriptions about recommendations when detecting errors 7.7.1.5 I2c_MasterTxErrorNotification 7.7.1.6 I2c_MasterRxErrorNotification 7.7.1.7 I2c_SlaveTxErrorNotification 7.7.1.8 I2c_SlaveRxErrorNotification
*L	2024-11-29	Added description for two new configuration parameters in the following sections 2.2.1 Configuration outline 4.2 I2C configuration 5.1.5.3 Buffer status 7.4.17 I2c_UpdateTxBuffer Added description for two new notifications in the following sections 2.3 Adapting your application 7.2.8 I2c_TransferDirectionType 7.2.9 I2c_AcknowledgeType 7.2.10 I2c_Slave CompleteEventType 7.7.1 I2C notification functions

Version	Date	Description
*L (Contd.)	2024-11-29	Modified and added the description regarding to slave mode operation 5.1.4 Slave mode operation 7.3.4 API service IDs 7.4.18 I2c_SlaveStartTransfer 7.7.1.11 I2c_SlaveCompleteNotification 7.7.1.12 I2c_SlaveAddressMatchNotification 8.1 SCB Modified the description related to removing the channel state check in I2c_DeInit. 5.1.8 Disabling the I2C driver 5.5.1 Vendor-specific development errors 7.4.2 I2c_DeInit
*M	2025-04-21	Modified and added the description regarding to de-initialization and re-initialization of specified SCB channel 5.1.9 De-initialize and Re-initialize at HW unit 5.5 API parameter checking 5.7 Reentrancy 7.3.4 API service IDs 7.4.19 I2c_InitHwUnit 7.4.20 I2c_DeinitHwUnit 7.7.1 I2C notification functions (add code examples for each function) 8.1 SCB

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2025-04-21

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email:

erratum@infineon.com

Document reference

002-31274 Rev. *M

Important notice

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics (“Beschaffenheitsgarantie”).

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer’s compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer’s products and any use of the product of Infineon Technologies in customer’s applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer’s technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies’ products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.