Montana Firmware Reference

M-20875-004
October 2021

onsemi™

Table of Contents

Introduction

1.1 Purpose

IMPORTANT: onsemi acknowledges that this document might contain the inappropriate terms “white list", "master" and "slave”. We have a plan to work with other companies to identify an industry wide solution that can eradicate non-inclusive terminology but maintains the technical relationship of the original wording. Once new terminologies are agreed upon, future products will contain new terminology.

This group of topics describes the firmware included in the Montana System-on-Chip. It includes descriptions, function listings, and usage examples to help you understand the firmware and its parts.

The firmware described in this manual provides developers with a convenient software layer on which to build their applications. It is also responsible for system-level tasks such as booting the system and implementing portions of the security layer. The firmware consists of include files, macros, libraries, ROM code, and executables.

1.2 Intended Audience

This group of topics is intended for use by developers who are designing and implementing applications for the Montana System-on-Chip. Both novice and experienced developers can benefit. The descriptions and code examples can help novice users to learn the system, while experienced developers can go straight to the reference chapters that describe the available functions, macros, libraries and executables.

These topics assume that the reader has a basic understanding of:

- The architecture of the Montana chip
- The C programming language
- The fundamentals of the Arm® Thumb®-2 assembly language
- The integrated development environment and toolchains that form the development tools for the Montana SoC

1.3 Conventions

`monospace font`

Assembly code, macros, functions, defines and addresses.

italics

File and path names, or any portion of them.

`<angle brackets>`

Optional parameters and placeholders for specific information. To use an optional parameter or replace a placeholder, specify the information within the brackets; do not include the brackets themselves.

*

Wild card placeholder; typically used to indicate a place where two or more numeric constants could be used.

CHAPTER 2

Firmware Overview

Montana is supported by a set of firmware components that provide:

- A thin layer of support between the hardware and the developer. This firmware allows developers to focus on application development, reducing the number of details they need to know about the underlying Montana hardware.
- A support layer for common complex run time operations frequently included in user applications
- Security and lifetime provisioning elements, which help support the security elements needed throughout a device's lifecycle
- Wireless protocol support functionality for Bluetooth® Low Energy applications
- Manufacturing and debug support elements

2.1 Introduction

The system firmware provides hardware definitions and a hardware abstraction layer for common operations. These definitions and abstraction layer are easier to use and understand than low-level C or assembly code. The firmware components supporting common run time, debug, and manufacturing elements simplify incorporation of the device into a system throughout the development cycle. In addition, the security and wireless protocol firmware elements provide more advanced functionality, which can form the basis for applications developed to take advantage of the hardware that the Montana SoC provides.

When multiple programmers are involved in development, using the firmware leads to increased consistency, which in turn leads to increased overall robustness and correctness of code.

In some cases, depending on the particulars of the application, the firmware implementation might not be optimal; however, even in these situations, the firmware serves as an example and an advanced starting point for custom-developed functions and macros.

All firmware components execute on the Arm Cortex®-M33 processor, and all of these components are CMSIS-compatible.

2.1.1 Compliance Exceptions

The firmware provided for the Arm Cortex-M33 processor is generally compliant with the MISRA-C rules, as required by the CMSIS standard. The Montana firmware exceptions in compliance are the same compliance exceptions that are part of the CMSIS-Core standard.

The Montana firmware and CMSIS-Core violate the following MISRA-C rules:

- Required Rule 8.5, object/function definition in header file. Violated because function definitions in header files are used to allow inlining of functions.
- Required Rule 18.4, declaration of union type or object of union type: {...}. Violated because unions are used for effective representation of core registers.
- Advisory Rule 19.7, function-like macro defined. Violated because function-like macros are used to allow for more efficient code.

2.2 Firmware Components

The firmware files consist of include files (denoted with *.h* extensions) and precompiled library binaries (denoted with *.a* extensions). Some precompiled libraries are also provided in source code format. Descriptions of the firmware components, and detailed API references, are provided in the remainder of this group of topics.

Applications that use the libraries provided must:

1. Include the associated firmware include file.
2. Link against any dependencies of the desired library.
3. Link against a version of the desired library.

The Arm CryptoCell™-312 libraries are available in the following formats:

- Debug
- Release

The event kernel support firmware and the Bluetooth Low Energy protocol stack are available as a library.

2.3 Firmware Naming Conventions

For clarity and ease of use, many firmware components follow several naming conventions for library functions and macros. These conventions are compatible with the CMSIS naming requirements.

- Macros (defined using a `#define` statement) use all capitals in the macro name. These macro names include an all-capital prefix indicating the library or other firmware element they are supporting (e.g., `SYS_`). If the macro supports a specific target component, this prefix is followed by the name of the component it supports. The rest of the macro name indicates the intended functionality of the macro.
- Inline and standard firmware functions use camel-case function names (e.g., `CalcPhaseCnt`). All functions use a prefix to indicate which library provides the function (e.g., `Sys_`). The remainder of a function's name indicates the block it affects and the function's intended functionality.

Table 1 lists the prefixes for each of the firmware libraries that use prefixes.

Table 1. Library Function Naming Convention

Library	Macro Prefix	Function Prefix
Hardware Abstraction Layer, System Library	`SYS_`	`Sys_`
Flash Support Library	`FLASH_`	`Flash_`
Event Kernel	`KE_`	`Kernel_`, `ke_`
Supplemental Calibration Library	N/A	`Calibrate_`

Elements that follow other naming conventions include the following:

- The CMSIS library and drivers follow the CMSIS standard, which provides standard names for all CMSIS macros and functions.
- The Arm CryptoCell-312 libraries largely use an API defined by Arm for the Arm TrustZone® CryptoCell-312 IP.
- The Bluetooth Library uses naming and terminology from the Bluetooth Core Specification; for more information on the naming conventions for the Bluetooth Library, see the CEVA® documentation provided with your Montana install.
- The swmTrace library is a logging library that is paired with the Real-Time Transfer (RTT) viewer that is part of the onsemi IDE and uses the RTT interface defined by SEGGER®.

2.4 Firmware Resource Usage

The firmware uses the Arm Cortex-M33 processor system stack. It expects that the Arm Cortex-M33 processor's stack pointer points to a valid stack that grows downward (i.e., decreasing memory addresses).

Other system memory used by the system are listed in the "Reserved Resources" table (Table 2).

Table 2. Reserved Resources

Reservation	Addresses	Notes
NVR0 to NVR3	0x00080000 to 0x000803FF	CryptoCell-312 support records and tables
NVR7	0x00080700 to 0x000807FF	Trim records; accessible using the `TRIM_Type` structure from *montana_map.h*
MNVR	0x00080800 to 0x000808FF	Manufacturing and manufacturing trim records; cannot be modified outside of manufacturing.
FLASH_RSVD	0x00158000 to 0x00158BFF	Arm CryptoCell-312 DEU transfer space
FLASH_BOND_RSVD	0x00158C00 to 0x001593FF	Recommended location for Bluetooth Low Energy bond information

Other system components (DMA channels, LSAD inputs, etc.) that are used by the firmware are noted in the documentation for the specific component.

2.5 VERSIONS

Version symbols are provided for each major firmware component. The version symbols can be used directly or indirectly to verify the version of the components being used to build an application. There are three types of version symbols available:

Define
A preprocessor define or set of defines containing the version information.

Symbolic
A compiled symbol contained within a binary library

Global Variable
A global variable in memory containing the symbol

As an example, the available version information for the flash library firmware component is listed in the "Example Firmware Versions - Flash Library" table (Table 3).

Table 3. Example Firmware Versions - Flash Library

Type	Version Symbol	Description
Define	`FLASH_FW_VER_MAJOR`	Major component of the library version; updated for non-backward compatible changes
Define	`FLASH_FW_VER_MINOR`	Minor component of the library version; updated for backward compatible changes, reset if major version is incremented
Define	`FLASH_FW_VER_REVISION`	Revision for the library version; incremented for minor changes or bug-fixes that do not affect library use

Table 3. Example Firmware Versions - Flash Library (Continued)

Type	Version Symbol	Description
Define	`FLASH_FW_VER`	Combined library version (16 bits): • 15:12 major version • 11:8 minor version • 7:0 revision
Global Variable	`FlashLib_Version`	Constant variable assigned to hold the combined library version definition

CHAPTER 3

Hardware Definitions

Hardware definition files are integral to the system firmware. The hardware definitions apply a layer of data structures and address mappings to the underlying hardware, so that every control register, bit field in the system, memory, and interrupt vector is easily accessible from C and assembly code. This set of header files provides a system definition for the Montana SoC, including:

- Register and bit descriptions for registers accessible to the processor
- Memory maps for all of the memories and memory-mapped elements that are accessible to the processor
- Interrupt vector table descriptions for the processor
- Macros that support the Arm Cortex-M33 processor's basic core functionality

The format and configuration of all of these support files conform to CMSIS compatibility requirements wherever possible. As required by CMSIS, the hardware definitions are included by the top-level CMSIS include file (*montana.h*) alongside the other CMSIS requirements. For more information about CMSIS, and this top-level include file, see "Introduction" on page 57.

If an application includes the top-level header file, and defines `MONTANA_CID` to match the chip identifier of the relevant Montana device, then all of the support macros and HAL functions that are available to support that processor development on the chip are made accessible to that application.

NOTE: All devices that share a chip identifier are guaranteed to be compatible with the same firmware.

3.1 Register and Register Bit-Field Definitions

Using the hardware definition files allows you to refer to system components by C structures, assembly code, and preprocessor symbols instead of by addresses and bit fields. This greatly improves the readability, reliability and maintainability of your application code. The use of hardware definitions in an application also means that some hardware changes, such as changes to addresses or bit field values, are transparent to your application code.

Hardware register descriptions for the Arm Cortex-M33 processor's private peripherals are provided by the CMSIS package from Arm. Hardware register descriptions for all other components, and bit settings that are appropriate for use with the hardware register descriptions for the private peripherals, are available in the following files:

- *montana_hw.h*: This generic include file selects the desired underlying header file appropriate to your hardware by using the `MONTANA_CID` definition.
- *montana_hw_cid*.h*: This include file is the header file that is appropriate for all devices that are compatible with the defined `MONTANA_CID` (i.e., devices sharing the same chip version and major revision).
- *montana_hw_flat_cid*.h*: An unstructured version of the *montana_hw_cid*.h* header file, which includes all of the same definitions but no structure typedefs (useful for application elements written in assembly).

NOTE: For applications that are intended to operate in non-secure application modes, the *montana_hw_cid*_ns.h* headers are used, as long as `NON_SECURE` is defined by the preprocessor.

Hardware descriptions in the register include files provide definitions supporting the SoC components using the defines and C objects listed in the "Hardware Register Components" table (Table 4).

Table 4. Hardware Register Components

Item	Example	Description
Component Register Structure	`GPIO_Type`	Provides a list of all registers that support a specified component, and the read/write types for those registers
Component Register Instance	`GPIO`	Links the component register structure to the underlying hardware or sets of hardware
Bit-Field Positions	`GPIO_CFG_DRIVE_Pos`	Defines the base position for any bit-field within a register
Bit-Field Mask	`GPIO_CFG_DRIVE_Mask`	Defines a bit mask for any bit-field of more than one bit within a register
Register Structure	`GPIO_CFG_Type`	Provides a list of all sub-registers and alias structures • Sub-registers are defined byte (8-bit) or short (16-bit) access interfaces to part of a register that includes all elements belonging to the same configuration area. • Aliases are Arm Cortex-M33 processor bitband aliases that provide bit access to individual single-bit bit-fields where the underlying hardware supports this single-bit access.
Register Instance	`GPIO_CFG`	Links the register structure to the underlying hardware or sets of hardware for sub-registers
Bit-Setting	`GPIO_MODE_GPIO_IN_0`	Defines providing human-readable equivalents to settings that can be applied to a register bit-field to obtain the desired behavior
Bit-Field Sub-Register Positions	`GPIO_CFG_IO_MODE_BYTE_Pos`	Defines the base position for any bit-field within a register’s sub-register
Bit-Field Sub-Register Mask	`GPIO_CFG_IO_MODE_BYTE_Mask`	Defines a bit mask for any bit-field of more than one bit within a register’s sub-register
Sub-Register Bit-Setting	`GPIO_MODE_GPIO_IN_0_BYTE`	Defines providing human-readable equivalents to settings that can be applied to a register’s sub-register bit-field to obtain the desired behavior

3.2 Memory Map Definition

The *montana_map.h* header file contains the specific memory map definitions that provide the locations of the following structures within the memory maps of the processor:

- Instruction and data bus memory structures
- System bus memory structures
- Peripheral bus memory-mapped control registers (including the base of control register groups for each system component)
- Private peripheral bus internal memory-mapped control registers
- System variables
- Calibration trim records

For more information on the memory map and memory mapped elements accessible to the Arm Cortex-M33 processor, see the *Montana PTS*.

