SPI 3.0 handler/driver user guide

TRAVEO™ T2G family

About this document

Scope and purpose

This guide describes the architecture, configuration, and use of the serial peripheral interface (SPI) handler/driver. This document explains the functionality of the driver and provides a reference to the driver's API.

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

Intended audience

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

Document structure

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

Chapter 2 Using the SPI handler/driver details the steps on how to use the SPI handler/driver in your application.

Chapter 3 Structure and dependencies describes the file structure and the dependencies for the SPI handler/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 SPI handler/driver.

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

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

Abbreviations and definitions

Table 1 Abbreviation

Abbreviation	Description
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASIL	Automotive Safety Integrity Level
AUTOSAR	Automotive Open System Architecture
Basic Software	Standardized part of software which does not fulfill a vehicle functional job.
DEM	Diagnostic Event Manager
DET	Default Error Tracer
GCE	Generic Configuration Editor

About this document

Abbreviation	Description
EB tresos Studio	Elektrobit Automotive configuration framework
ISR	Interrupt Service Routine
µC	Microcontroller
MCAL	Microcontroller Abstraction Layer
MPU	Memory Protection Unit
PCLK	Peripheral Clock
SPI	Serial Peripheral Interface
SCB	Serial Communication Block
UTF-8	8-Bit Universal Character Set Transformation Format

Related documents

AUTOSAR requirements and specifications

Bibliography

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

[2] Specification of SPI handler/driver, 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.

[5] Specification of memory mapping, AUTOSAR release 4.2.2

Elektrobit automotive documentation

Bibliography

[6] EB tresos Studio for ACG8 user’s guide.

Hardware documentation

The hardware documents are listed in the delivery notes.

Related standards and norms

Bibliography

[7] Layered software architecture, AUTOSAR release 4.2.2.

Table of contents

Table of contents

Table of contents

Table of contents

1 General overview

1.1 Introduction to the SPI handler/driver

The SPI handler/driver is a set of software routines, which enables you to support SPI communication on special output pins of the CPU.

The SPI handler/driver provides services for reading from and writing to devices connected via SPI buses. The SPI handler/driver provides access to SPI communication for multiple users (e.g., EEPROM, watchdog, and I/O ASICs). Only SPI Master mode and full-duplex operation are supported.

The SPI handler/driver provides three levels of scalable functionality as specified in the AUTOSAR *Specification of SPI handler/driver* [2]:

- Level 0 is a simple synchronous SPI handler/driver using a FIFO policy for multiple accesses.
- Level 1 is a basic asynchronous SPI handler/driver supporting interruptible sequences and priority-based scheduling.
- Level 2 is an enhanced SPI handler/driver supporting one hardware peripheral using synchronous transfers as well as asynchronous transfers for the other peripherals.

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

The SPI handler/driver conforms to the AUTOSAR standard and is implemented according to the AUTOSAR *Specification of SPI handler/driver* [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

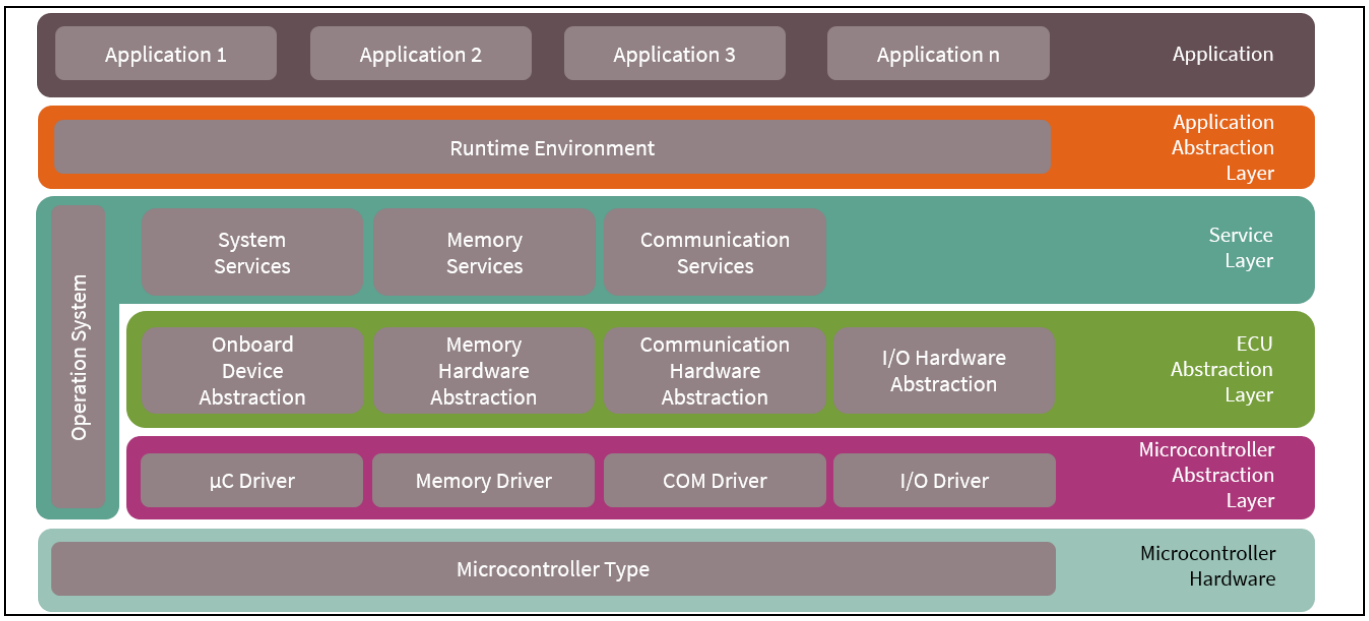

Figure 1 Overview of AUTOSAR software layers

Figure 1 depicts the layered AUTOSAR software architecture. The SPI handler/driver (Figure 2) is part of the microcontroller abstraction layer (MCAL), the lowest layer of basic software in the AUTOSAR environment.

For an exact overview of the AUTOSAR layered software architecture, see *layered software architecture* [7].

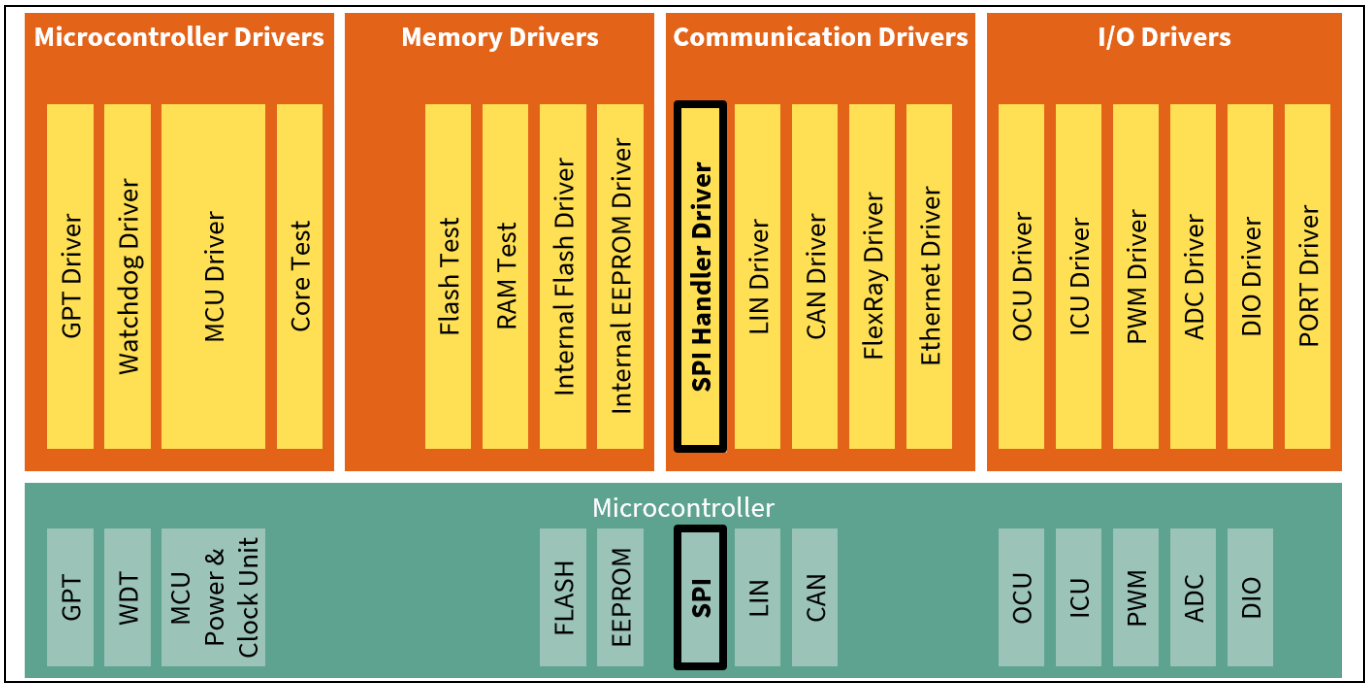

Figure 2 SPI handler/driver in MCAL layer

1.4 Supported hardware

This version of the SPI handler/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 modules Base, Dio, Make, Mcu, Port and Resource are needed for proper functionality of the SPI handler/driver.

1.6 Character set and encoding

All source code files of the SPI 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.

1.7 Multicore support

The SPI driver supports multicore type II. The driver also supports multicore type III for some APIs (for example, read-only API or atomic-write API). For each multicore type, see the following sections.

Note: If multicore type III is desired, the section including data related to read-only API or atomic write API must be allocated to the memory which can be read from any cores.

1.7.1 Multicore type

In the following section, type I, type II, and type III are defined as multicore characteristics.

1.7.1.1 Single core only (multicore type I)

For this multicore type, the driver is available only on a single core. This type is referred as “Multicore Type I”.

Multicore type I has the following characteristic:

- The peripheral channels are accessed by only one core.

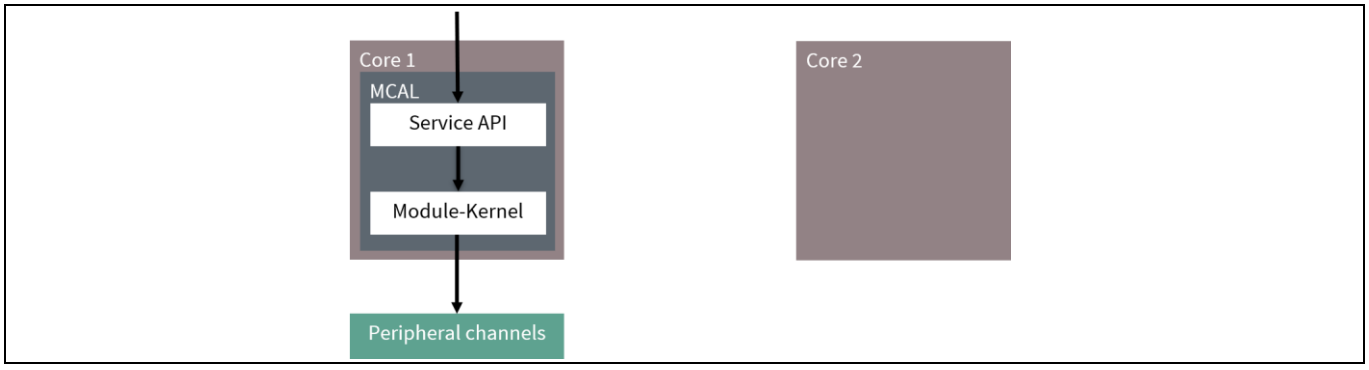

Figure 3 Overview of the multicore type I

1.7.1.2 Core dependent instances (multicore type II)

For this multicore type, the driver has the core-dependent instances with individually allocable hardware. This type is referred as “Multicore Type II”.

Multicore type II has the following characteristics:

- The driver code is shared among all cores:
  - A common binary is used for all cores.
  - A configuration is common for all cores.
- Each core runs an instance of the driver.
- Peripheral channels and their data can be individually allocated to cores, but cannot be shared among cores.
- One core will be the master, and the master core is needed to be initialized first:
  - Cores other than the master core are called satellite cores.

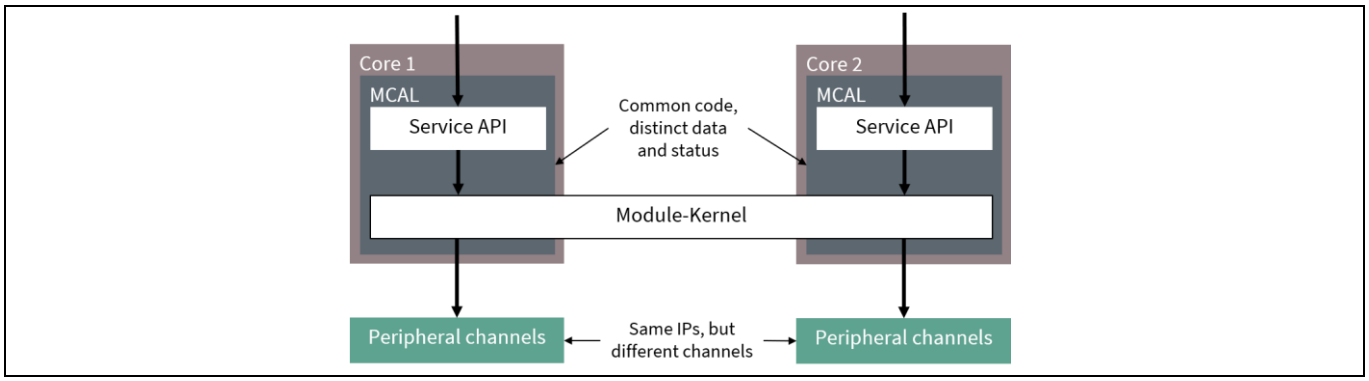

Figure 4 Overview of the multicore type II

1.7.1.3 Core independent instances (multicore type III)

For this multicore type, the driver has the core independent instances with globally available hardware. This type is referred as “Multicore Type III”.

Multicore type III has the following characteristics:

- The code of the driver is shared among all cores:
  - A common binary is used for all cores.
  - A configuration is common for all cores.
- Each core runs an instance of the driver.
- Peripheral channels are globally available for all cores.

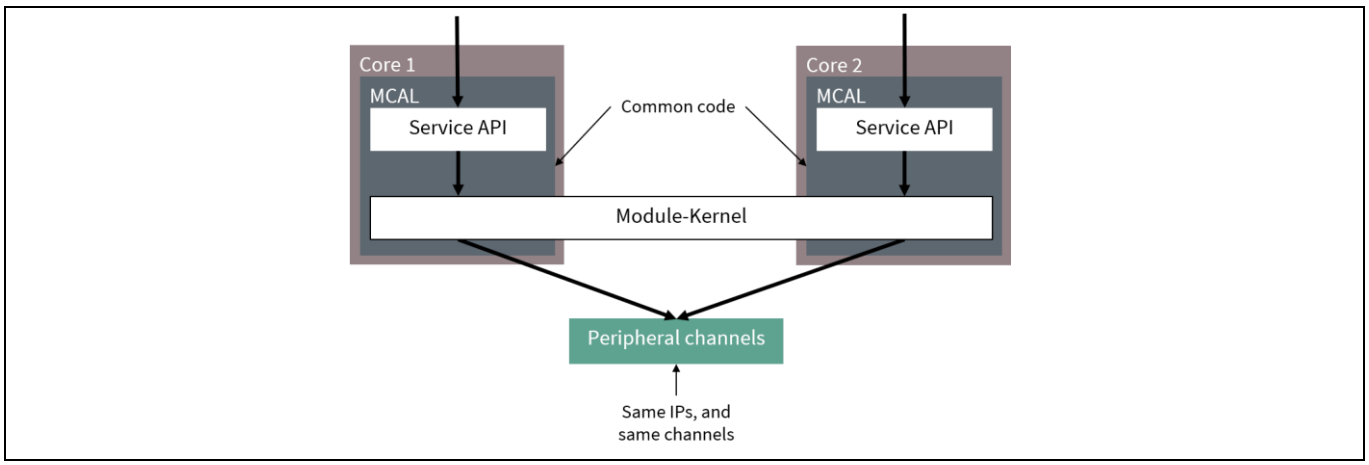

Figure 5 Overview of the multicore type III

1.7.2 Virtual core support

The SPI driver supports a number of cores. The configured cores need not be equal to the physical cores.

The SPI driver asks a configurable callout function (`SpiGetCoreIdFunction`) to know the core that is currently executing the code. This function can be implemented in the integration scope. The function can be written such that it does not return the physical core, but instead returns the SW partition ID, OS application ID, or any attribute/parameter. By interpreting these as the core, the SPI driver can support multiple SW partitions on a single physical core.

2 Using the SPI handler/driver

This chapter describes all necessary steps to incorporate the SPI handler/driver into your application.

2.1 Installation and prerequisites

Note: Before continuing with this chapter, see the EB tresos Studio for ACG8 user's guide [6]. You can find the required basic information about the installation procedure of EB tresos AUTOSAR components and the use of the EB tresos and the EB tresos AUTOSAR build environment. You will also find information on how to set up and integrate your own application within the EB tresos AUTOSAR build environment there.

The installation of the SPI handler/driver corresponds with the general installation procedure for EB tresos AUTOSAR components given in the documents mentioned above.

This document assumes that you have set up your project using the application template. This template provides the necessary folder structure, project, and makefiles needed to configure and compile your application within the build environment. You must be familiar with the use of the command shell.

2.2 Configuring the SPI driver

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

2.2.1 Architecture specifics

- `SpiSetupDelay`: Specifies the timing to start transmission after chip select is activated.
- `SpiHoldDelay`: Specifies the timing of chip select to be inactive after a transmission is finished.
- `SpiDeselect`: Specifies the timing of chip select to be active again after being inactive.
- `SpiUseDma`: Enables or disables the DMA channel for communication.
- `SpiUseFifo`: Enables or disables the transmission using the FIFO functionality.
- `SpiDmaChannelRx`: Specifies the DMA channel to be used for receiving data.
- `SpiDmaChannelTx`: Specifies the DMA channel to be used for sending data.
- `SpiForceOverwrite`: Enables or disables forced overwrite of the control register.
- `SpiClockRef`: Specifies the frequency for the specific transmission unit.
- `SpiErrorCalloutFunction`: Specifies the error callout function.
- `SpiIncludeFile`: Specifies a file that must be included by *Spi_ExternalInclude.h*.
- `SpiCoreAssignment`: Specifies the reference to a container of `SpiCoreConfiguration` to select an assignment core for a channel and an external device.
- `SpiCoreConsistencyCheckEnable`: Enables or disables the core consistency check during runtime.
- `SpiGetCoreIdFunction`: Specifies the API to be called to get the core ID.
- `SpiMasterCoreReference`: Specifies the reference to a container of `SpiCoreConfiguration` to select the master core configuration.
- `SpiCoreConfigurationId`: Specifies a logical number of the core ID
- `SpiCoreId`: Specifies the core ID. This ID is returned from the configured `SpiGetCoreIdFunction` to identify the executing core.

2.3 Adapting your application

To use the SPI handler/driver in your application, include the header files of SPI and PORT driver by adding the following lines of code in your source file:

```
#include "Mcu.h"  /* AUTOSAR MCU Driver */
#include "Port.h" /* AUTOSAR PORT Driver */
#include "Spi.h"  /* AUTOSAR SPI Handler/Driver */
```

This publishes all required function and data prototypes and symbolic names of the configuration into the application.

To use the SPI handler/driver, the appropriate port pins, SCB clock setting and SPI interrupts must be configured in PORT driver, MCU driver and OS. For detailed information, see Hardware resources.

Initialization of MCU, PORT, and SPI handler/driver needs to be done in the following order:

For the master core:

```
Mcu_Init(&Mcu_Config[0]);
Port_Init(&Port_Config[0]);
Spi_Init(NULL_PTR);
```

For the satellite core:

```
Mcu_Init(&Mcu_Config[0]);
Spi_Init(NULL_PTR);
```

The function `Port_Init()` is called with a pointer to a structure of type `Port_ConfigType`, which is published by the PORT driver itself. This function must be called on the master core only.

The master core must be initialized prior to the satellite core. All cores must be initialized with the same configuration.

If level 1 or level 2 functionality is used, an interrupt service routine must be configured in the AUTOSAR OS for each asynchronous SPI peripheral as described in Interrupts.

When using level 2 functionality and the “polling” asynchronous mode, you must call the `Spi_MainFunction_Handling` function cyclically. This can either be done by configuring the BSW scheduler accordingly or by calling the `Spi_MainFunction_Handling` function from any other cyclic task. Note that the “polling” mode is the default mode after initialization of the SPI handler/driver when using level 2 functionality. To set “interrupt” mode instead, use the `Spi_SetAsyncMode` API function as described in Spi_SetAsyncMode.

All required input clocks for the configured hardware units (SCB) must be activated prior to initialization of the SPI handler/driver. See MCU driver.

Your application must provide the notification functions and its declarations that you configured. The file containing the declarations must be included using the `SpiDriverConfiguration/SpiIncludeFile` or `SpiDriverConfiguration/SpiUserCallbackHeaderFile` parameter. The `SpiJobEndNotification` function and the `SpiSeqEndNotification` function take no parameters and have void return type:

```
void MyNotificationFunction(void)
{
/* Insert your code here */
}
```

The notification function is called from an interrupt or polling context and synchronous transmission process.

2.4 Starting the build process

Do the following to build your application.

Note: For a clean build, you must use the build command with target `clean_all` before (make clean_all).

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

   ```
   > make generate
   ```

2. Type the following command to resolve required file dependencies:

   ```
   > make depend
   ```

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

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

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

2.5 Measuring stack consumption

Do the following to measure stack consumption. It requires the Base module for proper measurement.

Note: All files (including library files) should be rebuilt with the dedicated 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 Starting the build process.
4. Measure the stack consumption by following the instructions given in the release notes.

2.6 Memory mapping

The *Spi_MemMap.h* file in the *$(TRESOS_BASE)/plugins/MemMap_TS_T40D13M0I0R0/include* directory is a sample. This file is replaced by the file generated by MEMMAP module. Input to MEMMAP module is generated as *Spi_Bswmd.arxml* in the *$(PROJECT_ROOT)/ output/generate_swcd/swcd* directory of your project folder.

2.6.1 Memory allocation keyword

- `SPI_START_SEC_CODE_ASIL_B` / `SPI_STOP_SEC_CODE_ASIL_B`

  The memory section type is CODE. All executable code is allocated in this section.
- `SPI_START_SEC_CONST_ASIL_B_UNSPECIFIED` / `SPI_STOP_SEC_CONST_ASIL_B_UNSPECIFIED`

  The memory section type is CONST. All configuration data is allocated in this section.
- `SPI_CORE[SpiCoreConfigurationId]_START_SEC_VAR_CLEARED_ASIL_B_LOCAL_UNSPECIFIED` / `SPI_CORE[SpiCoreConfigurationId]_STOP_SEC_VAR_CLEARED_ASIL_B_LOCAL_UNSPECIFIED`

The memory section type is VAR. All non-initialized variables with non-alignment are allocated in this section.

- `SPI_CORE[SpiCoreConfigurationId]_START_SEC_VAR_CLEARED_ASIL_B_GLOBAL_UNSPECIFIED / SPI_CORE[SpiCoreConfigurationId]_STOP_SEC_VAR_CLEARED_ASIL_B_GLOBAL_UNSPECIFIED`

  The memory section type is VAR. All non-initialized variables with non-alignment are allocated in this section.

- `SPI_CORE[SpiCoreConfigurationId]_START_SEC_VAR_CLEARED_ASIL_B_LOCAL_32 / SPI_CORE[SpiCoreConfigurationId]_STOP_SEC_VAR_CLEARED_ASIL_B_LOCAL_32`

  The memory section type is VAR. The variable for internal buffers of transmission with 4 bytes alignment are allocated in this section.

- `SPI_CORE[SpiCoreConfigurationId]_START_SEC_VAR_CLEARED_ASIL_B_DMA_READBUFF_32 / SPI_CORE[SpiCoreConfigurationId]_STOP_SEC_VAR_CLEARED_ASIL_B_DMA_READBUFF_32`

  The memory section type is VAR. The variable for internal buffers of transmission with 4 bytes alignment are allocated in this section.

- `SPI_CORE[SpiCoreConfigurationId]_START_SEC_VAR_CLEARED_ASIL_B_DMA_WRITEBUFF_32 / SPI_CORE[SpiCoreConfigurationId]_STOP_SEC_VAR_CLEARED_ASIL_B_DMA_WRITEBUFF_32`

  The memory section type is VAR. The variable for internal buffers of transmission with 4 bytes alignment are allocated in this section.

- `SPI_CORE[SpiCoreConfigurationId]_START_SEC_VAR_INIT_ASIL_B_LOCAL_UNSPECIFIED / SPI_CORE[SpiCoreConfigurationId]_STOP_SEC_VAR_INIT_ASIL_B_LOCAL_UNSPECIFIED`

  The memory section type is VAR. All initialized variables with non-alignment are allocated in this section.

- `SPI_CORE[SpiCoreConfigurationId]_START_SEC_VAR_INIT_ASIL_B_GLOBAL_UNSPECIFIED / SPI_CORE[SpiCoreConfigurationId]_STOP_SEC_VAR_INIT_ASIL_B_GLOBAL_UNSPECIFIED`

  The memory section type is VAR. All initialized variables with non-alignment are allocated in this section.

2.6.2 Memory allocation and constraints

All memory sections that store init or uninit status must be zero-initialized before any driver function is executed on any core. If core consistency checks are disabled, inconsistent parameters would be detected and reported by PPU and SMPU.

- `SPI_CORE[SpiCoreConfigurationId]_START_VAR_[INIT_POLICY]_ASIL_B_LOCAL_[ALIGNMENT] / SPI_CORE[SpiCoreConfigurationId]_STOP_VAR_[INIT_POLICY]_ASIL_B_LOCAL_[ALIGNMENT]`

  This section is read/write accessed only from the core represented by `SpiCoreConfigurationId`. Therefore, this section can be allocated to any RAM region. It is recommended to allocate the section to cache-able SRAM, not TCRAM.

- `SPI_CORE[SpiCoreConfigurationId]_START_VAR_[INIT_POLICY]_ASIL_B_GLOBAL_[ALIGNMENT] / SPI_CORE[SpiCoreConfigurationId]_STOP_VAR_[INIT_POLICY]_ASIL_B_GLOBAL_[ALIGNMENT]`

  This section is read/write accessed from the core represented by `SpiCoreConfigurationId` and read accessed from the other cores. Therefore, this section must not be allocated to TCRAM. For the core represented by `SpiCoreConfigurationId`, this section must be allocated to either non-cache-able or write-through cache-able SRAM area. For performance, it is recommended to allocate the section to write-through cache-able SRAM. For the other cores, this section must be allocated to non-cache-able SRAM area.

- `SPI_CORE[SpiCoreConfigurationId]_START_VAR_[INIT_POLICY]_ASIL_B_GLOBAL_[ALIGNMENT] / SPI_CORE[SpiCoreConfigurationId]_STOP_VAR_[INIT_POLICY]_ASIL_B_GLOBAL_[ALIGNMENT]`

For multicore type III, this section is read accessed from other cores. So, this section must not be allocated to TCRAM. For the core represented by `SpiCoreConfigurationId`, this section must be allocated to either non-cache-able or write-through cache-able SRAM area. For performance, it is recommended to allocate the section to non-cache-able SRAM. For the other cores, this section must be allocated to non-cache-able SRAM area.

- `SPI_CORE[SpiCoreConfigurationId]_START_VAR_[INIT_POLICY]_ASIL_B_DMA_READBUFF_[ALIGNMENT] / SPI_CORE[SpiCoreConfigurationId]_STOP_VAR_[INIT_POLICY]_ASIL_B_DMA_READBUFF_[ALIGNMENT]`
  - When using DMA:
    The section is allocated to a user-specific memory region configured by the CPU's memory protection unit (MPU) as non-cache-able.
  - When not using DMA:
    There is no restriction.
- `SPI_CORE[SpiCoreConfigurationId]_START_VAR_[INIT_POLICY]_ASIL_B_DMA_WRITEBUFF_[ALIGNMENT] /`
- `SPI_CORE[SpiCoreConfigurationId]_STOP_VAR_[INIT_POLICY]_ASIL_B_DMA_WRITEBUFF_[ALIGNMENT]`
  - When using DMA:
    For the core represented by `SpiCoreConfigurationId`, the section is allocated to a user-specific memory region configured by the MPU as write-through cache-able or non-cache-able. For performance, it is recommended to allocate the section to non-cache-able SRAM. For the other cores, there is no restriction.
  - When not using DMA:
    There is no restriction.
- The section that contains external buffers (EB) used for RX:
  - When using DMA:
    The section is allocated to a user-specific memory region configured by the MPU as non-cache-able.
  - When not using DMA:
    There is no restriction.
- The section that contains EB used for TX:
  - When using DMA:
    For the core which access the EB, the section is allocated to a user-specific memory region configured by the MPU as write-through cache-able or non-cache-able. For performance, it is recommended to allocate the section to non-cache-able SRAM. For the other cores, there is no restriction.
  - When not using DMA:
    There is no restriction.
- STACK section:

  TCRAM has dedicated memory for each core at the same address, and because of its performance it is recommended to allocate STACK to TCRAM.

Note: The CPU has an individual cache that is not shared with the DMA bus master. Therefore, ensure that data related to DMA is in specific region where it can be accessed by the DMA. Besides some sections need to be allocated in specific memory regions. This driver does not support the use of data related to DMA placed in CPU's tightly coupled memories (TCMs) and internal video RAM (VRAM).

Note: *This restriction is applied only to Arm® Cortex®-M7 devices because they include TCMs, VRAM and inner cache. There is no restriction when using Cortex®-M4 devices.*

Note: *All buffers accessed by DMA require 4-byte alignment.*

For details of `INIT_POLICY` and `ALIGNMENT`, see the *Specification of memory mapping* [5].

3 Structure and dependencies

The SPI handler/driver consists of static, configuration, and generated files.