3.3 Interrupt Vector Definition

Interrupt vectors provide you with access to interrupts that facilitate orderly processing operations, for optimal application processing speed and minimal delays. Interrupt vector definitions are provided for the Arm Cortex-M33 processor in the *montana_vectors.h* header file.

Interrupt handling functionality in the Arm Cortex-M33 processor is provided by a nested vector interrupt controller (NVIC) implemented with the processor. The NVIC handles predefined interrupts internal to the core including a non-maskable interrupt (NMI), and interrupts external to the processor, that are linked to interfaces and peripherals. The NVIC is supported by standard firmware as part of the CMSIS library (as described in Chapter 5 "CMSIS Library" on page 57), and by the Hardware Abstraction Layer library (described in Chapter 4 "Hardware Abstraction Layer" on page 56), which offers additional supporting functions. These definitions have the form **`<interrupt_name>`**`_IRQn`.

For more information and a complete list of interrupt vectors, along with enumeration defines and values, see the Arm Cortex-M33 Processor chapter of the *Montana PTS*.

CHAPTER 4

Hardware Abstraction Layer

The Hardware Abstraction Layer (HAL) library provides a set of multiple register access functions for the majority of the system components included in the Montana SoC. These functions, and the included function headers, can easily be used to configure the hardware blocks through safe register accesses with no need to refer to register tables and corresponding settings in the Montana.

The HAL library is distributed in both source-code and pre-compiled form in the Montana SDK. All the library functions are blocking and interruptible. The library does not use any interrupts or DMA channels.

4.1 USAGE

To use the HAL library functions, the HAL library must be included via the project's **.rteconfig* file, under **Libraries**. Additionally, the *montana.h* header file must be included in the application source.

The HAL functions follow `Sys_`**`<hardware block>_<function>`** name syntax, with:

- **`<hardware block>`** indicating the system component being configured or acted upon
  - In some cases, the hardware block is prefixed by a standard operation such as `Set_` or `Get_`
- **`<function>`** describing the functionality implemented by the HAL firmware

Simple wrapper macros, with the same names but capitalized, are also provided for many functions. These macros do not require a user application to provide a pointer to the block being configured or operated upon, as the macros default to the first instance available. For example, `SYS_I2C_CONFIG` is a macro that calls the `Sys_I2C_Config` function with I2C0 for the pointer parameter.

For the complete HAL library API, refer to Chapter 17 "Hardware Abstraction Layer Reference" on page 138.

CHAPTER 5

CMSIS Library

The Arm Cortex-M33 processor is supported by a standards-compliant CMSIS library and extensions to the CMSIS requirements.

5.1 Introduction

The Arm Cortex-M33 processor is supported by a standards-compliant CMSIS library and extensions to the CMSIS requirements. The CMSIS library performs the following functions:

- Provides access to the generic functions provided by the standard Arm CMSIS implementation
    - The functions are included in the CMSIS header files, and reference documentation is provided by the standard Arm CMSIS documentation (http://arm-software.github.io/CMSIS_5/Core/html/modules.html).
- Implements the CMSIS-required device-specific functions with an appropriate implementation for Montana
- Adds extensions to the generic and required CMSIS functions to provide additional common support functions, including:
    - Weak definitions for all interrupt handlers for the Arm Cortex-M33 processor and associated NVIC
    - A C start-up routine for ensuring that applications reach main in a safe state
    - Dynamic memory allocation (through an implementation of `sbrk`)
    - Hardware-safe functions for updating the system clock source and operating frequency

The weakly defined interrupt handlers included in the CMSIS library provide empty implementation for the interrupt vectors, each of which simply returns from the interrupt. These interrupt handlers have the form **`<interrupt_name>`**`_IRQHandler()`. If a user application defines a function with this same name, the user application's definition of the interrupt handler replaces the default (empty) handlers.

The CMSIS library is supported by the top-level include file for Montana, *montana.h*, which includes all CMSIS, Hardware Definitions, and Hardware Abstraction Layer elements. To use the CMSIS library, include *montana.h* and link against the *libcmsis.a* library archive.

For the complete API, refer to Chapter 16 "CMSIS Reference" on page 118.

CHAPTER 6

CMSIS Drivers

The CMSIS drivers are generic device-independent APIs for peripherals, including I^2C, SPI, and UART. These drivers are created to allow easy setup and use of peripherals and interfaces by handling most of the manual configuration work in the back-end. The drivers can be used in the sample applications by including the appropriate interface header files and the CMSIS drivers library.

6.1 Introduction

This topic presents the description of all the available CMSIS drivers and the instructions on how to use the configuration file (*RTE_Device.h*). The specifications and headers for all of the functions used in these CMSIS drivers have been provided by Arm, and the corresponding documentation can be found on the Arm Keil website, on this page: http://www.keil.com/pack/doc/CMSIS/Driver/html/index.html.

This topic assumes that the user is familiar with importing projects into the IDE, and also with the location of the sample applications. For more information about these topics, see the *RSL15 Getting Started Guide*.

A list of interfaces with CMSIS driver support is shown in the "Interfaces for Which a CMSIS Driver is Available" table (Table 5).

Table 5. Interfaces for Which a CMSIS Driver is Available

CMSIS Driver	Interfaces Supported	Sample Application
DMA	DMA channels 0 to 3	dma
I2C	I2C0, I2C1	i2c_cmsis
SPI	SPI0, SPI1	spi_cmsis
USART	USART0	uart_cmsis

The drivers used for CMSIS pack sample applications are shown in the "Types of Drivers" table (Table 6):

Table 6. Types of Drivers

CMSIS Driver
SPI, I2C, USART

6.2 Library Structure

A drivers library is organized as follows:

- *drivers* - root folder
    - *code*: Interface and utility source files
    - *include*: Header files for the interfaces and the utility functions, and the specifications provided for the interfaces by Arm
    - *RTE_Device.h*: Run-Time Environment device configuration file for all available interfaces and instances. See Section 6.3 “Configuring The Driver Run-time Environment” on page 59 for configuration instructions.

6.3 Configuring The Driver Run-time Environment

The driver run-time environment can be configured using the *RTE_Device.h* file. A local instance of this file needs to be maintained as part of a drivers library instance in the local workspace. Every time this file is changed, the drivers library needs to be compiled again to include the latest configuration. The resulting *libdrivers.a* file must also be built into any application project that wants to use this new configuration. The *RTE_Device.h* file uses configuration wizard annotations, allowing the configuration to be completed in a GUI or text-based editor.

After configuring all the settings using either of these methods, build the library project to generate the required *.a* file to be included in the projects that use the CMSIS drivers. The CMSIS sample applications mentioned in Section 6.2 "Library Structure" can be imported for use as reference projects.

6.3.1 Using a GUI-Based Editor

An Eclipse plugin to edit the configuration wizard annotations is included in the SDK. To configure the CMSIS drivers using the editor with a graphical user interface:

1. Import the drivers library from the *source/Cortex-M33* folder in the installation directory, right-click on *RTE_ Device.h*, and choose to **Open With CMSIS Configuration Wizard**.
2. All interfaces are enabled by default. However, each individual instance can be disabled by unchecking the check-box in the **Value** column.
3. After enabling an interface, the adjustable parameters can be configured by typing in the **Value** column or by choosing one of the options from the pull-down menus.
4. After configuring the required interfaces, the file can be saved by choosing **Save** in the **File** menu.

6.3.2 Using a Text-Based Editor

To configure the CMSIS drivers using a text editor:

1. Open the *RTE_Device.h* file in any text editor of preference.
2. Enable the required interfaces by setting the enable (`RTE_interface_ENABLED`) define statements to 1.
3. After enabling the interfaces, configure their parameters by changing the preset values to any values in the range or options specified in the comments preceding the define statements. For example, to change the I2C0 interface's drive strength to 4X, update the `RTE_I2C0_GPIO_DRIVE_OPTION` define to 2, as mentioned in the comments that precede it.
4. Save the file after all the parameters have been configured.

6.4 Using the CMSIS Drivers

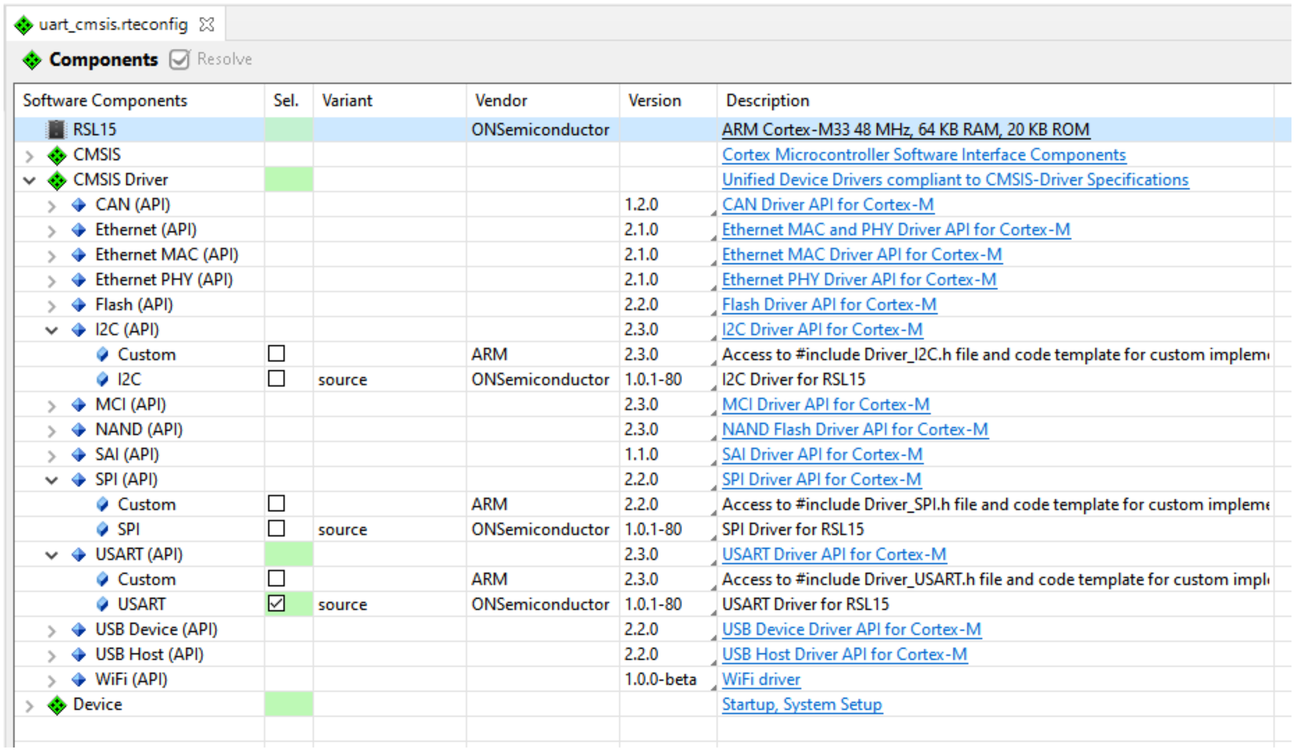
CMSIS drivers provide generic peripheral interfaces for middleware and application code. To use the CMSIS driver library, the drivers must be included in the application. Instructions for using the CMSIS drivers are shown in the following sections.