3.1 Static files

- *$(PLUGIN_PATH)=$(TRESOS_BASE)/plugins/Spi_TS_** is the path to the SPI handler/driver plugin.
- *$(PLUGIN_PATH)/lib_src* contains all static source files of the SPI handler/driver. These files contain the functionality of the driver that does not depend on the current configuration. The files are grouped into a static library.
- *$(PLUGIN_PATH)/src* comprises configuration-dependent source files or special derivate 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 SPI handler/driver is enabled.

- *$(PLUGIN_PATH)/include* is the basic public include directory needed by the user to include *Spi.h*.
- *$(PLUGIN_PATH)/autosar* directory contains the AUTOSAR ECU parameter definition with vendor, architecture and derivative-specific adaptations to create a correct matching parameter configuration for the SPI handler/driver.

3.2 Configuration files

The configuration of the SPI handler/driver is done via EB tresos Studio. The file containing the SPI handler/driver's configuration is named *Spi.xdm* and is in the directory *$(PROJECT_ROOT)/config*. This file serves as the input for the generation of the 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* sub folder of your project folder.

- *include/Spi_Cfg.h*
- *include/Spi_Cfg_Der.h*
- *include/Spi_ExternalInclude.h*
- *src/Spi_PBCfg.c*
- *src/Spi_PBCfg_Der.c*
- *src/Spi_Irq.c*
- *src/Spi_MainFunction_Handling.c*

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.*

- *swcd/Spi_Bswnd.arxml*

Note: *Additional steps are required for the generation of BSW module description. In EB tresos Studio, follow the menu path **Project** > **Build Project** and click **generate_swcd**.*

3.4 Dependencies

3.4.1 PORT driver

Although the SPI handler/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 SPI handler/driver will show undefined behavior. The PORT driver needs to be initialized before the SPI handler/driver is initialized.

3.4.2 MCU driver

The MCU driver needs to be initialized and all MCU clock reference points referenced by the hardware units (SCB) via the configuration parameter `SpiClockRef` must have been activated (via calls of MCU API functions) before initializing the SPI handler/driver. `Mcu_GetCoreID` can optionally be set to the configuration parameter `SpiGetCoreIdFunction`. See the *MCU driver's user guide* for details.

Note that the clock, prescaler, or PLL settings are controlled by the MCU driver. There are no shared resources with the SPI handler/driver. Depending on the configuration, changes in the clock settings may affect the operation of the SPI handler/driver.

3.4.3 DIO driver

The SPI handler/driver allows you to optionally control chip select by the software using a GPIO pin. This can be configured by setting the `SpiCsSelection` parameter of an external device to `CS_VIA_GPIO`. In this case, the SPI handler/driver uses the multicore DIO driver to control the DIO channel configured in the `SpiCsIdentifier` parameter for chip select operation. This cannot be used single core DIO.

3.4.4 AUTOSAR OS

The AUTOSAR operating system handles the interrupts used by the SPI handler/driver. `GetCoreID` can optionally be set to the configuration parameter `SpiGetCoreIdFunction`. See Interrupts for more information.

3.4.5 BSW scheduler

The BSW scheduler handles the critical sections that are used by the SPI handler/driver.

3.4.6 DET

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

This driver reports DET error codes as instance 0.

3.4.7 DEM

If the DEM event report is enabled in the SPI module configuration, the DEM needs to be installed, configured, and integrated into the application as well.

To enable DEM support in the SPI handler/driver, the `SPI_E_HARDWARE_ERROR` production error needs to be defined in the DEM configuration in the `SpiDemEventParameterRefs` container.

3.4.8 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 configuration parameter `SpiErrorCalloutFunction`.

3.4.9 DMA

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

The SPI module 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 SPI.

4 EB tresos Studio configuration interface

The GUI is not part of this delivery. For further information, see *EB tresos Studio for ACG8 user's guide* [6].

4.1 General configuration

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

- `SpiDmaErrorHandlingPolling` specifies the DMA error handling mode. When enabled in the interrupt mode, the DMA error is handled by the polling mode.
- `SpiCancelApi` enables or disables the cancel API function.
- `SpiChannelBuffersAllowed` is the allowed buffers type to be used.
  - `0`: Internal buffers only
  - `1`: External buffers only
  - `2`: Both buffers
- `SpiDevErrorDetect` enables or disables the DET functionality for the SPI handler/driver.
- `SpiHwStatusApi` enables or disables the hardware status API function.
- `SpiInterruptibleSeqAllowed` enables or disables the interruptible sequences.

  If `SpiLevelDelivered` is set to '1' or '2', this parameter is editable.
- `SpiLevelDelivered` is the level of driver to be used.
  - `0`: Level 0 simple synchronous mode
  - `1`: Level 1 basic asynchronous mode
  - *2*: Level 2 enhanced mode
- `SpiSupportConcurrentSyncTransmit` specifies whether concurrent `Spi_SyncTransmit` calls for different sequences is supported.
- `SpiUserCallbackHeaderFile` specifies the header file names that will be included by the SPI driver.
- `SpiVersionInfoApi` specifies whether the API function `Spi_GetVersionInfo` is available.

4.2 SPI driver configuration

- `SpiMaxChannel` is not used. It is calculated and generated by the generator automatically.
- `SpiMaxJob` is not used. It is calculated and generated by the generator automatically.
- `SpiMaxSequence` is not used. It is calculated and generated by the generator automatically.

4.2.1 Channel configuration

Note that the channel name and ID of a channel must be unique.

- `SpiChannelId` is the ID for the channel. It is used as a parameter for API functions.

Note: Channel IDs must be zero-based and consecutive.

- `SpiCoreAssignment` specifies the reference to `SpiCoreConfiguration` for the channel core assignment.

Note: `SpiCoreAssignment` must have the target's `SpiCoreConfiguration` setting.

- `SpiChannelType` is the type of buffering to be used for this channel.
  - `IB`: Internal buffering

  - EB: External buffering

Note: A selectable value depends on the SpiChannelBuffersAllowed setting.

- SpiDataWidth is the data width setting for transmission in bits.

Note: List of values available for configuration depends on the subderivative.

Note: If SpiDataWidth=8-bit and the total data is more than 32 bytes, the data is divided into several portions; the SPI driver sends each data portion to FIFO. So, if the SPI interrupt is blocked by another interrupt or the main function is not being called frequently, FIFO empty occurs and CS will be de-asserted. To avoid this situation, do one of the following:
- Set the SPI interrupt as a high-priority interrupt
- Call Spi_MainFunction_Handling frequently
- Set the SPI baudrate low
- Use SpiDataWidth=16-bit/32-bit.
- Use DMA (SpiUseDma)

- SpiDefaultData is the default value setting for transmission.

Note: The configured value must be within the range configured by SpiDataWidth.

Note: If SpiDefaultData is disabled, the default value setting is 0.

- SpiEbMaxLength is the maximum size of a data buffer (Range: 1 to 65535); type Spi_NumberOfDataType.
  If EB is selected as SpiChannelType and 1 or 2 is selected as SpiChannelBuffersAllowed, this parameter is editable.
- SpiAlignedBuffer requires a data-width-aligned external buffer
  If a data-width-aligned buffer is required, Spi_SetupEB will check the assigned data buffer. The required 1-, 2-, or 4-byte alignment depends on the declared data width.
  The alignment is required to allow DMA-supported transmission of the channel.
- SpiIbNBuffers is the size of the data buffers (Range: 1 to 65535; type Spi_NumberOfDataType.
  If IB is selected as SpiChannelType and 0 or 2 is selected as SpiChannelBuffersAllowed is, this parameter is editable.

Note: Maximum size differs according to SpiDataWidth. Maximum size is 65535 if SpiDataWidth is 8 bits or less. Maximum size is 32767 if SpiDataWidth is 9 bits to 16 bits. Maximum size is 16383 if SpiDataWidth is 17 bits or more.

- SpiTransferStart is the bit ordering for transmission.
  - LSB: Least significant bit first
  - MSB: Most significant bit first

4.2.2 Job configuration

Note that the name and ID of a Job must be unique.

- SpiHwUnitSynchronous is the job setting for synchronous or asynchronous transmission.
  - SYNCHRONOUS: Synchronous
  - ASYNCHRONOUS: Asynchronous

Note: If the parameter is not set, `SpiJob` uses the driver also in an asynchronous way.

Note: All `SpiJob` parameters that belong to the same external device specified by `SpiDeviceAssignment` will have the same `SpiHwUnitSynchronous` setting.

- `SpiJobEndNotification` specifies the function that will be called by the driver on completion of the job. This function is to be implemented by the user.

  If `SpiJobEndNotification` is blank, the function is not called.

  If `SpiJobEndNotification` is disabled, the function is not called.
- `SpiJobId` is the ID of the job. This value will be assigned to the following symbolic names:
  - The symbolic name derived from the `SpiJob` container short name.
  - The symbolic name derived from the `SpiJob` container short name prefixed with "`Spi_`".
  - The symbolic name derived from the `SpiJob` container short name prefixed with "`SpiConf_SpiJob_`".

Note: Job IDs must be zero-based and consecutive.

- `SpiJobPriority` is the priority for the job; priorities lie in the range of 0 to 3, 0 being the lowest.
- `SpiDeviceAssignment` specifies the external device to be used for the job.
- `SpiChannelList` references to SPI, the channels, and their order within the job.
  - `SpiChannelIndex`: specifies the order of channels within the job.

Note: `SpiChannelIndex` must have the same value as the index of the actual entry in `SpiChannelList`.

  - `SpiChannelAssignment`: specifies a list of channels associated with this Job.

Note: The `SpiDataWidth` for each channel that is assigned in one job must have the same width when using the peripheral chip select (`SpiEnableCs` = enabled and `SpiCsSelection` = CS_VIA_PERIPHERAL_ENGINE).

Note: `SpiTransferStart` for each channel that is assigned in one job must have the same first starting bit.

Note: The total size of all channels' data buffers (`SpiEbMaxLength` and `SpiIbNBuffers`) must not exceed 65535 bytes.

Note: The bytes may be a multiple of units depending on the `SpiDataWidth` entry.
If `SpiDeviceAssignment` selects an external device with DMA support, the channels of the job must allow buffer alignment even if the data width declared is 8 bits or less.

4.2.3 External device configuration

- `SpiForceOverwrite` enables or disables forced overwrite of the control register. When this parameter is enabled, control information in the control register is overwritten even if the transfer is to the same external device.
- `SpiClockRef` is the reference to the clock source configuration, which is set in the MCU driver configuration.

Note: During configuration, an applicable clock will be selected. The runtime system is responsible for activating the selected configuration before using the external device.

- `SpiCoreAssignment` specifies the reference to `SpiCoreConfiguration` for the external device core assignment.

Note: *`SpiCoreAssignment` must have the target's `SpiCoreConfiguration` setting. The same SCB channel cannot be allocated to multiple cores.*

- `SpiBaudrate` is the communication baud rate. This parameter allows using a range of values, from the point of view of the configuration tools, from Hz up to MHz. The value is in Hz.

Note: *The hardware supports discrete baud rates in a range depending on the frequency of source clock as follows:*
`(SpiClockRef.McuClockReferencePointFrequency / (OVSValue+1))`, `OVSValue=3,4,5,…,15`

Note: *You can enter any baud rate value in this range, without respecting the hardware support of the concrete baud rates. The code generator will automatically select the next lower allowed baud rate without reporting a warning.*
The tresos system supports checking and selecting the real baud rate. After entering the expected baud rate, you can let the system calculate its exact value. If the given baud rate cannot be supported, the calculation makes a weighted selection between the next higher or lower baud rates. This weighting prefers four times more deviation for the lower baud rate selection than the higher one. The configuration will support this calculated baud rate.
Before calculation, the clock reference point must be selected and correctly configured. The calculation also works well if the given baud rate is outside the accepted range. In this case, the highest or lowest accepted baud rate will be selected.

- `SpiEnableCs` enables or disables the chip select handling functions. If this parameter is enabled, `SpiCsSelection` provides further details of the type of chip select control; if disabled, `SpiCsSelection` is ignored.

Note: *Even if this parameter is set to disable, the SCB hardware function internally outputs `SPI_SELECT0`. Make sure `SPI_SELECT0` is not output to the outside in the Port driver.*

- `SpiCsSelection` specifies if the chip select is handled automatically by the SCB hardware function or via general-purpose I/O.
  - `CS_VIA_GPIO`: Handled via general-purpose I/O by the SPI driver.
  - `CS_VIA_PERIPHERAL_ENGINE`: Handled automatically by the SCB hardware function. The parameters `SpiSetupDelay`, `SpiHoldDelay`, and `SpiDeselect` take effect on the chip select signal only in this mode.

Note: *When `CS_VIA_GPIO` is selected for this parameter, the SCB unit internally outputs `SPI_SELECT0`. Make sure `SPI_SELECT0` is not output to the outside in the Port driver.*

Note: *If DMA is not used for SCB, the chip select might be de-asserted during a job transmission. To avoid this situation, do either of the following;*
- Use CS_VIA_GPIO (SpiCsSelection)
- Use DMA (SpiUseDma)
- Use data, which is 32 elements or less, for a job

- `SpiCsIdentifier` specifies the chip select pin allocated to this Job. Available pins depend on the setting of `SpiCsSelection`:

  - CS_VIA_GPIO: all configured Dio channels are listed
  - CS_VIA_PERIPHERAL_ENGINE: *SPI_SELECT0...SPI_SELECT3*, depending on the configured SCB

  If SpiEnableCs is enabled, this parameter is editable.
- SpiHwUnit is the hardware unit to be used for this external device.
  - SCB0: SCB Channel 0
  - SCB1: SCB Channel 1
  - ...
  - SCBn: SCB Channel n

Note: Selectable hardware units depend on the subderivative.

Note: If the same SpiHwUnit is set to multiple SpiExternalDevice containers, note the settings of the following parameters.
The chip select pin must be set to each SpiCsIdentifier.
If multiple SpiExternalDevice share the same SCB, the same value must be set for the following parameters:
- SpiCsSelection
- SpiEnableCs
- SpiDmaChannelRx
- SpiDmaChannelTx
If multiple SpiExternalDevice share the same SCB and SpiCsIdentifier, the same value must be set for the following parameters:
- SpiDataShiftEdge
- SpiShiftClockIdleLevel
- SpiCsPolarity
- SpiSetupDelay
- SpiHoldDelay

- SpiCsPolarity specifies the active polarity of the chip select.

  If SpiEnableCs is enabled, this parameter is editable.
  - LOW: Low level
  - HIGH: High level
- SpiDataShiftEdge specifies the data shift edge.
  - LEADING: Leading edge
  - TRAILING: Trailing edge

  If SpiDataShiftEdge is set to LEADING, the SpiSetupDelay must be configured such that the sampling of the first bit takes place after the chip select pin becomes active.
- SpiShiftClockIdleLevel specifies the shift clock idle level.
  - LOW: Low level
  - HIGH: High level
- SpiTimeClk2Cs allows using a range of values from 0 up to 100 microseconds. This parameter is not used and not editable.
- SpiSetupDelay specifies the time in SPI serial clock count to start the transmission after chip select is activated.

  This parameter is only enabled, if SpiEnableCs is enabled. The parameter is editable and effective on the signal only if a hardware-controlled chip select, i.e., if SpiCsSelection is set to CS_VIA_PERIPHERAL_ENGINE.

Note: This parameter will be selected from the selection list. Allowed value depends on `SpiDataShiftEdge`

- `SpiHoldDelay` specifies the time the SPI serial clock count of chip select takes to become inactive after the transmission is completed.

  This parameter is only enabled, if `SpiEnableCs` is enabled. It is only editable and effective on the signal if a hardware-controlled chip select, i.e., if `SpiCsSelection` is set to `CS_VIA_PERIPHERAL_ENGINE`.

Note: This parameter will be selected from the selection list. Allowed value is depend on `SpiDataShiftEdge`

- `SpiDeselect` specifies the time chip select takes to become active again after it is inactive. This parameter is not used and is not editable.
- `SpiUseFifo` enables or disables the transmission using the FIFO functionality. This parameter is fixed to enable and not editable.

Note: FIFO transferable max entries depend on the subderivative. It is Max/4 entries.

- `SpiUseDma` determines whether the DMA controller is used to handle transfers for the specified peripheral.

  If DMA is used for a peripheral, the two configuration parameters, `SpiDmaChannelsRx` and `SpiDmaChannelTx`, must be set to specify the DMA channel for Rx and Tx:

Note: The DMA controller is used only for asynchronous transmission.

Note: DMA operation is not supported for all hardware instances. The configurator will report an error if `SpiUseDma` is enabled and the selected hardware instance does not support DMA transfer.

  - `SpiDmaChannelRx` specifies the DMA channel to be used to handle specified peripheral reception.
  - `SpiDmaChannelTx` specifies the DMA channel to be used to handle specified peripheral transmission.

4.2.4 Sequence configuration

Note that the name and ID of a sequence must be unique.

- `SpiInterruptibleSequence` specifies whether the sequence can be interrupted, i.e., jobs from another sequence may run before the jobs for this sequence depending on the job priorities set.
- If `SpiInterruptibleSeqAllowed` is checked, this parameter is editable.
- `SpiSeqEndNotification` specifies the function that will be called by the driver on completion of the sequence. You need to implement this function.
- If `SpiSeqEndNotification` is blank, the function is not called. If `SpiSeqEndNotification` is disabled, the function is not called.
- `SpiSequenceId` is the ID for the sequence to be used as a parameter for API functions.

Note: Sequence IDs must be zero-based and consecutive.

- `SpiJobAssignment` specifies a list of jobs associated with this sequence.

  Note: Jobs must be ordered in the descending order of their priorities.

  Note: The SPI sequence must not mix synchronous and asynchronous jobs.

Note: The priorities of a job can only be between 0 (lowest) and 3 (highest); therefore, it is not possible to have more than four jobs in a sequence with differing (decreasing) values. Jobs with equal priority will be processed in the order of configuration in the sequence.

4.2.5 SPI DEM event parameter references

This is the container holding the references to DemEventParameter elements that are invoked using the `Dem_ReportErrorStatus` API if the corresponding error (`SPI_E_HARDWARE_ERROR`) occurs.

- `SPI_E_HARDWARE_ERROR` is the reference to the `DemEventParameter` which will be issued when the hardware error has occurred.

4.2.6 SPI published information

This is container holding all SPI-specific published information parameters.

- `SpiMaxHwUnit` specifies the maximum number of different SPI hardware microcontroller serial peripherals (units/buses) available and handled by this SPI handler/driver module. This value is dummy. See the hardware data sheet for the actual number of units.

4.2.7 SpiMulticore

`SpiMulticore` defines the multicore functional configuration of the SPI handler/driver.

- `SpiCoreConsistencyCheckEnable` enables core consistency check during runtime. If enabled, SPI function checks if the provided parameters are allowed on the current core.

Note: Development error detect is enabled in SPI driver to enable this parameter.

- `SpiGetCoreIdFunction` specifies the API to be called to get the core ID. e.g., `GetCoreId()`

Note: `SpiGetCoreIdFunction` must be a valid C function name.

- `SpiMasterCoreReference` specifies the reference to the master core configuration.

Note: `SpiMasterCoreReference` must have the target's `SpiCoreConfiguration` setting.

4.2.8 SpiCoreConfiguration

`SpiCoreConfiguration` defines the core configuration of the SPI driver.

- `SpiCoreConfigurationId` is a zero-based, consecutive integer value. This is used as a logical core ID.

Note: `SpiCoreConfigurationId` must be unique across `SpiCoreConfiguration`.

- `SpiCoreId` is the core ID assigned to channels and external devices. This ID is returned from the configured `SpiGetCoreIdFunction` execution to identify the executing core.

Note: `SpiCoreId` must be unique across `SpiCoreConfiguration`.
`The combination of SpiCoreConfigurationId` and `SpiCoreId` must be unique across `SpiCoreConfiguration`.

4.3 Vendor and driver-specific parameters

4.3.1 Container SpiGeneral

4.3.1.1 SpiErrorCalloutFunction

Description

Error callout function. Syntax:

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

The error callout function is called on every error. The ASIL level of this function limits the ASIL level of the SPI handler/driver.

Type

`FunctionNameParamDef`

4.3.1.2 SpiIncludeFile

Description

A list of file names that will be included within the driver. Any application-specific symbol that is used by the SPI configuration (e.g., error callout function) should be included by configuring this parameter.

Type

StringParamDef

4.4 Other modules

4.4.1 PORT driver

The pins given in Ports and pins must be configured in the PORT driver.

The trigger multiplexer given in DMA and trigger multiplexer must be configured in the PORT driver.

4.4.2 DET

DET must be configured, if default error detection is activated.

4.4.3 AUTOSAR OS

The SPI handler/driver's interrupts (listed in Interrupts) must be configured in the AUTOSAR operating system.

Note: The AUTOSAR OS must only configure those interrupts that are used by the SPI handler/driver.

4.4.4 BSW scheduler

The SPI handler/driver uses the following services of the BSW scheduler (SchM) to enter and leave critical sections

- `SchM_Enter_Spi_SPI_EXCLUSIVE_AREA_[SpiCoreConfigurationId](void)`
- `SchM_Exit_Spi_SPI_EXCLUSIVE_AREA_[SpiCoreConfigurationId](void)`

You must ensure that the BSW scheduler is properly configured and initialized before using the SPI services.

The exclusive area must prevent all tasks or interrupts from calling any SPI API function or SPI interrupt service routine.

5 Functional description

The SPI handler/driver may be used with three different levels of functionality; level 0 offers basic synchronous transmission, level 1 offers asynchronous transmission with job scheduling between multiple sequences, and level 2 offers enhanced features handling both synchronous and asynchronous transmissions. The basic operation of the driver is based on the configuration of channels, Jobs, and sequences. These are described in more detail in this chapter

5.1 Channels, jobs, and sequences

The SPI handler/driver supports one or more channels, Jobs, and sequences to drive different kinds of hardware devices. Data transmission depends on the configuration of these.

Figure 6 shows the correlation between channel, Job, and sequence.

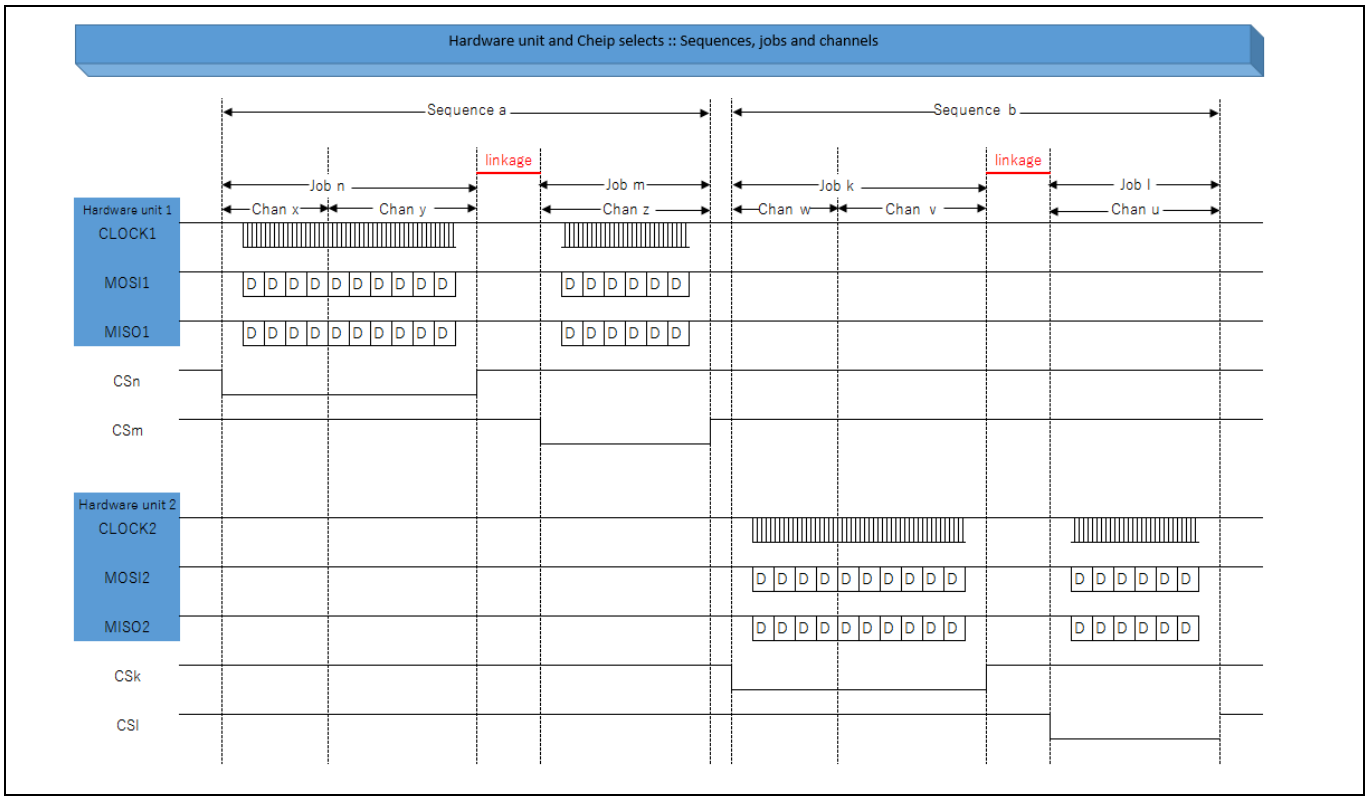

Figure 6 Correlation between sequences, jobs, and channels

5.1.1 Channels

5.1.1.1 General

A channel defines a data channel that can be used to send data to a hardware device. Each channel has a unique identifier. It is possible to have more than one channel set up for one hardware device.

For instance, the following are the channels for an EEPROM device on SPI:

- Channel for command
- Channel for address
- Channel for data

Buffers for the different channels set up can have different sizes and can be located internally in the driver or externally in your application. These are referred to as internally buffered (IB) or externally buffered (EB) channels.

5.1.1.2 Internally buffered channels

Internal buffers (IB) are used for small data transfer devices and daisy chain implementations. The maximum size is defined by Spi_NumberOfDataType. The actual size of the IB to be used must be set in the configuration. This is then fixed for all transmissions using this channel.

The SPI handler/driver provides a transmit buffer for each IB channel. Before starting of a transmission, data needs to be written to the buffer by using the `Spi_WriteIB` function. After that, a synchronous or asynchronous transmission can be started by using `Spi_SyncTransmit` or `Spi_AsyncTransmit` respectively.

Note that the SPI handler/driver is not able to ensure integrity of the data residing in the buffer during transmission. In addition, each request of `Spi_WriteIB` on a channel will overwrite the previous content in its transmit buffer, regardless of whether a transmission has been performed with this data.

The SPI handler/driver provides a receive buffer for the IB channel with the same size as the transmit buffer. The buffer is overwritten with new data at each transmission on that channel. Therefore, make sure that the received data is read before a new transmission on that channel is initiated.

Reading of data from the receive buffer is done by using the `Spi_ReadIB` function, which should only be called after completion of a transmission.

5.1.1.3 Externally buffered channels

Externally buffered (EB) channels can be used to transmit large streams for communication: for EEPROM data read and write, or for controlling complex hardware chips. The maximum size, defined by Spi_`NumberOfDataType`, must be set in the configuration, but the buffer is in the users' application. Before transmission, you must provide the addresses of source and destination buffers together with their length by using the API function `Spi_SetupEB`.

For EB channels, you must the buffer. You must ensure the consistency of the buffered data. You also provide the pointers to the buffers for reception and transmission as well as the size of those buffers. The size should not exceed the maximum size configured.

A transmission is initiated in the same way as for IB channels, by calling either `Spi_SyncTransmit` or `Spi_AsyncTransmit` operation.

Note: Before using the channel for transmit and receive operations, an application must call `Spi_SetupEB` at least once to configure the channel's parameters such as channel length, transmit, and receive buffer pointers. If data is sent without calling the function `Spi_SetupEB`, the single default data is transmitted. The default data is set by the configuration parameter `SpiDefaultData` and the width is set by the configuration parameter `SpiDataWidth`. If the channel's length or the transmit and receive buffer's location has changed in the application, it is mandatory to reconfigure the channel's parameters with `Spi_SetupEB` before using the channel. If the channel's length, transmit and receive buffer's location are not changed, it is not necessary to call `Spi_SetupEB`. While updating the channels parameters, the application must make sure that the channel is not currently being used by driver.
The channel's status can be identified by the status of `SpiJob` from `Spi_GetJobResult`. All `SpiJobs` that share the channel must be checked. `Spi_SetupEB` can be called if each JobResult is either SPI_JOB_OK or SPI_JOB_FAILED.

5.1.1.4 Data buffers

The TX buffer that is passed to a channel (using `Spi_WriteIB` or `Spi_SetupEB`) must contain the data in a certain manner, depending on the setting of `SpiDataWidth`. The RX buffer is filled the same way during transmission.

- `SpiDataWidth` <= 8
- One byte (B0) of the buffer represents one data element (e.g., d0..d7) consisting of not more than 8 bits each.
- 8 < `SpiDataWidth` <= 16
- Two bytes (B0, B1) of the buffer represent one data element (e.g., d0..d15) consisting of more than 8 and not more than 16 bits each. The lower byte (B0) must be filled with the lower bits of the data element (d0..d7). The higher byte (B1) must be filled with the remaining bits (d8..d15), starting at the lowest bit of B1.
- 16 < `SpiDataWidth` <= 32
- Four bytes (B0, B1, B2, B3) of the buffer represent one data element (e.g., d0..d31) consisting of more than 16 and not more than 32 bits each. The lowest byte (B0) must be filled with the lowest bits of the data element (d0..d7). The next byte (B1) must be filled with the next bits (d8..d15), and so on. If `SpiDataWidth` <= 24, the data in fourth byte (B3) is ignored (TX case) or filled with zero (RX case). All 4 bytes (B0, B1, B2, B3) are allocated even if `SpiDataWidth` <= 24.