6.4.1 Adding a CMSIS Driver

To add a CMSIS driver to an application, perform the following steps:

1. Open the **`<sample_name>`***.rtconfig* file from the project folder and select the CMSIS driver(s) that is/are required for the application. CMSIS drivers, including UART, SPI and I2C, are found under **CMSIS Drivers** in the application's *.rteconfig* file.
2. After enabling required drivers, save the **`<sample_name>`***.rtconfig* file.

See the "Adding the UART CMSIS Driver to the uart_cmsis Sample Application" figure (Figure 1) as an example of how to add the USART CMSIS driver to the *uart_cmsis* sample application. As seen in the figure, the USART driver for Montana is checked, and is therefore included in the sample application.

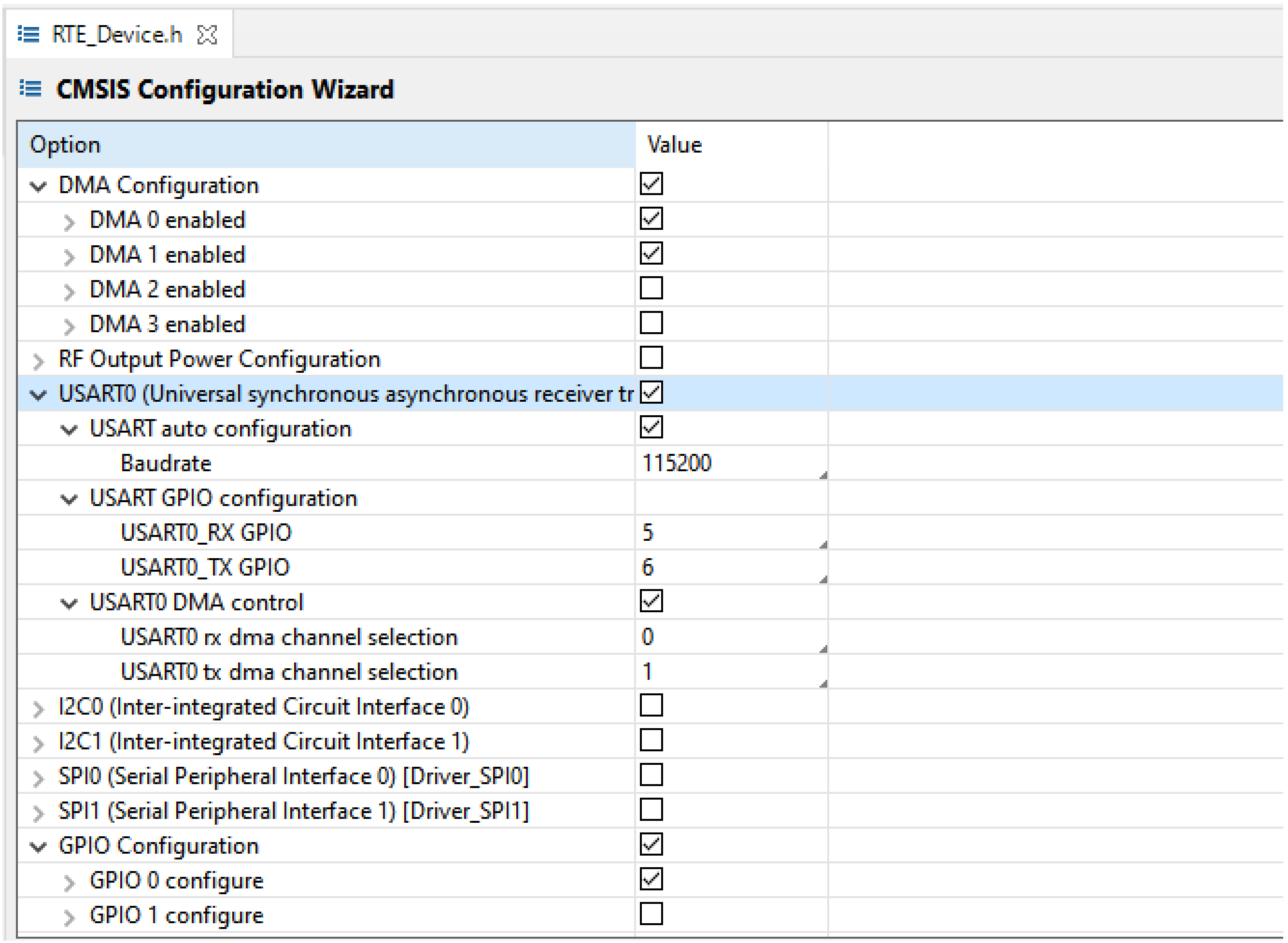


Software Components	Sel.	Variant	Vendor	Version	Description
RSL15	🟩		ONSemiconductor		ARM Cortex-M33 48 MHz, 64 KB RAM, 20 KB ROM
CMSIS					Cortex Microcontroller Software Interface Components
CMSIS Driver	🟩				Unified Device Drivers compliant to CMSIS-Driver Specifications
CAN (API)				1.2.0	CAN Driver API for Cortex-M
Ethernet (API)				2.1.0	Ethernet MAC and PHY Driver API for Cortex-M
Ethernet MAC (API)				2.1.0	Ethernet MAC Driver API for Cortex-M
Ethernet PHY (API)				2.1.0	Ethernet PHY Driver API for Cortex-M
Flash (API)				2.2.0	Flash Driver API for Cortex-M
I2C (API)				2.3.0	I2C Driver API for Cortex-M
Custom	☐		ARM	2.3.0	Access to #include Driver_I2C.h file and code template for custom implem
I2C	☐	source	ONSemiconductor	1.0.1-80	I2C Driver for RSL15
MCI (API)				2.3.0	MCI Driver API for Cortex-M
NAND (API)				2.3.0	NAND Flash Driver API for Cortex-M
SAI (API)				1.1.0	SAI Driver API for Cortex-M
SPI (API)				2.2.0	SPI Driver API for Cortex-M
Custom	☐		ARM	2.2.0	Access to #include Driver_SPI.h file and code template for custom impleme
SPI	☐	source	ONSemiconductor	1.0.1-80	SPI Driver for RSL15
USART (API)	🟩			2.3.0	USART Driver API for Cortex-M
Custom	☐		ARM	2.3.0	Access to #include Driver_USART.h file and code template for custom impl
USART	☑	source	ONSemiconductor	1.0.1-80	USART Driver for RSL15
USB Device (API)				2.2.0	USB Device Driver API for Cortex-M
USB Host (API)				2.2.0	USB Host Driver API for Cortex-M
WiFi (API)				1.0.0-beta	WiFi driver
Device	🟩				Startup, System Setup

Figure 1. Adding the UART CMSIS Driver to the uart_cmsis Sample Application

6.4.2 Configuring CMSIS Drivers with RTE_Device.h

CMSIS drivers require I/O pin assignments and optional setup for DMA when being configured for use in a sample application. Both I/O pin assignments and DMA setup are configured in *RTE_Device.h*. In any sample application, the path to find the *RTE_Device.h* file is as follows: *RTE* > *Device* > *Montana* > *RTE_Device.h*. To view and configure *RTE_Device.h* in an easily readable GUI, right click on *RTE_Device.h* in the Project Explorer view and select **Open With** > **CMSIS Configuration Wizard**. The Wizard window opens, as shown in the "Viewing and Configuring RTE_ Device.h with the CMSIS Configuration Wizard" figure (Figure 2).



RTE_Device.h

CMSIS Configuration Wizard

Option	Value
DMA Configuration	☑
DMA 0 enabled	☑
DMA 1 enabled	☑
DMA 2 enabled	☐
DMA 3 enabled	☐
RF Output Power Configuration	☐
USART0 (Universal synchronous asynchronous receiver tr	☑
USART auto configuration	☑
Baudrate	115200
USART GPIO configuration	
USART0_RX GPIO	5
USART0_TX GPIO	6
USART0 DMA control	☑
USART0 rx dma channel selection	0
USART0 tx dma channel selection	1
I2C0 (Inter-integrated Circuit Interface 0)	☐
I2C1 (Inter-integrated Circuit Interface 1)	☐
SPI0 (Serial Peripheral Interface 0) [Driver_SPI0]	☐
SPI1 (Serial Peripheral Interface 1) [Driver_SPI1]	☐
GPIO Configuration	☑
GPIO 0 configure	☑
GPIO 1 configure	☐

Figure 2. Viewing and Configuring RTE_Device.h with the CMSIS Configuration Wizard

NOTE: This file (*RTE_Device.h*) is best viewed using the CMSIS Configuration Wizard.

As seen in the "Viewing and Configuring RTE_Device.h with the CMSIS Configuration Wizard" figure (Figure 2), optional setup for DMA control is available in the sample application's *RTE_Device.h* file via the CMSIS Configuration Wizard. In this file, under **DMA Configuration**, the DMA channels can be configured.

DMA channel configuration can be enabled by checking the box for a channel, and disabled by unchecking it.. In the "Viewing and Configuring RTE_Device.h with the CMSIS Configuration Wizard" figure (Figure 2), the **DMA 0 enabled** and **DMA 1 enabled** boxes are checked, meaning that DMA channels 0 and 1 have their configurations enabled and are ready to be used for data transfers.

The CMSIS Configuration Wizard view of the *RTE_Device.h file* also shows options under **USART0 DMA Control** that allow channel selection for the USART0 RX DMA channel and USAR0 TX DMA channel. In the "Viewing and Configuring RTE_Device.h with the CMSIS Configuration Wizard" figure (Figure 2) the RX DMA channel for the USART0 interface is configured to use DMA channel 0, while the TX DMA channel uses DMA channel 1.

CHAPTER 7

Program ROM

The Montana Program ROM is responsible for ensuring that an Montana device correctly performs a controlled boot sequence allowing application code to execute. The Program ROM does this while taking into account the possible life cycle states of the device, the active power modes, and any required security functionality.

In addition to bringing the system up, the program ROM also ensures that the default state of a device is consistent after a cold boot sequence.

Finally, the ROM provides access to commonly-used functionality that can be called from application code, allowing these functions to be eliminated from the application footprint.

7.1 Overview

7.1.1 Versions

There are three version numbers reported by the Montana ROM. Each version number is defined as a 32-bit value containing major, minor and revision numbers in the form `major.minor.revision`. These items are stored at predefined locations in the ROM program memory, as shown in the "ROM Versions" table (Table 7).

Table 7. ROM Versions

Address	Version Type	Version
0x00000010	Program ROM	1.5.00
0x00000014	Secure Boot ROM Library (Arm)	1.0.00
0x00000018	Flash Library	3.0.00

7.2 ROM Initialization Sequence

The ROM is split into two main parts:

1. The low level initialization and setup, which is written in assembler: this part is responsible for ensuring that the power supplies are set up, and that memory instances are enabled for default power-up conditions.
2. The higher level functionality implemented in C dealing with the various peripherals, life cycle states and security features: these are defined in more detail in Section 7.4 “Security Subsystem” on page 67.

The ROM initialization sequence is shown graphically in the "ROM Initialization Sequence Flowchart" figure (Figure 3). The flowchart describes the various paths through the boot sequence, taking into account the possible life cycle states and power modes.