The addresses of the TX and RX buffers must be integer multiples of the data element size, i.e.,:

- `SpiDataWidth` <= 8: any address
- 8 < `SpiDataWidth` <= 16: address mod 2 must be 0
- 16 < `SpiDataWidth` <= 32: address mod 4 must be 0

5.1.2 Jobs

A Job is composed of one or several channels with the same chip select (is not released during the processing of the Job). A Job is considered atomic and therefore cannot be interrupted by another Job. A Job has an assigned priority.

A Job contains at least one channel. It can contain more than one channel. These channels are configured in a list for that Job. A Job has a priority that can be from 0 up to 3, where 0 is the lowest priority. A Job can belong to more than one sequence.

A chip select is attached to a Job definition. The chip select is set at the beginning of the Job transmission and released at the end of the Job.

At the end of the Job, a '`SpiJobEndNotification`' is called, if configured.

5.1.3 Sequences

A sequence is a number of consecutively transmitted Jobs. Jobs configured for a sequence must be in the order of priority starting with the highest priority first.

If a level 1 or level 2 driver is configured, sequences may be configured as either interruptible or non-interruptible. If a sequence is interruptible and asynchronously transmitted, Jobs from another sequence may run depending on priority.

If a sequence is configured as non-interruptible, a new sequence is scheduled after the transmitting sequence, if the sequences are using the same hardware unit. If different hardware units are used, more than one sequence can be transmitted at the same time.

Note that while sequences may be configured to have shared Jobs, sequences that have shared Jobs may not be transmitted at the same time, i.e., the driver will reject a request to transmit a sequence if it has Jobs that are configured as part of a sequence already in transmission.

At the end of the sequence, a '`SpiSeqEndNotification`' is called, if configured.

5.1.4 Scheduling

Jobs have assigned priorities. They will have decreasing priorities if they are linked in a sequence, i.e., the first Job will have the highest priority.

If an interruptible sequence is configured, the system will check for another pending sequence at the end of a Job transmission. If there is a Job for the same hardware with a higher priority, this Job will be transmitted next.

When using interruptible sequences, note that the same channels should not be configured in those sequences, as otherwise the data of the channels may be overwritten by a Job with a higher priority before you have read the data. You must make sure of the consistent use of channels.

5.2 Inclusion

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

5.3 Initialization

The SPI handler/driver must be initialized on each core before use by calling the API function `Spi_Init`. The module PORT must also be initialized in a similar way.

Note: *`Spi_Init()` must be called on the master core before any other cores are initialized. If `Spi_Init()` is called on the satellite core, the master core must be already initialized. If no SCB channel is assigned to the satellite core, `Spi_Init()` is not required on that core.*

5.4 De-initialization

The SPI handler/driver can be de-initialized once on each core after use. De-Initialization of the SPI handler/driver is made by calling `Spi_DeInit()`.

Note: *`Spi_DeInit()` must be called on the master core after all satellite cores are de-initialized. If `Spi_DeInit()` is called on the satellite core, the master core must be already initialized. The integrated system must prevent other cores from calling the SPI API while `Spi_DeInit()` is being called.*

5.5 Runtime reconfiguration

All configuration parameters can be not changed at runtime.

5.6 API parameter checking

The driver's services perform regular error checks.

When an error occurs, the error hook routine (configured via `SpiErrorCalloutFunction`) is called and the error code, service ID, module ID, and instance ID are passed 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.

The AUTOSAR specified development error and vendor-specific development error checks are performed by the services of the SPI handler/driver.

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

5.6.1 AUTOSAR specified development errors

Any API function - except `Spi_Init` and `Spi_GetVersionInfo` - called with the driver in uninitialized state reports the error code `SPI_E_UNINIT`.

If `Spi_Init` is called on the master core when any cores are already initialized, the error code `SPI_E_ALREADY_INITIALIZED` is reported.

If the function `Spi_Init` is called on the satellite core when the satellite core is already initialized, the error code `SPI_E_ALREADY_INITIALIZED` is reported.

If the functions `Spi_WriteIB`, `Spi_ReadIB` or `Spi_SetupEB` are called with an incorrect channel parameter, the error code `SPI_E_PARAM_CHANNEL` is reported.

If the function `Spi_GetJobResult` is called with the wrong Job parameter, the error code `SPI_E_PARAM_JOB` is reported.

If the function `Spi_GetSequenceResult`, `Spi_AsyncTransmit`, `Spi_SyncTransmit`, and `Spi_Cancel` are called with the wrong parameter sequence, the error code `SPI_E_PARAM_SEQ` is reported.

If the function `Spi_SetupEB` is called with the wrong parameter length, the error code `SPI_E_PARAM_LENGTH` is reported.

If the function `Spi_GetHWUnitStatus` is called with the wrong parameter `HwUnit`, the error code `SPI_E_PARAM_UNIT` is reported.

If the function `Spi_GetVersionInfo` is called with a NULL pointer, the error code `SPI_E_PARAM_POINTER` is reported.

If one of the functions `Spi_Init`, `Spi_DeInit`, `Spi_SetAsyncMode`, `Spi_GetStatus` or `Spi_MainFunction_Handling` is called and the core ID is invalid, the error code `SPI_E_INVALID_CORE` is reported.

If one of the functions `Spi_WriteIB`, `Spi_ReadIB`, `Spi_SetupEB`, `Spi_SyncTransmit`, `Spi_AsyncTransmit`, `Spi_Cancel`, `Spi_Terminate`, `Spi_ChangeOvsSetting` or the interrupt handlers is called from unexpected core, the function is executed on unexpected core, the error code `SPI_E_INVALID_CORE` is reported.

If the function `Spi_DeInit` is called from master core when not all cores are uninitialized, the error code `SPI_E_BUSY` is reported.

5.6.2 Vendor-specific development errors

The error code SPI_E_INVALID_HW is reported if the `Spi_SyncTransmit` function is called for a sequence having Jobs for asynchronous hardware units or the `Spi_AsyncTransmit` function is called for a sequence having Jobs for the synchronous hardware unit.

If the Spi_SetupEB function is called with buffer pointers that are not aligned and the buffer alignment required (SpiAlignedBuffer is checked), the error code *SPI_E_PARAM_POINTER* is reported. A buffer pointer is aligned if <buffer address> mod <required bytes per data unit> = 0. The number of required bytes per data unit depends on SpiDataWidth (see the section called data buffers).

If the function Spi_SetAsyncMode is called with an undefined parameter value buffer, the error code SPI_E_PARAM_BAD_MODE is reported.

If the function Spi_ReadIB is called with the parameter DataBufferPointer as NULL pointer, the error code SPI_E_PARAM_POINTER is reported.

The vendor-specific function Spi_GetBufferStatus reports SPI_E_UNINIT if the driver is not in the initialized state, SPI_E_PARAM_CHANNEL if an invalid channel parameter, and SPI_E_PARAM_POINTER if NULL has been passed to one or more of its remaining parameters.

If the Spi_AsyncTransmit function is called with the parameter sequence using the same HwUnit while transmitting with the Spi_SyncTransmit function, the error code SPI_E_SEQ_PENDING is reported.

If the, Spi_SyncTransmit function is called with the parameter sequence using the same HwUnit while transmitting with the Spi_AsyncTransmit function, the error code SPI_E_SEQ_IN_PROCESS is reported.

In the Spi_Init function is called with an invalid driver configuration set parameter the error code SPI_E_PARAM_CONFIG is reported.

When an interrupt from an unconfigured SCB or DMA is detected, SPI's ISR reports SPI_E_PARAM_CONFIG.

The vendor-specific, Spi_Terminate function reports SPI_E_UNINIT if the driver is not in initialized state and reports SPI_E_PARAM_SEQ in case of an invalid sequence parameter.

The vendor-specific Spi_ChangeOvsSetting function reports:

- SPI_E_UNINIT if the driver is not in initialized state
- SPI_E_PARAM_OTHER in case of an invalid over sampling parameter (ScbOvsValue)
- SPI_E_PARAM_UNIT in case of an invalid external device id (ExtDev)

5.7 Production errors

If receive FIFO overflow is detected during asynchronous transfer (as used in levels 1 and 2), or if timeout error is detected during synchronous transfer, or executed Spi_Terminate API during asynchronous transfer (as used in levels 1), SPI_E_HARDWARE_ERROR is reported to the DEM - provided that its usage is enabled in the configuration.

For synchronous transmission timeout detection is implemented as a loop cycle counter with constant counter values. Transmission timeout counter is restarted after each channel data word that was successfully transmitted. You must make sure that the expected transmission duration and chip select durations are within timeout limits.

5.8 Reentrancy

All services except Spi_Init, Spi_DeInit, Spi_SetAsyncMode and Spi_MainFunction_Handling are reentrant.

5.9 Sleep mode

The SPI handler/driver and the hardware controlled by the SPI handler/driver do not provide a dedicated Sleep mode.

Note: All SPI sequences must be completed or stopped before entering the DeepSleep mode. SPI operation in DeepSleep mode is not guaranteed.

5.10 Debugging support

The SPI handler/driver does not support debugging.

5.11 Execution time dependencies

The execution of the API function is dependent on certain factors. Table 2 lists these dependencies.

Table 2 Execution time dependencies

Affected function	Dependency
`Spi_Init()`	Runtime depends on the number of configured hardware units, Jobs, sequences, and channels.
`Spi_DeInit()` `Spi_MainFunction_Handling()`	Runtime depends on the number of configured hardware units.
`Spi_AsyncTransmit()`	Runtime depends on the number of Jobs configured for the requested sequence and the total number of configured channels.
`Spi_SyncTransmit()`	Runtime depends on the number of Jobs configured for the requested sequence.

5.12 Deviation from AUTOSAR

By AUTOSAR standard, level 2 functionality will allow only one dedicated hardware instance for synchronous transmission. All other instances may be used for asynchronous transmission. The operation of synchronous and asynchronous transmission on the same hardware instance is not specified.

This SPI handler/driver allows synchronous transmission on multiple hardware instances (i.e., SCB units). Furthermore, it is possible to operate synchronous and asynchronous transmissions on the same hardware instance, provided they do not overlap in time.

5.13 Caveats

This section provides a non-exhaustive list of items that are responsible for your application:

- [SWS_Spi_00052] [SWS_SPI_00053] [SWS_SPI_00049] [SWS_SPI_00084]: The application will take care of the consistency of data in the external buffers and internal buffers during transmission. The application will ensure that any SPI channel is not used by more than one hardware channel at a time. The application will not call `Spi_SetupEB`, `Spi_WriteIB`, or `Spi_ReadIB` for channels that are currently in transmission.
- [SWS_SPI_00037]: The SPI handler/driver's environment will call the `Spi_SetupEB` function once for each SPI channel with EB declared before the SPI handler/driver's environment calls a transmit method on them.
- [SWS_SPI_00173]: The SPI handler/driver's environment will call the `Spi_AsyncTransmit` function after a function call of `Spi_SetupEB` for EB channels or a function call of `Spi_WriteIB` for IB channels but before the function call `Spi_ReadIB`.

- [SWS_SPI_00027]: The SPI handler/driver's environment will call the `Spi_ReadIB` function after a transmit method call to have relevant data within IB channel.
- [SWS_SPI_00257]: The SPI handler/driver's environment will not call `Spi_WriteIB` or `Spi_ReadIB` for channels that are currently in transmission because the SPI driver cannot prevent overwriting of the IB channel buffer.
- [SWS_SPI_00038] [SWS_SPI_00042] [SWS_SPI_00287]: The SPI handler/driver's environment will call the function to inquire the job status or the sequence status or the SPI hardware status (that is, `Spi_GetJobResult`, `Spi_GetSequenceResult`, or `Spi_GetHWUnitStatus`).

Your application must prevent synchronous and asynchronous transmissions on the same SCB from running concurrent transmission (asynchronous/synchronous or synchronous/asynchronous) when it transmits synchronously. This includes the case when a sequence is cancelled and one job is still in transmission. The transmission end can be checked by a sequence end notification or `Spi_GetHWUnitStatus`.

DMA usage for configured SCB, the corresponding TX, RX, or both interrupt service routines (ISRs) might not be generated. In such cases, the unused interrupt channels must be disabled at the interrupt controller (OS configuration); that is, they must not be mapped to an unhandled interrupt ISR.

Asynchronous mode (`SPI_POLLING_MODE`/`SPI_INTERRUPT_MODE`) must not be changed during the execution of `Spi_MainFunction_Handling`, that is. `Spi_SetAsyncMode` and `Spi_MainFunction_Handling` must not be called concurrently.

`Spi_MainFunction_Handling` must not interrupt or pre-empt other SPI handler/driver functions (interruption/pre-emption of the `Spi_MainFunction_Handling` by other SPI handler/driver functions is permitted according to their corresponding permitted reentrancy). `Spi_MainFunction_Handling` will be called from the lowest-priority task with reference to all other tasks and interrupts that call other SPI handler/driver functions.

The `Spi_SyncTransmit` function and the `Spi_AsyncTransmit` function cannot be operated at the same time using the same `SpiHwUnit`.

5.14 Functions available without core dependency

Some APIs can be called on any core regardless of resource assignment.

The following function is available on any core without any restriction:

- `Spi_GetVersionInfo(), Spi_GetStatus()`

The following functions are available on any cores with a specific section allocation described in the Note:

- `Spi_GetHWUnitStatus()`
- `Spi_GetJobResult()`
- `Spi_GetSequenceResult()`
- `Spi_GetBufferStatus()`

Note: The section `VAR_[INIT_POLICY]_ASIL_B_GLOBAL_[ALIGNMENT]` must be allocated to the memory. This can be read from any core to call these APIs on any cores.
For the details of `INIT_POLICY` and `ALIGNMENT`, see the Specification of memory mapping [5].

6 Hardware resources

6.1 Ports and pins

The SPI handler/driver uses the SCB instances of the TRAVEO™ T2G family microcontrollers. The pins listed in Table 3 are used. Make sure that the pins are correctly set in the PORT driver's configuration.

Table 3 Pins for SPI operation

Pin name	Direction	Drive mode	Description
SCB<n>_MISO	Input	high-Z	SCB channel <*n*> serial data input pin
SCB<n>_MOSI	Output	strong pull down | strong pull up	SCB channel <*n*> serial data output pin
SCB<n>_CLK	Output	strong pull down | strong pull up	SCB channel <*n*> clock I/O pin
SCB<n>_SELECT<m>	Output	strong pull down | strong pull up	Serial chip select <*m*> I/O pin of SCB channel

6.2 Timer

The SPI handler/driver does not use any hardware timers.

6.3 Interrupts

The interrupt services listed in Table 4 must be configured correctly for peripherals used by the SPI handler/driver. If a peripheral is not used, the corresponding interrupt service must not be present in the configuration.

Table 4 IRQ vectors and ISR names

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

Note: The OS must be associated with the named ISRs with the corresponding SCB interrupt. For example, if the hardware unit SCB ch.2 is configured, `Spi_Interrupt_SCB2_Cat2()` *must be called from the (OS-)interrupt service routine of SCB ch.2 interrupt. In case of category1 usage, the address of* `Spi_Interrupt_SCB2_ Cat1()` *must be the entry for SCB ch.2 interrupt in the (OS) interrupt vector table.*

Note: DMA completion ISRs are only generated if the given DMA channel is used by an SCB instance for SPI transmission. If there is an SCB channel that uses DMA, the interrupt handlers for SCB is required.

Table 5 Interrupt handler registration

Interrupt handler registration	Used DMA	Unused DMA
	DMA completion interrupt request ch.<*i*> for TX DMA completion interrupt request ch.<*j*> for RX SCB<*n*> interrupt request	- SCB<*n*> interrupt request

Note:

1. *Nesting interrupts are not supported because they may cause unexpected behavior. Therefore, all interrupts of the same SCB (including DMA channels) must be set to the same interrupt priority to avoid nesting interrupts itself and if you are using different* `HwUnits`*, it is possible to set different interrupt levels for each* `HwUnit`*.*
2. *The same interrupt priority will not nest itself. However, it allows nesting of other interrupts.*

Note: On the Arm® Cortex®-M4 CPU, priority inversion of interrupts may occur under specific timing conditions in the integrated system with TRAVEO™ T2G MCAL. 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.

TRAVEO™ T2G MCAL 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 SPI handler/driver uses DMA channels, which can be configured by the user and will be enabled/disabled by the SPI handler/driver as required. The DMA hardware itself must be enabled globally by the user before the SPI handler/driver can be used for DMA transfer.

When using DMA, ensure that one to one trigger multiplexer is correctly set in the PORT driver's configuration.

7 Appendix A – API reference

7.1 Include files

The *Spi.h* file is the only file that needs to be included to use functions from the SPI handler/driver.

7.2 Data types

7.2.1 Spi_StatusType

Type

```
typedef enum
{
  SPI_UNINIT,
  SPI_IDLE,
  SPI_BUSY
} Spi_StatusType;
```

Description

`Spi_StatusType` defines the range of specific status for the SPI handler/driver. This datatype holds the SPI handler/driver status and can be obtained by calling the API service `Spi_GetStatus`.

7.2.2 Spi_JobResultType

Type

```
typedef enum
{
  SPI_JOB_OK,
  SPI_JOB_PENDING,
  SPI_JOB_FAILED,
  SPI_JOB_QUEUED
} Spi_JobResultType;
```

Description

`Spi_JobResultType` defines the range of a specific job's status for the SPI handler/driver. This datatype holds the SPI handler/driver Job status and can be obtained by calling the API service `Spi_GetJobResult` with the job ID.

7.2.3 Spi_SeqResultType

Type

```
typedef enum
{
  SPI_SEQ_OK,
  SPI_SEQ_PENDING,
  SPI_SEQ_FAILED,
  SPI_SEQ_CANCELED
} Spi_SeqResultType;
```

Description

`Spi_SeqResultType` defines the range of a specific sequence status for the SPI handler/driver. This datatype holds the SPI handler/driver sequence status and can be obtained by calling the API service `Spi_GetSequenceResult` with the sequence ID.

7.2.4 Spi_DataBufferType

Type

`uint8`

Description

`Spi_DataBufferType` defines the type of application data buffer elements.

7.2.5 Spi_NumberOfDataType

Type

`uint16`

Description

`Spi_NumberOfDataType` defines the number of data elements of the `Spi_DataType` type used to send or receive on a channel.

7.2.6 Spi_ChannelType

Type

`uint8`

Description

`Spi_ChannelType` specifies the identification (ID) for a channel.

The type is numbered from 0 – <number of Channels-1>.

7.2.7 Spi_JobType

Type

`uint16`

Description

The `Spi_JobType` specifies the identification (ID) for Job. The type is numbered from 0 – <number of Jobs -1>.

7.2.8 Spi_SequenceType

Type

`uint8`

Description

The `Spi_SequenceType` specifies the identification (ID) for a sequence of Jobs. The type is numbered from 0 – <number of Sequences -1>.

7.2.9 Spi_HWUnitType

Type

`uint8`

Description

The `Spi_HWUnitType` specifies the identification (ID) for a SPI hardware peripheral unit.

7.2.10 Spi_AsyncModeType

Type

```
typedef enum
{
  SPI_POLLING_MODE,
  SPI_INTERRUPT_MODE
} Spi_AsyncModeType;
```

Description

`Spi_AsyncModeType` specifies the asynchronous mechanism mode for SPI busses handled asynchronously in level 2.

The type consists of the values `SPI_POLLING_MODE` and `SPI_INTERRUPT_MODE`.

7.2.11 Spi_ExtDeviceType

Type

`uint8`

Description

`Spi_ExtDeviceType` specifies the identification (ID) for a SPI external device.

7.2.12 Spi_OvsValueType

Type

`uint8`

Description

`Spi_OvsValueType` specifies the serial interface bit period oversampling factor.

7.3 Constants

7.3.1 Error codes

A service may return one of the error codes, listed in Table 6, if default error detection is enabled.

Table 6 Error codes

Name	Value	Description
SPI_E_PARAM_CHANNEL	10	Channel is not configured
SPI_E_PARAM_JOB	11	Job is not configured
SPI_E_PARAM_SEQ	12	Sequence is not configured
SPI_E_PARAM_LENGTH	13	Length is out of range
SPI_E_PARAM_UNIT	14	Hardware unit is out of range
SPI_E_PARAM_POINTER	16	versioninfo is NULL pointer
SPI_E_UNINIT	26	No Spi_Init done
SPI_E_SEQ_PENDING	42	Sequence is pending or shared job in pending sequence
SPI_E_SEQ_IN_PROCESS	58	Sequence is on transmission and SpiSupportConcurrentSyncTransmit is disabled or another sequence is on transmission on the same bus
SPI_E_ALREADY_INITIALIZED	74	API Spi_Init service is called while the SPI handler/driver has already been initialized
SPI_E_INVALID_CORE	90	Function called with a parameter which does not belong to this core
SPI_E_DIFFERENT_CONFIG	91	Intended config initialization of this core does not match with the initialized config of other cores
SPI_E_INIT_FAILED	92	Spi_Init service was failed

7.3.2 Vendor-specific error codes

Besides the error codes given in Error codes, this SPI handler/driver defines the errors listed in Table 7.

Table 7 Vendor-specific error codes

Name	Value	Description
SPI_E_INVALID_HW	82	The transmit API function is called for a sequence containing Jobs for an invalid hardware unit.
SPI_E_HW_ERROR	83	A hardware error occurred during transmission.
SPI_E_PARAM_BAD_MODE	84	Bad value for parameter mode supported.
SPI_E_BUSY	85	The specified channel is busy
SPI_E_PARAM_OTHER	86	Bad value for the other parameter supported.
SPI_E_PARAM_CONFIG	87	Incorrect value for the pointer of the configuration.

7.3.3 Version information

Table 8 Version information

Name	Value	Description
SPI_SW_MAJOR_VERSION	see release notes	Vendor-specific major version number
SPI_SW_MINOR_VERSION	see release notes	Vendor-specific minor version number
SPI_SW_PATCH_VERSION	see release notes	Vendor-specific patch version number

7.3.4 Module information

Table 9 Module information

Name	Value	Description
SPI_MODULE_ID	83	Module ID (Spi)
SPI_VENDOR_ID	66	Vendor ID

7.3.5 API service IDs

Table 10 lists the API service IDs used when reporting errors via DET or via the error callout function.

Table 10 API service IDs

Name	Value	API name
SPI_API_INIT	0x0	Spi_Init
SPI_API_DEINIT	0x1	Spi_DeInit
SPI_API_WRITEIB	0x2	Spi_WriteIB
SPI_API_ASYNCTRANSMIT	0x3	Spi_AsyncTransmit
SPI_API_READIB	0x4	Spi_ReadIB
SPI_API_SETUPEB	0x5	Spi_SetupEB
SPI_API_GETSTATUS	0x6	Spi_GetStatus
SPI_API_GETJOBRESULT	0x7	Spi_GetJobResult
SPI_API_GETSEQUENCERESULT	0x8	Spi_GetSequenceResult
SPI_API_GETVERSIONINFO	0x9	Spi_GetVersionInfo
SPI_API_SYNCTRANSMIT	0xA	Spi_SyncTransmit
SPI_API_GETHWUNITSTATUS	0xB	Spi_GetHWUnitStatus
SPI_API_CANCEL	0xC	Spi_Cancel
SPI_API_SETASYNCMODE	0xD	Spi_SetAsyncMode
SPI_API_MAINFUNCTION_HANDLING	0x10	Spi_MainFunction_Handling

7.3.6 Vendor-specific API service IDs

The following API service IDs are used when reporting errors via the error callout function:

Table 11 Vendor-specific API service IDs

Name	Value	Description
SPI_API_ISR	0x40	This API ID is used to indicate that an error occurred in a function that was called within an interrupt context.
SPI_API_GETBUFFERSTATUS	0x41	This is vendor-specific API ID for Spi_GetBufferStatus
SPI_API_HANDLER	0x42	This API ID is used to indicate that the hardware error occurred in an internal function.
SPI_API_TERMINATE	0x43	This is vendor-specific API ID for Spi_Terminate.
SPI_API_CHANGEOVSSETTING	0x44	This is vendor-specific API ID for. Spi_ChangeOvsSetting

7.3.7 Invalid core ID value

Table 12 Invalid core ID

Name	Value	Description
SPI_INVALID_CORE	0xFF	Invalid core ID

7.4 Functions

7.4.1 Spi_Init

Syntax

```
void Spi_Init(
  const Spi_ConfigType* ConfigPtr
)
```

Service ID

0x0

Sync/Async

Sync

Reentrancy

Non-reentrant

Parameters (in)

- ConfigPtr – Specifies the pointer to a configuration. If NULL pointer is specified, the first element of the configuration set array is used.

Parameters (out)

None

Return value

None

DET errors

- `SPI_E_ALREADY_INITIALIZED` – The SPI handler/driver has already been initialized.
- `SPI_E_PARAM_CONFIG` – The invalid pointer is specified.
- `SPI_E_INVALID_CORE` – The current core is not assigned.
- `SPI_E_INIT_FAILED` – `Spi_Init` services failed.
- `SPI_E_DIFFERENT_CONFIG` – Intended config initialization of this core does not match with the initialized config of other cores.

DEM errors

None

Description

This function initializes all local data for the configured channels, Jobs, and sequences. After initialization, the driver state will be SPI_IDLE, all sequence results will be SPI_SEQ_OK, and all Job results will be SPI_JOB_OK. This function will be called with NULL pointer. Only precompiled configuration parameters are used for initialization.

7.4.2 Spi_DeInit

Syntax

```
Std_ReturnType Spi_DeInit(
  void
)
```

Service ID

0x1

Sync/Async

Sync

Reentrancy

Non-reentrant

Parameters (in)

None

Parameters (out)

None

Return value

E_OK or E_NOT_OK

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.

- SPI_E_INVALID_CORE – The current core is not assigned.
- SPI_E_BUSY – The specified channel is busy.

DEM errors

None

Description

This function sets the driver state to SPI_UNINIT and returns E_OK.

Spi_DeInit returns E_NOT_OK, if the driver is in the SPI_BUSY state or in the SPI_UNINIT state.

7.4.3 Spi_WriteIB

Syntax

```
Std_ReturnType Spi_WriteIB(
  Spi_ChannelType Channel,
  const Spi_DataBufferType* DataBufferPtr
)
```

Service ID

0x2

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- Channel – Specifies the ID of the channel where data will be written.
- DataBufferPtr – Specifies the pointer to a data buffer containing data to be written. If DataBufferPtr is NULL, the default transmit value will be transmitted.

Parameters (out)

None

Return value

E_OK or E_NOT_OK

DET errors

- SPI_E_UNINIT – The driver is uninitialized.
- SPI_E_PARAM_CHANNEL – Undefined channel or incorrect channel type.
- SPI_E_INVALID_CORE – The current core and the resource assigned core are different. The current core is not assigned.

DEM errors

None

Description

This service writes data to the internal buffer associated with the parameter channel. You must ensure that the buffer given by `DataBufferPtr` has the same size as the internal buffer. If successful, it returns `E_OK`.

7.4.4 Spi_AsyncTransmit

Syntax

```
Std_ReturnType Spi_AsyncTransmit(
  Spi_SequenceType Sequence
)
```

Service ID

0x3

Sync/Async

Async

Reentrancy

Reentrant

Parameters (in)

- `Sequence` – Specifies the ID of the sequence that is to be transmitted.

Parameters (out)

None

Return value

E_OK or E_NOT_OK

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.
- `SPI_E_PARAM_SEQ` – Undefined sequence
- `SPI_E_SEQ_PENDING` – Sequence is pending or shares a job with a pending sequence or the sequence is included in the job of the same hardware unit as the synchronous transferring hardware unit.
- `SPI_E_INVALID_HW` – Sequence contains the jobs for an invalid hardware unit.
- `SPI_E_INVALID_CORE` – The current core and the resource assigned core are different. The current core is not assigned.

DEM errors

- `SPI_E_HARDWARE_ERROR` – Hardware error was detected. The error is reported after the job ends in the context of an interrupt or the main function.

Description

This function is the asynchronous service to transmit data on the SPI bus. This service takes the given parameter, initiates a transmission, sets the SPI handler/driver status to SPI_BUSY, sets the sequence result to SPI_SEQ_PENDING, sets all Jobs result to SPI_JOB_QUEUED, and returns. If a sequence requested by this hardware is pending, then the new sequence will be added to the transmit queue for this hardware unit;

otherwise, it will start immediately and set the first job result to `SPI_JOB_PENDING`. Note that you cannot call this function if a transmission is in progress on this channel. If successful, it returns `E_OK`.

7.4.5 Spi_ReadIB

Syntax

```
Std_ReturnType Spi_ReadIB(
  Spi_ChannelType Channel,
  Spi_DataBufferType* DataBufferPointer
)
```

Service ID

0x4

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `Channel` – Specifies the ID of the channel from which data will be read.
- `DataBufferPointer` – Specifies the pointer to a data buffer where the read data will be written.

Parameters (out)

None

Return value

E_OK or E_NOT_OK

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.
- `SPI_E_PARAM_CHANNEL` – Undefined channel or incorrect channel type
- `SPI_E_PARAM_POINTER` – Argument `DataBufferPointer` is NULL pointer
- `SPI_E_INVALID_CORE` – The current core and the resource assigned core are different. The current core is not assigned.

DEM errors

None

Description

This function reads data from the internal buffer specified by the parameter channel and writes this data to the area given by the `DataBufferPointer`. You must make sure that at least one transmission function has been called before attempting to read the buffer. You must also ensure that the area given by the `DataBufferPointer` is large enough to store the data from the internal buffer. Note that you must not call this function if a transmission is in progress on this channel. If successful, it returns E_OK.

7.4.6 Spi_SetupEB

Syntax

```
Std_ReturnType Spi_SetupEB(
  Spi_ChannelType Channel,
  const Spi_DataBufferType* SrcDataBufferPtr,
  Spi_DataBufferType* DesDataBufferPtr,
  Spi_NumberOfDataType  Length
)
```

Service ID

0x5

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `Channel` – Specifies the ID of the channel for which buffers are to be initialized
- `SrcDataBufferPtr` – Pointer to a data buffer that holds the transmit data
- `DesDataBufferPtr` – Pointer to a data buffer where incoming data is stored
- `Length` – Length of data to be transmitted/received; minimum length is 1 and the maximum length is set in configuration.