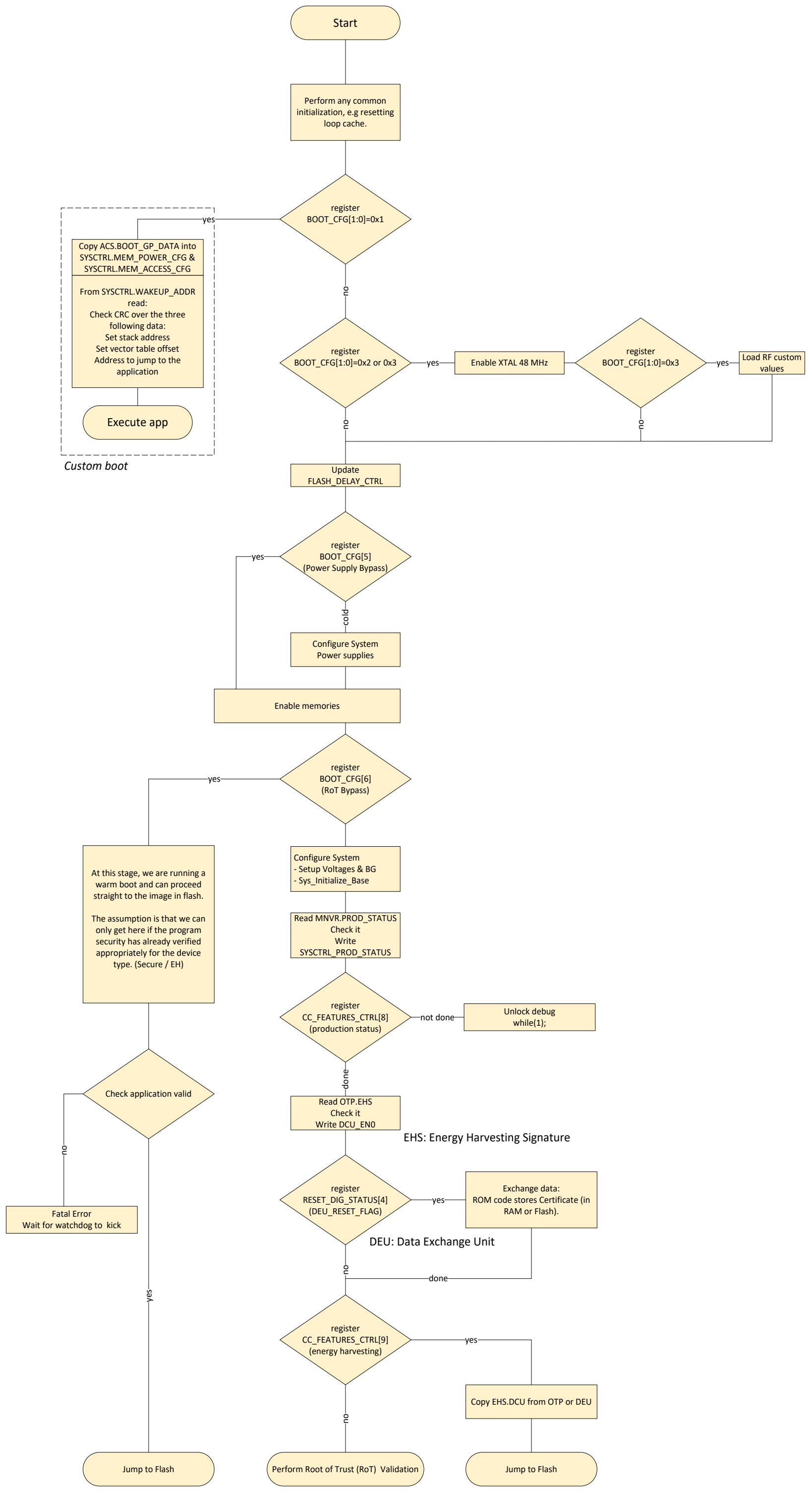

Start
Perform any common initialization, e.g resetting loop cache.
register BOOT_CFG[1:0]=0x1
yes
Copy ACS.BOOT_GP_DATA into SYSCTRL.MEM_POWER_CFG & SYSCTRL.MEM_ACCESS_CFG
From SYSCTRL.WAKEUP_ADDR read: Check CRC over the three following data: Set stack address Set vector table offset Address to jump to the application
Execute app
Custom boot
no
register BOOT_CFG[1:0]=0x2 or 0x3
yes
Enable XTAL 48 MHz
register BOOT_CFG[1:0]=0x3
yes
Load RF custom values
no
no
Update FLASH_DELAY_CTRL
register BOOT_CFG[5] (Power Supply Bypass)
yes
cold
Configure System Power supplies
Enable memories
register BOOT_CFG[6] (RoT Bypass)
yes
At this stage, we are running a warm boot and can proceed straight to the image in flash.
The assumption is that we can only get here if the program security has already verified appropriately for the device type. (Secure / EH)
Configure System - Setup Voltages & BG - Sys_Initialize_Base
Read MNVR.PROD_STATUS Check it Write SYSCTRL_PROD_STATUS
register CC_FEATURES_CTRL[8] (production status)
not done
Unlock debug while(1);
done
Check application valid
no
Fatal Error Wait for watchdog to kick
Read OTP.EHS Check it Write DCU_EN0
EHS: Energy Harvesting Signature
register RESET_DIG_STATUS[4] (DEU_RESET_FLAG)
yes
Exchange data: ROM code stores Certificate (in RAM or Flash).
DEU: Data Exchange Unit
no
done
yes
register CC_FEATURES_CTRL[9] (energy harvesting)
yes
Copy EHS.DCU from OTP or DEU
no
Jump to Flash
Perform Root of Trust (RoT) Validation
Jump to Flash

Figure 3. ROM Initialization Sequence Flowchart

7.2.1 ROM Basic Initialization

The basic system initialization function is called once the power supplies and memories have been set to known default states. At this stage the RAM is enabled and the C stack is available for use.

The primary sequence of this initialization is as follows:

- Set up the NVIC to allow only NMI and Hard Fault exceptions.
- Disable the MPU, bus, and usage faults, forcing them to be always promoted to hard faults.
- Disable all of the external interrupts and clear all pending status indicators.
- Assign the NMI to a constant low source.
- Stop any running flash or flash copier operations.
- Reset the GPIOs.
- Enable all JTAG pins.
- Disable Pad Retention Mode.
- Configure the watchdog timer for a maximum timeout, and refresh.
- Disable DMA.
- Configure Non Secure code zone.
- Disable the SAU and set the ALLNS to zero so everything is secure.
- Configure the non-secure accesses to RAM and peripherals as disabled.
- Make sure the flash is not busy before setting up the RC.
- Configure the RC for 12 MHz.
- Select the RC oscillator as the system clock.
- Set up the various clock pre-scalars.
- Disable RF access.
- Disable the baseband interface.
- Enable all memories.
- Ensure that the Arm CryptoCell-312 (CC312) is enabled and ready to run.
- Set the system core clock variable and flash delays, indicating that the device is configured to use the 12 MHz RC oscillator.

7.3 Vector Tables

The ROM provides some access to built-in common functionality via the use of a ROM Jump Table. The features provided in this jump table are defined in the "Jump Table Functions" table (Table 8), together with a brief description of the use of each. Access to these functions is provided through *rom_vect.h* (typically included through the top-level *montana.h* include file) and *flash_rom.h* (used in place of *flash.h*).

Table 8. Jump Table Functions

Offset	Feature	Description
0x00	__Sys_Initialize_Base void __Sys_Initialize_Base(void);	Base device initialization
0x04	__Sys_Delay void __Sys_Delay(unsigned int cycles);	Delay a number of cycles
0x08	__ProgramROM_ValidateApp uint32_t __ProgramROM_ValidateApp(uint32_t* app_addr);	Validate an application
0x0C	__ProgramROM_StartApp uint32_t __ProgramROM_StartApp(uint32_t* app_addr);	Validate an application, and if valid, start execution of that application
0x10	Flash_Initialize FlashStatus_t Flash_Initialize(unsigned no, FlashClockFrequency_t freq);	Initialize a flash bank
0x14	Flash_WriteWord FlashStatus_t Flash_WriteWord(bool enb_endurance);	Write 32-bit word to flash
0x18	Flash_ReadWord FlashStatus_t Flash_ReadWord(uint32_t addr, uint32_t *word);	Read 32-bit word from flash
0x1C	Flash_WriteDouble FlashStatus_t Flash_WriteDouble(uint32_t addr, const uint32_t *word, bool enb_endurance);	Write two 32-bit words to flash
0x20	Flash_ReadDouble FlashStatus_t Flash_ReadDouble(uint32_t addr, uint32_t *word);	Read two 32-bit words from flash

Table 8. Jump Table Functions (Continued)

Offset	Feature	Description
0x24	Flash_WriteBuffer FlashStatus_t Flash_WriteBuffer(uint32_t addr, uint32_t word_length, const uint32_t *words, bool enb_endurance);	Write an array of 32-bit words
0x28	Flash_ReadBuffer FlashStatus_t Flash_ReadBuffer(uint32_t flash_address, uint32_t dram_address, unsigned word_length);	Read an array of 32-bit words
0x2C	Flash_EraseFlashBank FlashStatus_t Flash_EraseFlashBank(uint32_t no);	Erase a specific flash bank
0x30	Flash_EraseChip FlashStatus_t Flash_EraseChip(void);	Erase all flash
0x34	Flash_EraseSector FlashStatus_t Flash_EraseSector(uint32_t addr, bool enb_endurance);	Erase Flash Sector
0x38	Flash_BlankCheck FlashStatus_t Flash_BlankCheck(uint32_t addr, unsigned word_length);	Verify area of flash is empty
0x3C	__ProgramROM_Read_MNVR void __ProgramROM_Read_MNVR(uint32_t addr, uint32_t *word, uint8_t *readECC);	Read data from MNVR

7.4 Security Subsystem

Montana includes security IP provided by Arm, specifically the Arm CryptoCell-312 (CC312) security processor. In addition, the Arm Cortex-M33 processor itself includes support for TrustZone.

The ROM ensures that the hardware Root-of-Trust (RoT) is verified when running as a secure device. This ensures that no untrusted code can be executed by the ROM.

Montana also supports deployment as a non-secure device, allowing much lower power utilization. This mode is provided primarily for energy harvesting applications. However, this is not limited by design; any application can be executed in the lower security model if required.

IMPORTANT: For detailed information about the device states, the secure Root of Trust (RoT), and tools provided to support the security subsystem, see the *RSL15 Security User's Guide***.**

7.4.1 Secure Boot Process

There are a limited number of steps involved when evaluating the secure Root of Trust (RoT) in the secure state:

- Verify the peripheral ID.
- Get the current life cycle state.
- Secure debug authentication:
    - If debug certificates are available, they must be validated.
    - If valid, debug facilities need to be enabled.
- Image verification phase:
    - Verify the certificate chain.
    - Handle key certificates.
    - Handle content certificates and content.
    - Lock resources.
- In the case of an error occurring during this flow, abort the boot sequence in a secure state.
- If validation completes and the image is authenticated, execute the validated image.

7.5 ROM Life Cycle States (LCS) & Operational Modes

The standard Arm CryptoCell-312 secure life cycle model allows for four life cycle states:

- CM - Chip Manufacture
- DM - Device Manufacture
- SE - Secure

The transitions between these states are strictly defined by the life cycle model, and we have maintained this in our system.

In addition to the standard Arm lifecycle states, we have added an additional production state before CM which reflects the possibility of a device coming from manufacture with an unconfigured OTP. In this case, the ROM does not allow application code to be run, but it does allow the device to be configured to enter CM state.

In parallel to the secure life cycle model, we also provide for a non-secure model in which the device is defined to be in Energy-Harvesting (EH) state. In this case, the Root of Trust security features are turned off and security is provided using a security mechanism dependent on valid applications unlocking the device.

A device in Production state can be configured to be in EH state. If this is not explicitly configured, then the device is treated as a secure device.

- A completely unconfigured device is defined to be in Production state.
    - In this state, the device can be configured as either a secure device or a non-secure device.
    - In this state, no executable code is run on power-up.
    - In manufacturing, we expect all devices to be set as non-secure devices.
- A non-secure device has a signature indicating its state in OTP.
    - In order to transition from the non-secure state to a secure state, this signature needs to be removed.
    - As the signature resides in OTP, it cannot be re-instated once it has been deleted. This provides a oneway transition from non-secure to secure state.
- Once a device is set to secure, there is no mechanism to bring it back to non-secure.

IMPORTANT: If a device is in the Production state, it is possible to go directly to a secure state without transitioning through the non-secure state. In this case, care must be taken to clear the non-secure signature locations, as otherwise it can be possible to revert the device to non-secure. See the *RSL15 Security User's Guide* **for more information on device transitions.**

7.6 Application Validation and Boot

The Montana program ROM contains a set of functions that are used to validate and boot applications, following the completion of the security processes.