Parameters (out)

None

Return value

E_OK or E_NOT_OK

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.
- `SPI_E_PARAM_CHANNEL` – Undefined channel or incorrect channel type
- `SPI_E_PARAM_LENGTH` – Length is out of range or does not match to data width
- `SPI_E_PARAM_POINTER` – At least one of the data buffers is not aligned according to the buffer alignment required by the configuration.
- `SPI_E_INVALID_CORE` – The current core and the resource assigned core are different. The current core is not assigned.

DEM errors

None

Description

This function sets up the buffers and the length of data for the external buffers (EB) of the SPI handler/driver for the given channel. This function should be called for each channel that is configured with external buffers before a transmission is attempted. If `SrcDataBufferPtr` is NULL, the default data configured will be

transmitted. If `DesDataBufferPtr` is NULL, the incoming data is ignored by the driver. Note that you cannot call this function if a transmission is in progress on this channel. If successful, it returns `E_OK`.

7.4.7 Spi_GetStatus

Syntax

```
Spi_StatusType Spi_GetStatus(
  void
)
```

Service ID

0x6

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

None

Parameters (out)

None

Return value

`SPI_UNINIT`, `SPI_IDLE`, or `SPI_BUSY`

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.
- `SPI_E_INVALID_CORE` – The current core is not assigned.

DEM errors

None

Description

The function returns the SPI handler/driver status. It returns `SPI_UNINIT` if `Spi_Init` has not yet been called. It returns `SPI_IDLE` if there is no sequence in progress. It returns `SPI_BUSY` if at least one sequence is in progress.

7.4.8 Spi_GetJobResult

Syntax

```
Spi_JobResultType Spi_GetJobResult(
  Spi_JobType Job
)
```

Service ID

0x7

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `Job` – ID of the Job.

Parameters (out)

None

Return value

`SPI_JOB_OK`, `SPI_JOB_PENDING`, `SPI_JOB_FAILED`, or `SPI_JOB_QUEUED`

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.
- `SPI_E_PARAM_JOB` – Undefined Job ID

DEM errors

None

Description

The function returns the last transmission result of the specified job. If the SPI handler/driver has not been initialized when this service is called, the return value is undefined. The function is used to verify if the Job transmission succeeded (`SPI_JOB_OK`), failed (`SPI_JOB_FAILED`), executing (`SPI_JOB_PENDING`), or queued (`SPI_JOB_QUEUED`).

7.4.9 Spi_GetSequenceResult

Syntax

```
Spi_SeqResultType Spi_GetSequenceResult(
  Spi_SequenceType Sequence
)
```

Service ID

0x8

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `Sequence` – ID of the sequence.

Parameters (out)

None

Return value

`SPI_SEQ_OK`, `SPI_SEQ_PENDING`, `SPI_SEQ_FAILED`, or `SPI_SEQ_CANCELED`

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.
- `SPI_E_PARAM_SEQ` – Undefined sequence ID.

DEM errors

None

Description

The function returns the last transmission result of the specified sequence. This function is used to verify whether the full sequence transmission succeeded (`SPI_SEQ_OK`), failed (`SPI_SEQ_FAILED`), executing (`SPI_SEQ_PENDING`), or canceled (`SPI_SEQ_CANCELED`). If the service is called before the SPI handler/driver is initialized, the return value will be undefined.

7.4.10 Spi_GetVersionInfo

Syntax

```
void Spi_GetVersionInfo(
  Std_VersionInfoType* versioninfo
)
```

Service ID

0x9

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

- SPI_E_PARAM_POINTER – versioninfo 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.

7.4.11 Spi_SyncTransmit

Syntax

```
Std_ReturnType Spi_SyncTransmit(
  Spi_SequenceType Sequence
)
```

Service ID

0xA

Sync/Async

Async

Reentrancy

Reentrant

Parameters (in)

- `Sequence` – ID of the sequence.

Parameters (out)

None

Return value

E_OK or E_NOT_OK

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.
- `SPI_E_PARAM_SEQ` – Undefined sequence ID
- `SPI_E_SEQ_IN_PROCESS` – The function is called at the wrong time or the sequence is included in the job of the same hardware unit as the asynchronous transferring hardware unit.
- `SPI_E_INVALID_HW` – Sequence contains the jobs for an invalid hardware unit.
- `SPI_E_SEQ_PENDING` – Sequence is pending or shares a job with a pending sequence.
- `SPI_E_INVALID_CORE` – The current core and the resource assigned core are different. The current core is not assigned.

DEM errors

- `SPI_E_HARDWARE_ERROR` – Timeout error was detected.

Description

This function provides synchronous transmission of data. It sets the SPI handler/driver status to `SPI_BUSY`, sets the sequence status to `SPI_SEQ_PENDING`, sets the first Job status to `SPI_JOB_PENDING`, and performs the transmission. The driver accepts concurrent `Spi_SyncTransmit()` if the sequences to be transmitted use a different bus and `SpiSupportConcurrentSyncTransmit` is enabled. If successful, it returns `E_OK`. Job and sequence results are updated accordingly.

7.4.12 Spi_GetHWUnitStatus

Syntax

```
Spi_StatusType Spi_GetHWUnitStatus(
  Spi_HWUnitType HWUnit
)
```

Service ID

0xB

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `HWUnit` – ID of the hardware unit.

Parameters (out)

None

Return value

SPI_UNINIT, SPI_IDLE or SPI_BUSY

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.
- `SPI_E_PARAM_UNIT` – Undefined hardware unit

DEM errors

None

Description

This function returns the status of the specified SPI hardware unit.

7.4.13 Spi_Cancel

Syntax

```
void Spi_Cancel(
  Spi_SequenceType Sequence
)
```

Service ID

0xC

Sync/Async

Async

Reentrancy

Reentrant

Parameters (in)

- `Sequence` – ID of the sequence to be canceled.

Parameters (out)

None

Return value

None

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.
- `SPI_E_PARAM_SEQ` – Undefined sequence ID
- `SPI_E_INVALID_CORE` – The current core and the resource assigned core are different. The current core is not assigned.

DEM errors

None

Description

This function cancels an ongoing sequence transmission. The sequence will be canceled between jobs i.e., a Job will not be canceled once started. The sequence status will be set to `SPI_SEQ_CANCELED`.

7.4.14 Spi_SetAsyncMode

Syntax

```
Std_ReturnType Spi_SetAsyncMode(
  Spi_AsyncModeType Mode
)
```

Service ID

0xD

Sync/Async

Sync

Reentrancy

Non-reentrant

Parameters (in)

- `Mode` – The mode to be used for asynchronous transmissions.

Parameters (out)

None

Return value

E_OK or E_NOT_OK

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.
- `SPI_E_PARAM_BAD_MODE` – Value for mode is not supported.
- `SPI_E_INVALID_CORE` – The current core is not assigned.

DEM errors

None

Description

This function sets the mode for handling asynchronous transmissions on SPI buses. This may be interrupt mode (`SPI_INTERRUPT_MODE`) or polling mode (`SPI_POLLING_MODE`). `Spi_SetAsyncMode` must not be called during the execution of `Spi_MainFunction_Handling`.

7.4.15 Spi_GetBufferStatus

Syntax

```
Std_ReturnType Spi_GetBufferStatus(
  Spi_ChannelType Channel,
  const Spi_DataBufferType** SrcDataBufferPtrPtr,
  Spi_DataBufferType** DesDataBufferPtrPtr,
  Spi_NumberOfDataType* SrcRemainingLengthPtr,
  Spi_NumberOfDataType* DesRemainingLengthPtr
)
```

Service ID

0x41

Sync/Async

Sync

Reentrancy

Reentrant

Parameters (in)

- `Channel` – Channel ID.

Parameters (out)

- `SrcDataBufferPtrPtr` – The pointer that will be filled with the pointer to source data buffer
- `DesDataBufferPtrPtr` – The pointer that will be filled with the pointer to destination data buffer
- `SrcRemainingLengthPtr` – Pointer to the variable that will be filled with the remaining length (number of date elements) of the source data yet to be transmitted from the source data buffer
- `DesRemainingLengthPtr` – Pointer to the variable that will be filled with the remaining length (number of date elements) of the destination data yet to be received to destination data buffer

Return value

`E_OK`: Output parameters have been filled with the buffer status.
`E_NOT_OK`: Output parameters could not be filled with the buffer status.

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.
- `SPI_E_PARAM_CHANNEL` – Undefined channel
- `SPI_E_PARAM_POINTER` – NULL_PTR was passed as the parameters `SrcDataBufferPtrPtr`, `DesDataBufferPtrPtr`, `SrcRemainingLengthPtr`, or `DesRemainingLengthPtr`.

DEM errors

None

Description

Vendor-specific service to read back the buffer status and the remaining length of data for the SPI handler/driver channel specified.

After the transmission starts started (including the case that it has already finished), `Spi_GetBufferStatus` returns the buffer position and the remaining length calculated from the values that will be used (or have been used) for copying data.

`Spi_GetBufferStatus` returns the buffer pointers (`SrcDataBufferPtrPtr` and `DesDataBufferPtrPtr`) pointing to the position after the position in the buffer that was read/written the last time; that is, the pointer to the "next" position is returned or the pointer to the position directly after the buffer is returned if it was completely processed.

Depending on the configuration of the SCB, the update of the internal variables takes place in chunks or in a single block. Therefore, during transmission, the returned values may not reflect the actual pointer and remaining length. Instead, the returned values may relate to the buffer positions at an earlier point in time. The returned buffer positions and remaining lengths are determined before the transmission starts and after the transmission ends.

If channel TX data was set to NULL_PTR (i.e., default TX data) before transmission, then `Spi_GetBufferStatus` returns undetermined pointer in `SrcDataBufferPtrPtr` and undetermined length in `SrcRemainingLengthPtr` during and after transmission. The returned values cannot be used for TX plausibility checks.

If channel RX data was set to NULL_PTR (i.e., ignore RX data) before transmission, then `Spi_GetBufferStatus` returns undetermined `DesDataBufferPtrPtr` and undetermined length in `DesRemainingLengthPtr` during and after transmission. The returned values cannot be used for RX plausibility checks.

7.4.16 Spi_Terminate

Syntax

```
Std_ReturnType Spi_Terminate(
  Spi_SequenceType Sequence
)
```

Service ID

0x43

Sync/Async

Async

Reentrancy

Reentrant

Parameters (in)

- `Sequence` – Sequence ID of sequence to be terminated.

Parameters (out)

None

Return value

E_OK or E_NOT_OK

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.
- `SPI_E_PARAM_SEQ` – Undefined sequence ID.
- `SPI_E_INVALID_CORE` – The current core and the resource assigned core are different. The current core is not assigned.

DEM errors

None

Description

Vendor-specific service to terminate transmission on the SPI bus only for the ongoing sequence. If successful, it returns `E_OK`. SPI hardware unit status is updated accordingly.

7.4.17 Spi_ChangeOvsSetting

Syntax

```
Std_ReturnType Spi_ChangeOvsSetting(
  Spi_ExtDeviceType ExtDev,

  Spi_OvsValueType ScbOvsValue
)
```

Service ID

0x44

Sync/Async

Async

Reentrancy

Non-reentrant

Parameters (in)

- `ExtDev` – External device ID of external device that to be changed baud rate.
- `ScbOvsValue` – Setting value of OVS bit in SCB CTRL register.

Parameters (out)

None

Return value

E_OK or E_NOT_OK

DET errors

- `SPI_E_UNINIT` – The driver is uninitialized.
- `SPI_E_PARAM_UNIT` – Undefined external device ID.
- `SPI_E_PARAM_OTHER` – Invalid OVS value.
- `SPI_E_INVALID_CORE` – The current core and the resource assigned core are different. The current core is not assigned.

DEM errors

None

Description

Vendor-specific service to change SPI over sampling setting for the changing clock. If successful, it returns `E_OK`. The set value is reflected at the next transfer.

7.5 Scheduled functions

7.5.1 Spi_MainFunction_Handling

Syntax

```
void Spi_MainFunction_Handling(
  void
)
```

Service ID

0x10

Sync/Async

Sync

Reentrancy

Non-reentrant

Parameters (in)

None

Parameters (out)

None

Return value

None

DET errors

None

DEM errors

- `SPI_E_HARDWARE_ERROR` – Hardware error was detected

Description

You must call this function periodically when polling mode is used in the level 2 driver.

7.6 Required callback functions

7.6.1 SPI notification functions

The SPI handler/driver uses the following callback routines to inform other software modules about certain states or state changes. These other modules are required to provide the routines in the expected manner.

Callback notifications are statically configurable.

Implementation of all notification functions is required to be reentrant.

Notification functions are called if it is enabled in configuration, regardless of synchronous or asynchronous transmission.

The following API functions may be called from the SPI handler/driver callback notifications:

- `Spi_ReadIB`
- `Spi_WriteIB`
- `Spi_SetupEB`
- `Spi_GetJobResult`
- `Spi_GetSequenceResult`
- `Spi_GetHWUnitStatus`
- `Spi_Cancel`

All other SPI handler/driver API calls are not allowed.

7.6.1.1 Spi_JobEndNotification

Syntax

```
void (*Spi_JobEndNotification)(
     void
)
```

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

The `Spi_JobEndNotification` is a callback routine provided by the user for each job to notify the caller that a job has been finished. If configured, it will be called at the end of a job transmission.

7.6.1.2 Spi_SeqEndNotification

Syntax

```
void (*Spi_SeqEndNotification)(
     void
)
```

Parameters (in)

None

Parameters (out)

None

Return value

None

Description

The `Spi_SeqEndNotification` is a callback routine provided by the user for each sequence to notify the caller that a sequence has been finished. If configured, it will be called at the end of a sequence transmission.

7.6.2 DET

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

7.6.2.1 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` – SpiCoreConfigurationId of the core that calls this function or SPI_INVALID_CORE.
- `ApiId` – ID of the API service that calls this function
- `ErrorId` – ID of the detected development error

Return value

Returns always E_OK.

Description

Service for reporting development errors.

7.6.3 DEM

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

7.6.3.1 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 test result of the given event

Return value

None

Description

Service for reporting diagnostic events.

7.6.4 Callout functions

7.6.4.1 Error callout API

The AUTOSAR SPI module requires an error callout handler. Each 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 parameter `SpiErrorCalloutFunction`.

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` – `SpiCoreConfigurationId` of the core that calls this function or SPI_INVALID_CORE.
- `ApiId` – ID of the API service that calls this function
- `ErrorId` – ID of the detected error

Return value

None

Description

Service for reporting errors.

7.6.5 Callout functions

Get core ID API

The AUTOSAR SPI module requires a function to get valid core ID. This function is being used to determine the core from which the code is getting executed. The name of the function to be called can be configured by `SpiGetCoreIdFunction` parameter.

Syntax

```
uint8 GetCoreID_Function_Name (void)
```

Reentrancy

Reentrant

Parameters (in)

None

Return value

- `CoreId` – ID of the current core.