The ROM considers an application valid if it starts with its vector table, and no errors that would prevent boot are detected. Possible errors, and the error codes reported for these errors, are described in the "Application Validation" table (Table 9). If the application validated does not successfully boot, the boot ROM writes this error code to VAR_ BOOTROM_ERROR (stored to the `SYSCTRL_SYSCLK_CNT_BASE` register from the activity counters).

Table 9. Application Validation

Error	Error Code	Description
None	0x0	No error detected
Bad Alignment	0x1	The Arm Cortex-M33 processor requires that the application's vector table is aligned to a 512-byte boundary in memory, for a device with the number of external interrupts that are included in the Montana SoC. The location of the specified application is not at a valid location in memory.
Bad Stack Pointer	0x2	The initial stack pointer must point to a valid memory location on the system bus or to a valid memory location in PRAM on the D-code bus. This requires that the specified stack pointer is 32-bit aligned, and that the next address stack data will be placed at is in DRAM, BB_DRAM, or PRAM.

Table 9. Application Validation (Continued)

Error	Error Code	Description
Bad Reset Vector	0x3	The program ROM checks that the reset handler is located immediately after the vector table (or after a CRC located after the vector table). This check is performed indirectly by confirming that the reset vector points to a location that: • Povides space for at least the minimum number of entries in the vector table (a minimum valid vector table contains 4 entries: the stack pointer, reset vector, NMI handler, and hard fault handler) • Provides space for no more than the stack pointer, the 88 potential vectors, and a CRC (maximum of 90 words between the base of the application and the reset vector's location)
Failed to Start the Application	0x6	Indicates that the application has failed to boot or has returned with no identifiable cause.
Bad CRC	0x7	A CRC-CCITT value can be placed between the vector table and the reset handler. The boot validation step validates if a CRC calculated over the vector table matches the value written at this location. NOTE: This error code is considered to be a non-fatal error, since the inclusion of a CRC is optional. The first entry on the application's stack after boot will indicate whether no-error has occurred (0x0) or if a bad CRC has been discovered (0x7).

If the ROM determines that an application should be booted, the ROM:

1. Sets the VTOR bit-field in the Arm Cortex-M33 processor's SCB register to point to the application's vector table
2. Loads the initial stack pointer value from the application's vector table to the Arm Cortex-M33 processor's SP register
3. Pushes the application's status code to the top of the newly defined stack (valid error codes for a booted application are None and Bad CRC, as described in the "Application Validation" table (Table 9))
4. Branches to the beginning of the reset handler, as indicated by the reset vector in the application's vector table

7.7 DATA EXCHANGE UNIT (DEU) SUPPORT

In order to enable the debug ports and any other features controlled by the Debug Control Unit, certificates must be provided to the device. These can reside in RAM for one time only use, or can reside in flash to allow the port enablement to survive a cold reboot.

The provision of these certificates is handled through the DEU. All interfacing with the DEU is performed via the ROM code. Application code has no access to this device.

The ROM supports the following features when communicating with the DEU:

- `LOAD` - Allows a certificate to be loaded to a device and stored in RAM
- `ERASE` - Erases any certificates which are currently stored in flash
- `WRITE` - Writes the current set of certificates from RAM to flash
- `SOCID` - Requests the SOC ID from the device

- `CONNECT` - Connects to a device via the DEU port
- `COMPLETE` - Completes the connection to the device

CHAPTER 8

Flash Library

The flash library provides support for erasing and programming parts of the built in flash memory. This library provides an API that abstracts the details needed properly handle the flash.

8.1 Library Forms

The flash library is available in two forms:

1. As a static library, accessed by including the *flash.h* header file and linking against the *libflashlib.a* library object
2. As a Program ROM component, accessed as memory mapped elements using the *flash_rom.h* header file

For the complete flash library API, refer to Chapter 1 "Flash Library Reference" on page 1.

IMPORTANT: A copy of the flash library has been built into the ROM, as described in Section 7.3 "Vector Tables" on page 64, for use in applications. All functions provided by the flash library must be executed from RAM or ROM, as executing them from flash can result in hidden, flash-access-related failures.

8.2 Flash Library Usage

The flash library can be used to program and erase the built-in flash, with the following considerations:

- A minimum SYSCLK frequency of 1 MHz is required for safe operation of the flash library.
- The MNVR section of flash, outside of the user-defined redundancy sector pointers, cannot be written using the flash library.
- The endurance of flash is limited by the total time flash cells are subjected to program and erase currents. As a result, a number of flash operations provide options around the endurance of the flash cells:
    - Flash writes can use a one stage normal write or a two stage endurance write. For use cases where an area of flash is written many times, a two stage write is recommended.
    - Use of mass erase can speed up flash operations, as it is the quickest way to erase the whole flash array - but use of mass erase subjects the whole array to an erase current for significantly longer than seen with any sector erase operation.
    - Flash sector erases can use a endurance erase that attempts to erase the flash sector up to four times, using a short erase pulse to maximize the number of program/erase cycles that a flash sector can be subjected to. For use cases that require maximum retention time, the normal flash sector erase needs to be used.

For more information about the built-in flash memory, see the *Montana PTS.*

CHAPTER 9

Security and Life Cycle Provisioning Elements

Information about the security and lifetime provisioning elements for Montana is primarily found in the *RSL15 Security User's Guide*.

9.1 Overview

The Security and Life Cycle Provisioning process can be managed using RSLSec, which is a software utility available for Montana. Its command line interface enables users to perform the transitions and access other security features through parameters, as described in the *RSL15 Security User's Guide*. This utility works to leverage the underlying security system for Montana, including the ROM and security IP.

9.2 Third Party Documentation

The Arm TrustZone CryptoCell-312 Software Developers Manual (SW Revision r0p0) is provided with the Montana SDK, in PDF form in the *Arm_documentation* folder.

Additional third-party information can be found on the https://developer.arm.com/ip-products/security-ip/trustzone/trustzone-for-cortex-m webpage.

CHAPTER 10

Arm CryptoCell-312 Security IP

The Montana system includes the Arm CryptoCell-312 IP, which is used to maintain security of the device, allow secure operation of the application firmware, and permit related user applications to access security features. A significant part of the Arm CryptoCell-312's operation is based on the use of the Mbed TLS libraries.

10.1 ARM CRYPTOCELL-312 MBED TLS

Montana provides a standardized interface to the cryptographic features on the device using a customized version of Mbed TLS, which has been tuned to make efficient use of the cryptographic acceleration hardware provided by the Arm CryptoCell-312.

At present, the Mbed TLS variant supported by Montana is version 2.16.2. Some features of this have been replaced by alternative versions to make the best use of the underlying hardware. The interface has been maintained so that standard Mbed TLS applications can work as expected.

For further information on developing cryptographic applications with Montana, refer to the *Arm CryptoCell-312 Runtime Software Developers Manual*, Issue 01, revision r1p3.

CHAPTER 11

Event Kernel

NOTE: Some of the material in this topic has been adapted with permission from CEVA documentation. This chapter is intended to complement the CEVA API documentation included with your Montana install.

The Montana event kernel is a small and efficient event and message handling system that can be used as a Real Time Operating System (RTOS) or as a process executed under an RTOS, offering the following features:

- Exchange of messages
- Message saving
- Timer functionality
- The kernel also provides an event functionality used to defer actions

The purpose of the event kernel is to provide messages (such as the ones in Section 11.2 “Messages” on page 74) and timed tasks to keep RF traffic on schedule and aligned with the specification requirements.

11.1 OVERVIEW

11.1.1 Include Files

The "Kernel File List" table (Table 10) shows a list of the kernel include files, with descriptions.

Table 10. Kernel File List

File	Description
ke.h	Contains the kernel environment definition
ke_event.h	Contains the event handling primitives
ke_mem.h	Contains the implementation of the heap management module
ke_msg.h	This file contains the scheduler primitives called to create or delete a task. It also contains the scheduler itself
ke_task.h	Contains the implementation of the kernel task management
ke_timer.h	Contains the scheduler primitives called to create or delete a timer task. It also contains the timer scheduler itself

11.1.2 Kernel Environment

The kernel environment structure contains the queues used for event, timer and message management, including:

- A queue of sent messages that have not yet been delivered to the receiver
- A queue of messages delivered to the receiver, but not yet consumed
- A queue for timer events

11.2 MESSAGES

11.2.1 Overview

Message queues provide a mechanism to transmit one or more messages to a task. (Queue names and purposes are defined in Section 11.1.2 “Kernel Environment” on page 74.)

Transmission of messages is performed in three steps:

- Sender task allocates a message structure
- Message parameters are filled
- Message structure is pushed in the kernel

A message is identified by a unique ID composed of the task type and an increasing number. A message has a list of parameters that is defined in a structure (see Section 11.2.2 “Message Format” on page 75).

11.2.2 Message Format

The structure of the message contains:

- `id`: Message identifier
- `dest_id`: Destination kernel identifier
- `src_id`: Source kernel identifier
- `param_len`: Parameter embedded structure length
- `param`: Parameter embedded structure. Must be word-aligned.

The source and destination tasks describe which task the message is coming from/to. As a task can implement multiple instances, these source and destination task identifiers are constructed with both the task index and the task type, as shown in the "Destination or source task identifier construction" figure (Figure 4).

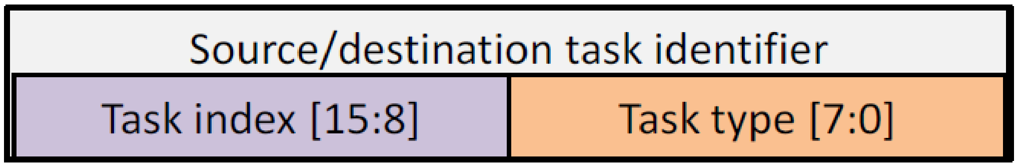


Figure 4. Destination or source task identifier construction

11.2.3 Parameter Management

During message allocation, the size of the parameter is passed and memory is allocated in the kernel heap. To store this data, the pointer on the parameters is returned. The scheduler frees this memory after the transition completion.

11.2.4 Message Queue Primitives

11.2.4.1 Message Allocation

`ke_msg_alloc`

Prototype:

```
void *ke_msg_alloc(ke_msg_id_t const id, ke_task_id_t const dest_id,
               ke_task_id_t const src_id, uint16_t const param_str)
```

Parameters:

Table 11. Message Allocation Parameters

Type	Parameters	Description
`ke_msg_id_t`	`id`	Message Identifier
`ke_task_id_t`	`dest_id`	Destination Task Identifier
`ke_task_id_t`	`src_id`	Source Task Identifier
`uint16_t`	`param_len`	Size of the message parameters to be allocated

Return:

Pointer to the parameter member of the `ke_msg`. If the parameter structure is empty, the pointer points to the end of the message and must not be used (except to retrieve the message pointer or to send the message).

Description:

This primitive allocates memory for a message that has to be sent. The memory is allocated dynamically on the heap, and the length of the variable parameter structure must be provided in order to allocate the correct size.

11.2.4.2 Message Allocation Macro (Static Structure)

```
KE_MSG_ALLOC
```

Prototype:

```
KE_MSG_ALLOC(id, dest_id,
               src_id, param_str)
```

Parameters:

Table 12. Message Allocation Macro Parameters for Static Structures

Type	Parameters	Description
`ke_msg_id_t`	`id`	Message Identifier
`ke_task_id_t`	`dest_id`	Destination Task Identifier
`ke_task_id_t`	`src_id`	Source Task Identifier
`void const *`	`param_str`	Structure tag for the message data

Return:

Pointer to the allocated structure cast to the correct type.

Description:

This macro calls `ke_msg_alloc()` and cast the returned pointer to the appropriate structure. Can only be used if a parameter structure exists for this message (otherwise, use `ke_msg_send_basic()`).