Description

Service for getting valid core ID.

Note: This function returns the core ID configured in `SpiMulticore/SpiCoreConfiguration/SpiCoreId`.
For example: Two cores are configured in the `SpiCoreConfiguration`.

Executing core	SpiCoreConfigurationId	SpiCoreId
CM7_0	0	15
CM7_1	1	16

- Upon calling this function from core CM7_0, it shall return 15.
- Upon calling this function from core CM7_1, it shall return 16.

8 Appendix B – Access register table

8.1 SCB

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
CTRL	31:0	Word (32 bits)	0x0100800F	Initialize CTRL register	Initialize SPI driver	0x9303970F	0x01000000
			0x81008000	De-initialize CTRL register	De-initialize SPI driver	0x9303D70F	0x81000000
			0x0100000 | SCB enable << 31 | over sampling value Depend on configuration	Set up CTRL register	From transfer start to transfer end	0x9303970F	0x01000000 bit[31]:Set on transfer stating/Clear on transfer ending bit[3:0]:Depend on baud rate of transfer
SPI_CTRL	31:0	Word (32 bits)	0x80000001	Initialize SPI_CTRL register	Initialize SPI driver	0x83014033	0x80000001
			0x03000010	De-initialize SPI_CTRL register	De-initialize SPI driver	0x8F017F3F	0x03000010
			0x80000001 | Chip select identifier << 26 | CS hold delay << 13 | CS set up delay << 12 | CS3 polarity << 11 | | CS2 polarity << 10 | CS1 polarity << 9 | CS0 polarity << 8 | Clock idle level << 3 | Data shift edge << 2 Depend on configuration	Set up SPI_CTRL register	When transfer start	0x83014033	0x80000001 bit[27:26]:Depend on chip select bit[13]:Depend on hold delay bit[12]:Depend on set up delay bit[11:8] :Depend on chip select polarity bit[3]:Depend on clock idle level bit[2]:Depend on data shift edge

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
SPI_TX_CTRL	31:0	Word (32 bits)	0x00000000	Initialize SPI_TX_CTRL register	Initialize SPI driver	0x00000030	0x00000000
			0x00000000	De-initialize SPI_TX_CTRL register	De-initialize SPI driver	0x00000030	0x00000000
			0x00000000	Refresh SPI_TX_CTRL register	When transfer start	0x00000030	0x00000000
SPI_RX_CTRL	31:0	Word (32 bits)	0x00000000	Initialize SPI_RX_CTRL register	Initialize SPI driver	0x00000130	0x00000000
			0x00000000	De-initialize SPI_RX_CTRL register	De-initialize SPI driver	0x00000130	0x00000000
			0x00000000	Refresh SPI_RX_CTRL register	When transfer start	0x00000100	0x00000000
TX_CTRL	31:0	Word (32 bits)	0x00000107	Initialize TX_CTRL register	Initialize SPI driver	0x00010000	0x00000000
			0x00000107	De-initialize TX_CTRL register	De-initialize SPI driver	0x0001011F	0x00000107
			0x00000000 | First transfer bit << 8 | Data width Depend on configuration	Set up TX_CTRL register	When transfer start	0x00010000	0x00000000 bit[8]:Depend on first transfer bit bit[4:0]:Depend on data width
TX_FIFO_CTRL	31:0	Word (32 bits)	0x00000000	Initialize SPI_TX_FIFO_CTRL register	Initialize SPI driver	0x00030000	0x00000000
			0x00000000	De-initialize SPI_TX_FIFO_CTRL register	De-initialize SPI driver	0x000300FF	0x00000000
			0x00000000 | invalidate FIFO << 16 | | FIFO trigger level Depend transfer mode	Set up transmitter FIFO control register	From transfer start to transfer end	0x00020000	0x00000000 bit[16]:Set on transmission starting/Clear on transmission ending

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
							bit[7:0]: Sync transfer: FIFO size/bytes per data element Async transfer (DMA): FIFO size/bytes per data element Async transfer (non-DMA interrupt): 1 Async transfer (non-DMA polling): FIFO size/bytes per data element
TX_FIFO_STATUS	31:0	Word (32 bits)	0x00000000	Read only register	Initialize SPI driver	0xFFFF81FF	0x00000000
			0x00000000	Read only register	De-initialize SPI driver	0xFFFF81FF	0x00000000
			0x00000000 | FIFO write pointer << 24 | FIFO read pointer << 16 | Amount of entries in FIFO Read only	Checking FIFO is not FULL.	During transfer	0x00008000	0x00000000
TX_FIFO_WR	31:0	Word (32 bits)	Transfer data	Transfer data	During transfer	-	- Write only register

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
RX_CTRL	31:0	Word (32 bits)	0x00000107	Initialize RX_CTRL register	Initialize SPI driver	0x00000200	0x00000000.
			0x00000107	De-initialize RX_CTRL register	De-initialize SPI driver	0x0000031F	0x00000107
			0x00000000 | First transfer bit << 8 | Data width Depend on configuration	Set up RX_CTRL register	During transfer	0x00000200	0x00000000. Bit[8]:Depend on first transfer bit bit[4:0]:Depend on data width
RX_FIFO_CTRL	31:0	Word (32 bits)	0x00000000	Initialize SPI_TX_FIFO_CTRL register	Initialize SPI driver	0x00030000	0x00000000
			0x00000000	De-initialize SPI_RX_FIFO_CTRL register	De-initialize SPI driver	0x000300FF	0x00000000
			0x00000000 | Invalidate FIFO << 16 | FIFO trigger level Depend transfer mode	Set up receiver FIFO control register	From transfer start to transfer end	0x00000200	0x00000000. Bit[16]:Set on receive starting/Clear on receive ending bit[7:0]: Sync transfer: FIFO size/bytes per data element Async transfer (DMA): 0 Async transfer (non-DMA interrupt): (FIFO size-24)/bytes per data element Async transfer (non-DMA polling): 0

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
RX_FIFO_STATUS	31:0	Word (32 bits)	0x00000000	Read only register	Initialize SPI driver	0xFFFF81FF	0x00000000
			0x00000000	Read only register	De-initialize SPI driver	0xFFFF81FF	0x00000000
			0x00000000 | FIFO write pointer << 24 | FIFO read pointer << 16 | Amount of entries in FIFO Read only	Checking received data exist.	During transfer	0x00008000	0x00000000
RX_FIFO_RD	31:0	Word (32 bits)	DATA[31:0]	Received data	-	-	- Can't monitoring
INTR_CAUSE	31:0	Word (32 bits)	0x00000000	Initialize	Initialize SPI driver	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
			0x00000000	De-initialize	De-initialize SPI driver		
			0x00000000 | RX interrupt << 3 | Master interrupt Read only	Interrupt cause	During transfer		
INTR_I2C_EC_MASK	31:0	Word (32 bits)	0x00000000	Initialize externally clocked I2C interrupt mask register	Initialize SPI driver	0x0000000F	0x00000000
			0x00000000	De-initialize externally clocked I2C interrupt mask register	De-initialize SPI driver	0x0000000F	0x00000000

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
INTR_SPI_EC_MASK	31:0	Word (32 bits)	0x00000000	Initialize externally clocked SPI interrupt mask register	Initialize SPI driver	0x0000000F	0x00000000
			0x00000000	De-initialize externally clocked SPI interrupt mask register	De-initialize SPI driver	0x0000000F	0x00000000
INTR_M	31:0	Word (32 bits)	0x000003FF	Initialize Master interrupt request register	Initialize SPI driver	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
			0x000003FF	De-initialize Master interrupt request register	De-initialize SPI driver		
			0x00000000 | SPI transfer done << 9	SPI bus idle checking	During transfer		
INTR_M_MASK	31:0	Word (32 bits)	0x00000000	Initialize Master interrupt mask register	Initialize SPI driver	0x00000317	0x00000000
			0x00000000	De-initialize Master interrupt mask register	De-initialize SPI driver	0x00000317	0x00000000
			0x00000000 | SPI transfer done interrupt mask	Enable or disable SPI_DONE interrupt	During transfer in interrupt mode	0x00000117	0x00000000 bit[9]:Set on complete TX data write to FIFO in non-DMA Async transfer Set on complete RX data receiving in DMA Async transfer

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
INTR_S_MASK	31:0	Word (32 bits)	0x00000000	Initialize Slave interrupt mask register	Initialize SPI driver	0x000007FF	0x00000000
			0x00000000	De-initialize Slave interrupt mask register	De-initialize SPI driver	0x000007FF	0x00000000
INTR_TX	31:0	Word (32 bits)	0x000007FF	Initialize transmitter interrupt request register	Initialize SPI driver	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
			0x000007FF	De-initialize transmitter interrupt request register	De-initialize SPI driver		
			0x000007FF	Clear all transmitter interrupt factor	When transition stop		
INTR_TX_MASK	31:0	Word (32 bits)	0x00000000	De-initialize transmitter interrupt mask register	Initialize SPI driver	0x00007FFF	0x00000000
			0x00000000	De-initialize transmitter interrupt mask register	De-initialize SPI driver	0x00007FFF	0x00000000
			0x00000000	Disable all transmitter interrupts	When transmission stop	0x00007FFF	0x00000000

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
INTR_RX	31:0	Word (32 bits)	0x00000FFF	Initialize receiver interrupt request register	Initialize SPI driver	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
			0x00000FFF	De-initialize receiver interrupt request register	De-initialize SPI driver		
			0x00000FFF	Clear all receiver interrupt factor	When receiving stop When receiver interrupt is cached		
			0x00000000 | FIFO over flow << 5	Checking transfer error	During transfer		
			0x00000000 | FIFO not empty << 2 | FIFO trigger	Checking received data exist.	During transfer		

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
INTR_RX_MASK	31:0	Word (32 bits)	0x00000000	Initialize receiver interrupt mask register	Initialize SPI driver	0x00000FFF	0x00000000
			0x00000000	De-initialize receiver interrupt mask register	De-initialize SPI driver	0x00000FFF	0x00000000
			0x0000000 | FIFO trigger interrupt enable	Enable receiver FIFO trigger interrupt	When transfer start without DMA in interrupt mode	0x00000F80	0x00000000 bit[0]: Set on Async transfer (non-DMA) starting/Clear on Async transfer (non-DMA) ending
			0x00000000	Disable all interrupts	When transfer start with DMA in interrupt mode or non-interrupt mode	0x00000FFF	0x00000000
			0x00000000	Disable all receiver interrupts	When receiving stop	0x00000FFF	0x00000000

8.2 DW

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
CH_CTL	31:0	Word (32 bits)	0x00000002	Initialize channel control register	Initialize SPI driver	0x80000BF4	0x00000000
			0x00000002	De-initialize channel control register	De-initialize SPI driver	0x80000BF4	0x00000000
			0x00000000 | DMA channel enable << 31	Start or Stop DMA	During transfer with DMA	0x00000BF4	0x00000000 bit[31]:Set on Async transfer (DMA) stating/Clear on Async transfer (DMA) ending
CH_STATUS	31:0	Word (32 bits)	-:Read only	Initialize channel status register	Initialize SPI driver	0x0000000F	0x00000001
			-:Read only	De-initialize channel status register	De-initialize SPI driver	0x0000000F	0x00000001
			Cause of interrupt Read only	Checking DW channel status.	During transfer with DMA	0x00000000	0x00000000 bit[3:0]:Clear on Async transfer (DMA) stating/ Set on Async transfer (DMA) ending
CH_IDX	31:0	Word (32 bits)	0x00000000	Initialize channel current indices	Initialize SPI driver	0x00000000	0x00000000
			0x00000000	De-initialize channel current indices	De-initialize SPI driver	0x0000FFFF	0x00000000
			0x00000000 | Y loop index << 8 | X loop index	Calculate buffer position	During transfer with DMA	0x00000000	0x00000000 bit[15:8] | bit[7:0] Clear on Async transfer (DMA) stating Change on during transfer

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
CH_CURR_PTR	31:0	Word (32 bits)	0x00000000	Initialize channel current descriptor pointer register	Initialize SPI driver	0x00000000	0x00000000
			0x00000000	De-initialize channel current descriptor pointer register	De-initialize SPI driver	0xFFFFFFFC	0x00000000
			ADDR[31:2]	Set descriptor address	When stating transfer with DMA	0x00000000	0x00000000 bit[31:2]:Set to current descriptor address on stating transfer
			ADDR[31:2]	Calculate buffer position	During transfer with DMA	0x00000000	0x00000000 bit[31:2]:Clear to 0 on ending transfer
INTR	31:0	Word (32 bits)	0x00000001	Initialize interrupt register	Initialize SPI driver	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
			0x00000001	De-initialize interrupt register	De-initialize SPI driver		
			0x00000001	Clear interrupt	When stating transfer with DMA When DMA interrupt is caught		
INTR_MASK	31:0	Word (32 bits)	0x00000000	Initialize interrupt mask register	Initialize SPI driver	0x00000001	0x00000000
			0x00000000	De-initialize interrupt mask register	De-initialize SPI driver	0x00000001	0x00000000
			0x00000000 | Enable interrupt	Disable or enable DMA interrupt	During transfer with DMA	0x00000000	0x00000000 bit[0]:Set on stating DMA/Clear on ending DMA
SRAM_DATA0	31:0	Word (32 bits)	0x00000000	Initialize SRAM data0 register	Initialize SPI driver	0x00000000	0x00000000 (monitoring is not needed.)

Register	Bit No.	Access size	Value	Description	Timing	Mask value	Monitoring value
						(monitoring is not needed.)	
			0x00000000	De-initialize SRAM data0 register	De-initialize SPI driver	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
SRAM_DATA1	31:0	Word (32 bits)	0x00000000	Initialize SRAM data1 register	Initialize SPI driver	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
			0x00000000	De-initialize SRAM data1 register	De-initialize SPI driver	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)

Revision history

Revision	Issue date	Description of change
**	2020-09-16	Initial release
*A	2020-11-20	2.6 Memory Mapping Changed Spi_MemMap.h file include folder. 2.6.2 Memory Allocation and Constraints Added the restriction of VRAM. 4.1 General Configuration Deleted restriction of SpiSupportConcurrentSyncTransmit. 4.2.3 External Device Configuration Changed and added Note description. SpiCsSelection SpiHwUnit 7.4.11 Spi_SyncTransmit Deleted restriction of SpiSupportConcurrentSyncTransmit. Migrated to Infineon template.
*B	2021-05-24	5.9 Sleep Mode Changed description and added Note. 5.1.1.3 Externally Buffered Channels Changed Note.
*C	2021-08-19	Added note in 6.3 Interrupts.
*D	2021-12-07	Updated to the latest branding guidelines.
*E	2023-10-06	Updated register information in 8.2. Corrected core identification keyword in sections 2.6 and 5.14.
*F	2023-12-08	Web release. No content updates.
*G	2024-03-18	4.4.4 BSW scheduler Changed BSW scheduler (SchM) section name 6.3 Interrupts Deleted Note
*H	2024-07-22	Updated description in 5.7 Production errors.

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

Edition 2024-07-22

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email:

erratum@infineon.com

Document reference

002-30203 Rev. *H

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.