11.2.4.3 Message Allocation Macro (Variable Structure)

```
KE_MSG_ALLOC_DYN
```

Prototype:

```
KE_MSG_ALLOC_DYN(id, dest_id,
                src_id, param_str, length)
```

Parameters:

Table 13. Message Dynamic Allocation Parameters

Type	Parameters	Description
`ke_msg_id_t`	`id`	Message Identifier
`ke_task_id_t`	`dest_id`	Destination Task Identifier
`ke_task_id_t`	`src_id`	Source Task Identifier
`void const *`	`param_str`	Structure tag for the message data
`uint16_t`	`length`	Length of the variable portion of the data structure

Return:

Pointer to the allocated structure of variable length, cast to the correct type.

Description:

This macro calls `ke_msg_alloc()` and cast the returned pointer to the appropriate structure. Can only be used if a parameter structure exists for this message (otherwise, use `ke_msg_send_basic()`).

NOTE: This function can only be used if the variable data array is located at the end of the structure to be allocated.

11.2.4.4 Message Free

```
ke_msg_free
```

Prototype:

```
void ke_msg_free(struct ke_msg const *param)
```

Parameters:

Table 14. Message Free Parameters

Type	Parameters	Description
`struct ke_msg const *`	`param`	Pointer to the message to be freed

Return:

None

Description:

Free allocated message.

11.2.4.5 Message Free Macro

```
KE_MSG_FREE
```

Prototype:

```
KE_MSG_FREE(param_ptr)
```

Parameters:

Table 15. Message Free Parameters

Type	Parameters	Description
`void const *`	`param_ptr`	Structure tag for the message data

Return:

None

Description:

This macro calls `ke_msg_free()` to free a previously allocated message.

11.2.4.6 Message Send

```
ke_msg_send
```

Prototype:

```
void ke_msg_send(void const *param_ptr)
```

Parameters:

Table 16. Message Send Parameters

Type	Parameters	Description
`void const *`	`param_ptr`	Pointer to the parameter member of the message that is to be sent

Return:

None

Description:

Send a message previously allocated with any `ke_msg_alloc()`-like functions. The kernel takes care of freeing the message memory. Once the function has been called, it is not possible to access the message's data any more as the kernel might have copied the message and freed the original memory.

11.2.4.7 Message Send Basic

```
ke_msg_send_basic
```

Prototype:

```
 void ke_msg_send_basic(ke_msg_id_t const id, ke_task_id_t const dest_id, ke_task_id_t
const src_id)
```

Parameters:

Table 17. Message Send Basic Parameters

Type	Parameters	Description
`ke_msg_id_t`	`id`	Message Identifier
`ke_task_id_t`	`dest_id`	Destination Task Identifier
`ke_task_id_t`	`src_id`	Source Task Identifier

Return:

None

Description:

Send a message that has a zero length parameter member. No allocation is required as this is performed internally.

11.2.4.8 Message Forward

```
ke_msg_forward
```

Prototype:

```
 void ke_msg_forward(void const *param_ptr, ke_task_id_t const dest_id, ke_task_id_t
const src_id)
```

Parameters:

Table 18. Message Forward Parameters

Type	Parameters	Description
`void const *`	`param_ptr`	Pointer to the parameter member of the message that is to be sent
`ke_task_id_t`	`dest_id`	Destination Task Identifier
`ke_task_id_t`	`src_id`	Source Task Identifier

Return:

None

Description:

Forward a message to another task by changing its destination and source task IDs.

11.3 SCHEDULER

11.3.1 Overview

The scheduler is called in the main loop of the user application using the `BLE_Kernel_Process()` function.

IMPORTANT: For maximum stability, ensure that the user application calls `BLE_Kernel_Process()` at least once during each Bluetooth connection interval. For more information, see Section 13.2 "Baseband and Kernel Functions" on page 103.

In the user application's main loop, the kernel checks if the event field is non-null, and executes the event handlers for which the corresponding event bit is set.

11.3.2 Requirements

11.3.2.1 Scheduling Algorithm

The "Scheduling Algorithm" figure (Figure 5) shows how the scheduler handles messages. The message handler pops messages from the message queue, passes them to the pre-defined message handler, and then handles either releasing or saving those messages based on the responses from those handlers.

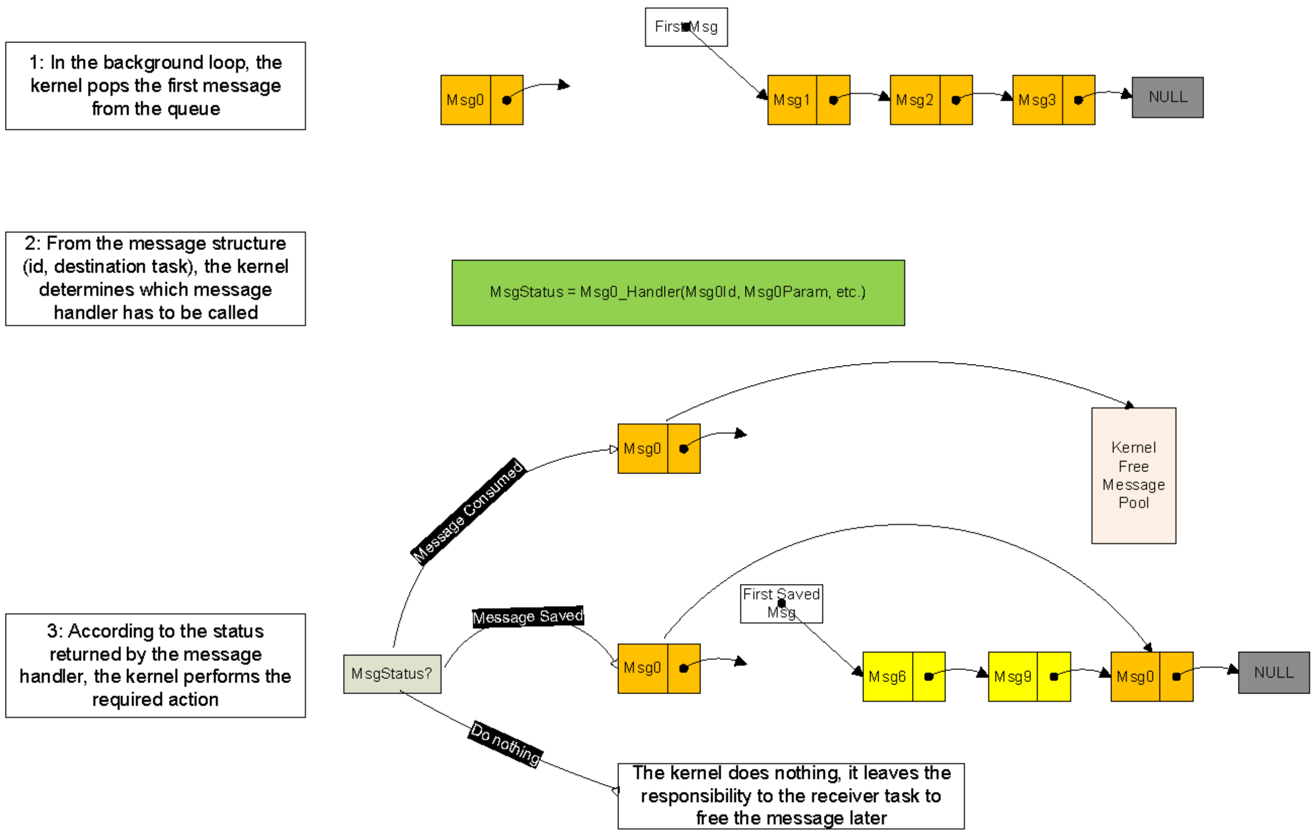

1: In the background loop, the kernel pops the first message from the queue
Msg0
FirstMsg
Msg1
Msg2
Msg3
NULL
2: From the message structure (id, destination task), the kernel determines which message handler has to be called
MsgStatus = Msg0_Handler(Msg0Id, Msg0Param, etc.)
Msg0
Kernel Free Message Pool
Message Consumed
3: According to the status returned by the message handler, the kernel performs the required action
MsgStatus?
Message Saved
Msg0
First Saved Msg
Msg6
Msg9
Msg0
NULL
Do nothing
The kernel does nothing, it leaves the responsibility to the receiver task to free the message later

Figure 5. Scheduling Algorithm

11.3.2.2 Save Service

The Save service can save a message (i.e., store it in memory without it being consumed). If the task state changes after a message is received, the scheduler tries to handle the saved message before scheduling any other signals.

11.4 TASKS

A kernel task is defined by:

- Its task type (i.e., a constant value defined by the kernel, unique for each task)
- Its task descriptor, which is a structure as shown in the "Task Descriptor Construction" figure (Figure 6) containing all the information about the task:
  - The messages handlers table
  - The states table
  - The number of instances of the task
  - The number of messages it can handle

The kernel keeps a pointer to each task descriptor, which is used to handle the scheduling of the messages transmitted from one task to another.

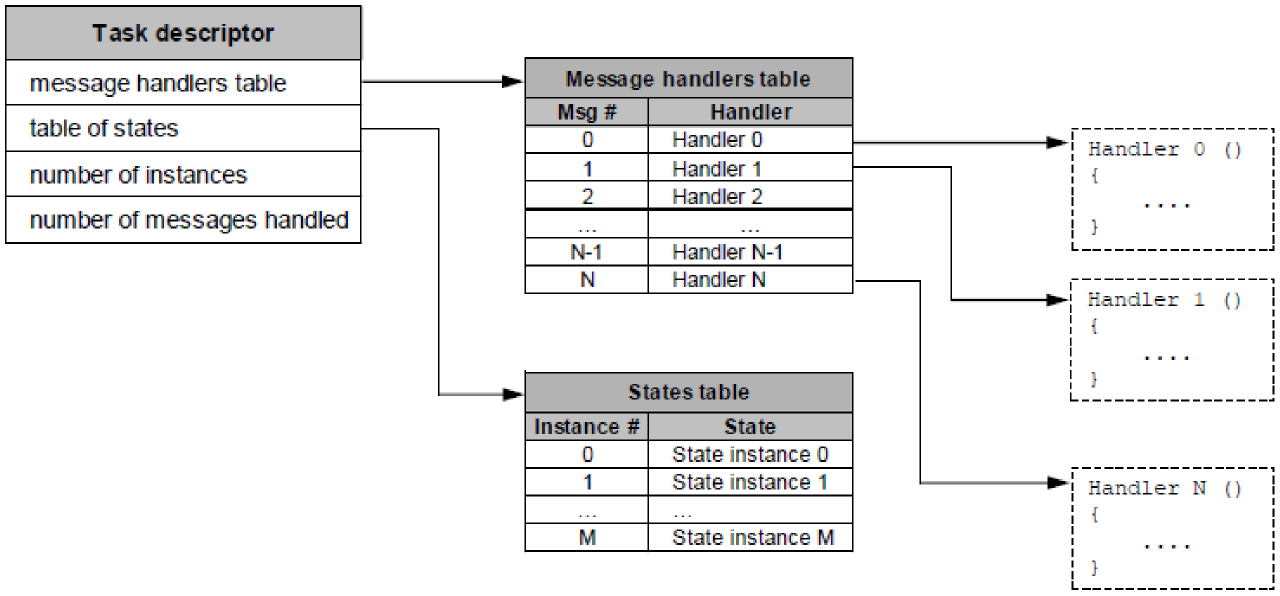


Figure 6. Task Descriptor Construction

11.5 Kernel Timer

11.5.1 Overview

The RW Kernel provides:

- A time reference (absolute time counter)
- Timer services to start and stop the timer

Timers are implemented by means of a reserved queue of delayed messages. Timer messages do not have parameters.

11.5.2 Time Definition

Time is defined as duration; the minimum step is a multiple of 1 ms.

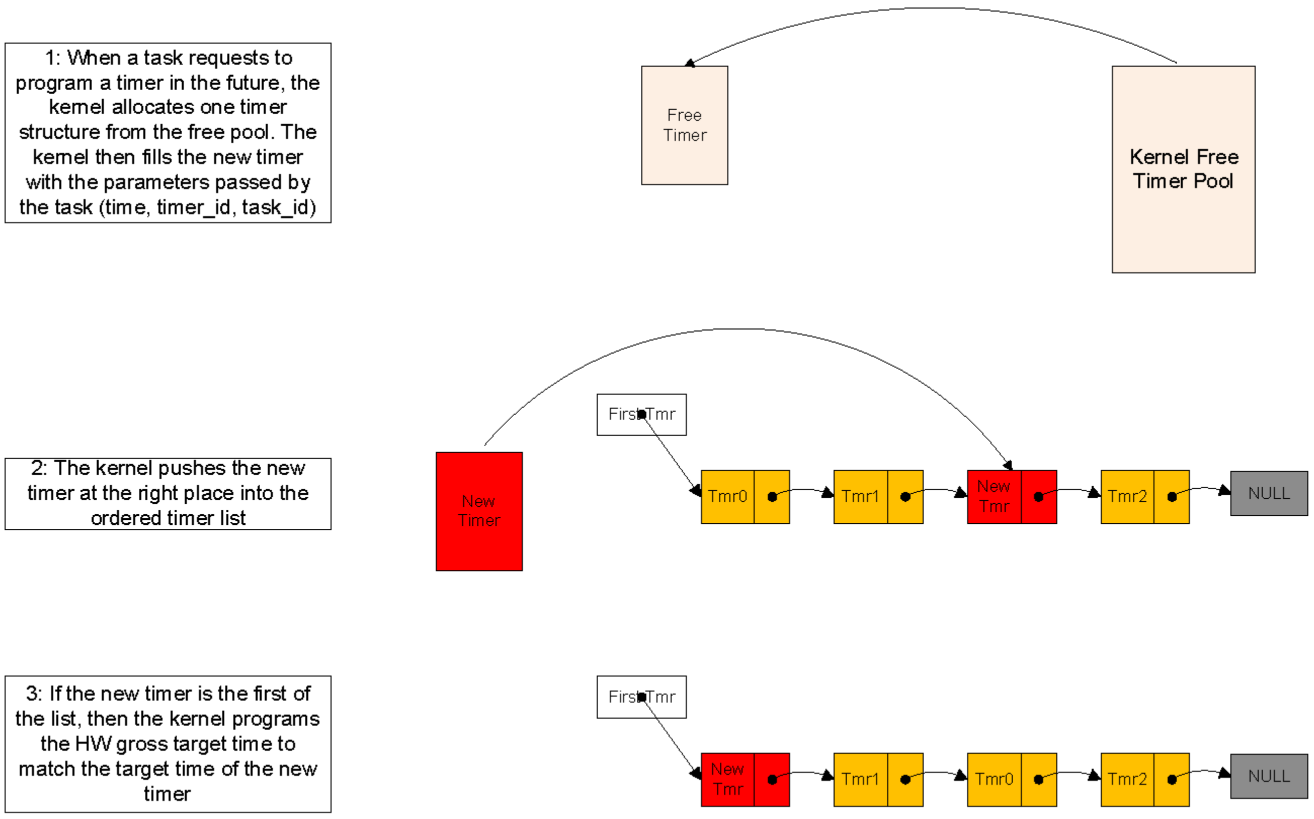
11.5.3 Timer Object

The structure of the timer message contains:

- `*next`: Pointer on the next timer
- `id`: Message identifier
- `task`: Destination task identifier
- `time`: Duration

11.5.4 Timer Setting

The "Timer Setting Flow" figure (Figure 7) shows the flow for setting up timer messages.


1: When a task requests to program a timer in the future, the kernel allocates one timer structure from the free pool. The kernel then fills the new timer with the parameters passed by the task (time, timer_id, task_id)

2: The kernel pushes the new timer at the right place into the ordered timer list

3: If the new timer is the first of the list, then the kernel programs the HW gross target time to match the target time of the new timer

Figure 7. Timer Setting Flow

11.5.5 Time Primitives

11.5.5.1 Timer Set

Start or restart a timer.

Prototype:

```
 void ke_timer_set(ke_msg_id_t const timer_id, ke_task_id_t const task, uint32_t const
delay);
```

Parameters:

Table 19. Timer Set Parameters

Type	Parameters	Description
`ke_msg_id_t`	`timer_id`	Timer identifier
`ke_task_id_t`	`task_id`	Task identifier
`uint16_t`	`delay`	Timer duration (multiple of 1 ms)

Return:

None

Description:

The function first cancels the timer if it exists; then it creates a new one. The timer can be one-shot, or periodic (i.e., it is automatically set again after each trigger).

11.5.5.2 Timer Clear

Remove a registered timer.

Prototype:

```
void ke_timer_clear(ke_msg_id_t const timer_id, ke_task_id_t const task);
```

Parameters:

Table 20. Timer Clear Parameters

Type	Parameters	Description
`ke_msg_id_t`	`timer_id`	Timer identifier
`ke_task_id_t`	`task_id`	Task identifier

Return:

None

Description:

This function searches for the timer element identified by its timer and task identifiers. If found, it is stopped and freed; otherwise an error message is returned.

11.5.5.3 Timer Activity

Check if a requested timer is active.

Prototype:

```
bool ke_timer_active(ke_msg_id_t const timer_id, ke_task_id_t const task);
```

Parameters:

Table 21. Timer Activity Parameters

Type	Parameters	Description
`ke_msg_id_t`	`timer_id`	Timer identifier
`ke_task_id_t`	`task_id`	Task identifier

Return:

`TRUE` if the timer identified by Timer ID is active for the Task ID; `FALSE` otherwise

Description:

This function pops the first timer from the timer queue and notifies the appropriate task by sending a kernel message. If the timer is periodic, it is set again; if it is one-shot, the timer is freed. The function also checks the next timers, and processes them if they have expired or are about to expire. The "Timer Expiry Flow" figure (Figure 8) shows the process flow for handling expired timers.

11.5.5.4 Timer Adjust

Adjust all kernel timers by specified adjustment delay.

Prototype:

```
void ke_timer_adjust_all(uint32_t delay);
```

Parameters:

Table 22. Timer Adjust Parameters

Type	Parameters	Description
`uint16_t`	`delay`	Adjustment delay is a multiple of 1 ms

Return:

None

Description:

This function updates all timers to align to a new `SYSCLK` after a system clock adjustment.

11.5.5.5 Timer Expiry

The "Timer Expiry Flow" figure (Figure 8) shows the flow of timer messages when the timer expires.

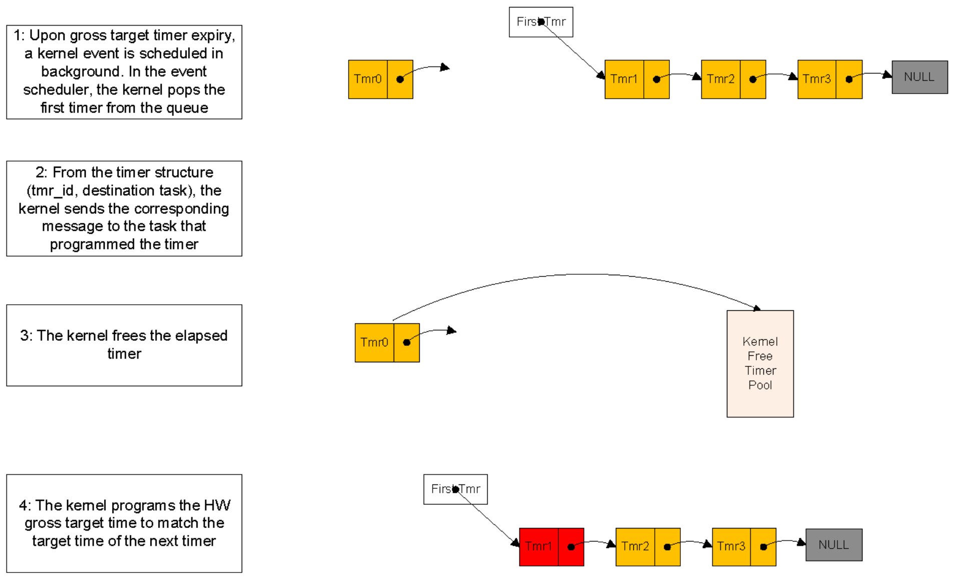

1: Upon gross target timer expiry, a kernel event is scheduled in background. In the event scheduler, the kernel pops the first timer from the queue
FirstTmr
Tmr0
Tmr1
Tmr2
Tmr3
NULL
2: From the timer structure (tmr_id, destination task), the kernel sends the corresponding message to the task that programmed the timer
3: The kernel frees the elapsed timer
Tmr0
Kernel Free Timer Pool
4: The kernel programs the HW gross target time to match the target time of the next timer
FirstTmr
Tmr1
Tmr2
Tmr3
NULL

Figure 8. Timer Expiry Flow

11.6 Useful Macros

- Builds the task identifier from the type and the index of that task:

```
#define KE_BUILD_ID(type, index) ( (ke_task_id_t)(((index) << 8)|(type)) )
```

- Retrieves task type from task id:

```
#define KE_TYPE_GET(ke_task_id) ((ke_task_id) & 0xFF)
```

- Retrieves task index number from task id:

```
#define KE_IDX_GET(ke_task_id) (((ke_task_id) >> 8) & 0xFF)
```

CHAPTER 12

Bluetooth Stack

NOTE: Some of the material in this topic has been adapted with permission from CEVA documentation. This chapter is intended to complement the CEVA API documentation included with your Montana install. Consult this documentation when API function information cannot be found in the CEVA documentation.

This topic explains how the Bluetooth stack, including the HCI (host/controller interface), GATT (generic attribute profile), and GAP (generic access profile), is implemented for Montana. This chapter also provides a description of the Bluetooth profile libraries that are provided with the Montana system to support standard use cases.

IMPORTANT: The default Bluetooth stack is built to support four Bluetooth links, as a balance between system flexibility and power/memory optimization. If your user application requires more links than the default Bluetooth stack library build supports, contact your onsemi Customer Service Representative for assistance.

12.1 INTRODUCTION

12.1.1 Include and Object Files

The Bluetooth stack is accessed through the *ble.h header* file. This header is supported by a set of include files located in the *include\ble* folder of the installation.

The stack is extended by a number of GATT-based profiles and services. The headers for these profiles are located in the *include\ble\profiles* folder, and a list of the available supporting objects is provided in the "Bluetooth GATT-Based Profile and Service Object Files" table (Table 23).

Table 23. Bluetooth GATT-Based Profile and Service Object Files

Profile Name	Profile	Profile Library Name	Profile Description
Battery Service	BAS	libbasc libbass	The Battery Service exposes the state of a battery within a device, or it reads the battery state of a peer device.
Device Information Service	DIS	libdisc libdiss	This service exposes manufacturer and/or vendor information about their own device or discovers peer device information.
Glucose Service	GLS	libglps	This service exposes glucose and other data from a personal glucose sensor for use in consumer healthcare applications.

All of the individual profile libraries use the Bluetooth Low Energy stack through the profile's specified interfaces. These interfaces are documented in the interface specifications. Because the Bluetooth Low Energy stack itself requires a reciprocal link in order to find all of the profile components, the stack library has been built with an object factory that instantiates calls to each of the profiles. If a profile is used by an application, the Bluetooth stack needs to use the specified profile library.

12.1.2 Bluetooth Stack

The Montana device supports a Bluetooth stack through a combination of hardware and firmware resources. The hardware components of the Bluetooth stack are described in the *Montana PTS*. The firmware components of the Bluetooth stack are accessible through a Bluetooth library and associated header files.

The Bluetooth stack is optionally accessible through HCI over UART or through the host (GAP, GATT, L2CAP) and profile APIs.

The "Bluetooth Stack and Kernel Object Files" table (Table 24) describes the Bluetooth stacks provided with Montana and their associated object files.

Table 24. Bluetooth Stack and Kernel Object Files

Stack Type	Library Name	Description
Standard stack	*\lib\ble_core\Release* *\lib\ble_profiles\Release*	These libraries provide the complete standard stack and support for all Montana Bluetooth features.
HCI stack	*\lib\ble_core\Release_HCI* *\lib\ble_profiles\Release_HCI*	These libraries can be used with an HCI interface over UART.

12.1.3 Stack Support Functions

The Bluetooth stack library includes a set of support functions that augment the stack firmware, as described in Chapter 13 "Bluetooth and Kernel Library" on page 101. All other stack APIs are described in their reference documentation, with support for specific Bluetooth layers described in the following documents:

GAP	*RW-BLE-GAP-IS.pdf*
GATT	*RW-BLE-GATT-IS.pdf*
L2CAP	*RW-BLE-L2C-IS.pdf*
Profiles	*RW-BLE-PRF-*-IS.pdf*

12.1.4 Managing Bluetooth Low Energy Stack RAM Usage

The Bluetooth Low Energy stack is provided as a complied library and cannot be modified by the user. However, some of the stack variables are defined at the application level, and the user has some control over the size of these variables. This is a guide to optimizing stack RAM memory usage by adjusting the application level variable defines as dependent upon the application use case.

The application level variables are defined to support the maximum number of connections and activities, which is 10 and 11 respectively, as described in the CEVA documentation provided with Montana. If the application use case does not require the maximum number of connections and activities, application level variable sizes can be reduced to save memory.

NOTE: It is not recommended to adjust application level variable sizes in the HCI stack, as this can adversely affect Bluetooth certification.

The simplest method and recommended starting point to reduce stack memory usage is to reduce the value of `APP_MAX_NB_CON`. This scales most of the other application level variables as well, resulting in a reduction of memory usage. This is usually sufficient for most users. If you are an advanced user requiring further optimization, read on.

The memory allocated by the stack is divided into different parts. The first part comprises the required environment and global variables that are independent of the number of connections or activities. This part cannot be changed by developers.

The second part is heap memory, which the kernel takes as a global variable (*.bss* section), and later the kernel and the Bluetooth Low Energy stack use it as the heap memory for their procedures. It is divided into four parts:

- Environment variables
- Message heap memory
- Data base memory
- Non-retention memory

By default, all parameters are set to address the maximum number of connections and activities, which are 10 and 11 respectively. If definition `APP_HEAP_SIZE_DEFINED` is defined in the *app.h* file of any Bluetooth Low Energy application, then the heap sizes can be customized in the *ble_protocol_support.c* file (where they are defined and allocated). The following code example shows how the required memory sizes can be calculated (defined in *ble_ protocol_support.c*).

```
/// Memory allocated for environment variables
uint32_t rwip_heap_env[RWIP_CALC_HEAP_LEN(APP_RWIP_HEAP_ENV_SIZE)];
/// Memory allocated for Attribute database
uint32_t rwip_heap_db[RWIP_CALC_HEAP_LEN(APP_RWIP_HEAP_DB_SIZE)];
/// Memory allocated for kernel messages
uint32_t rwip_heap_msg[RWIP_CALC_HEAP_LEN(APP_RWIP_HEAP_MSG_SIZE)];
/// Non Retention memory block
uint32_t rwip_heap_non_ret[RWIP_CALC_HEAP_LEN(APP_RWIP_HEAP_NON_RET_SIZE)];
```

We assume:

`APP_MAX_NB_ACTIVITY` and `APP_MAX_NB_CON` are defined in the application, based on the application's use case. Please note that `PP_BLE_CONNECTION_MAX` needs to be at least one value bigger than `APP_MAX_NB_CON`.

```
APP_RWIP_HEAP_ENV_SIZE =
(600 + (APP_MAX_NB_ACTIVITY) * 230)
     + APP_MAX_NB_CON * ((sizeof(struct gapc_env_tag)
     + KE_HEAP_MEM_RESERVED)
     + (sizeof(struct gattc_env_tag)  + KE_HEAP_MEM_RESERVED)
     + (sizeof(struct l2cc_env_tag)   + KE_HEAP_MEM_RESERVED))
     + ((APP_MAX_NB_ACTIVITY ) * (sizeof(struct gapm_actv_scan_tag)
                                        + KE_HEAP_MEM_RESERVED))
```

`APP_RWIP_HEAP_DB_SIZE`: This depends on how much data base memory (GATT services) is added. The default value is 3072 bytes, but developers can choose a smaller value based on their use case.

```
APP_RWIP_HEAP_MSG_SIZE =
(1650 + 2 * ((16 + (APP_MAX_NB_ACTIVITY - 1) * 56)
      + (58 + (APP_MAX_NB_ACTIVITY - 1) * 26)
      + ((APP_MAX_NB_ACTIVITY) * 66)
      + ((APP_MAX_NB_ACTIVITY) * 100)
      +((APP_MAX_NB_ACTIVITY) * 12)))
      +(((BLEHL_HEAP_MSG_SIZE_PER_CON * APP_MAX_NB_CON) > BLEHL_HEAP_DATA_THP_SIZE)
                                                         ?
       (BLEHL_HEAP_MSG_SIZE_PER_CON * APP_MAX_NB_CON) : BLEHL_HEAP_DATA_THP_SIZE)
```

APP_RWIP_HEAP_NON_RET_SIZE: By default, this is set to 656 bytes. It is used for security algorithm calculations. This part of memory does not need to be in retention mode when the use case calls for keeping the Bluetooth Low Energy link active and going to sleep with VDDM in retention. If enough memory is allocated — for example, in the database — the kernel can allocate this part of memory from another heap. Developers need to make sure that this default amount of memory is available, in non-ret heap or in another heap.

In this way, users can set their own APP_MAX_NB_CON and APP_MAX_NB_ACTIVITY values, and decrease the memory size allocated by default settings.

12.2 HCI

The role of the HCI is to provide a uniform interface method of accessing a Bluetooth Low Energy controller's capabilities from the host. The HCI layer is part of the Bluetooth Low Energy protocol stack, as shown in the "Bluetooth Low Energy Protocol Stack" figure (Figure 9).

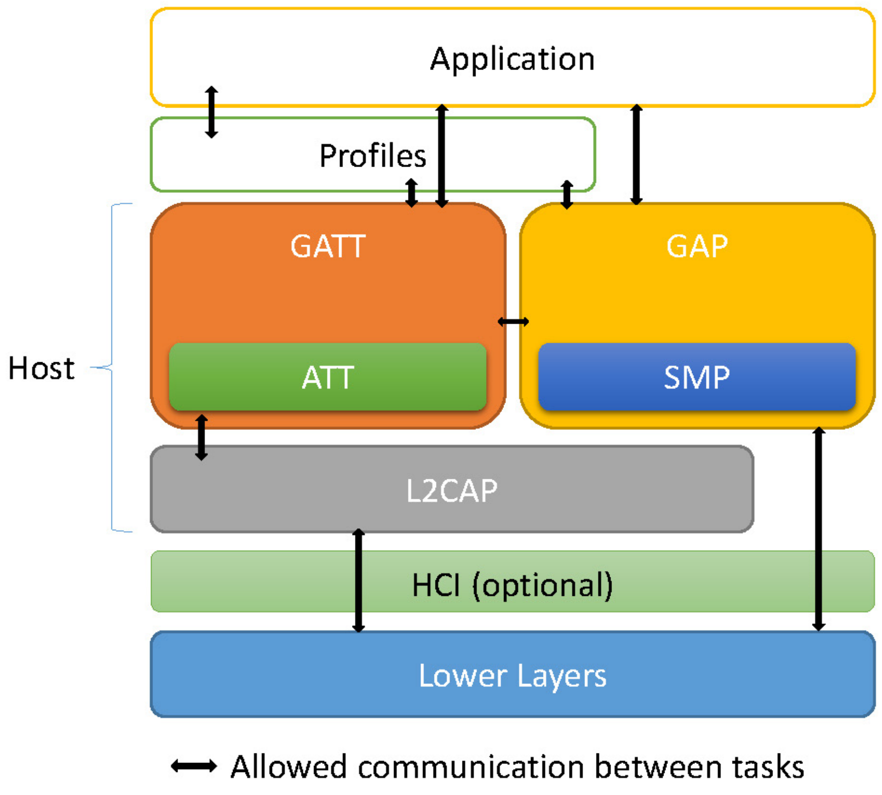


Figure 9. Bluetooth Low Energy Protocol Stack

The Bluetooth stack optionally provides an HCI layer, which provides direct access to the stack at an interface between the host and controller layers. The role of the HCI is to convey the information from one layer to the other by following the rules defined in the HCI portion of the Bluetooth standard. As shown in the sample code, the Montana HCI layer implementation can be used to interface with a transport layer that manages the reception and transmission of messages over a UART physical interface.

As shown in the "HCI" figure (12.2), the two main configurations are supported by the HCI software.

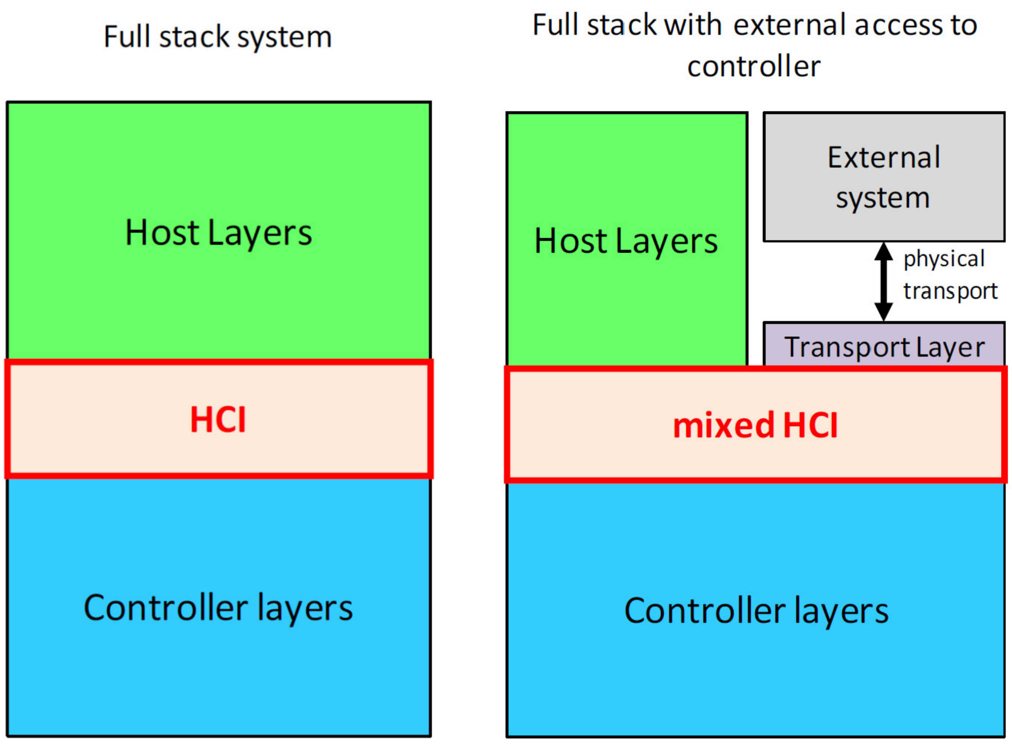


Figure 10. HCI Working Modes

Montana has both a full stack system, and compatibility providing an external system with aaccess to the Bluetooth Controller.

12.2.1 HCI Software Architecture

The HCI software is an interface communication block (depicted in the "HCI Software Interfaces" figure (Figure 11)) that can be used for three main purposes:

1. Communication between internal controller and external host
    - In this case, Montana can be used only as a controller and can communicate to an external host for verification and certification purposes or as a standalone Bluetooth Low Energy controller device (for example, connecting to a PC where an open host stack is running). It is demonstrated using the *hci* sample application from the Montana SDK.

2. Communication between internal controller and internal host
    - In this case, a fully embedded host and application is used. HCI cannot be used with an external UART interface.
3. Communication between internal host and external application
    - In this case, an external application communicates to the Montana host over UART. This interface is not based on the HCI standard (because there is no such use case or standard defined in the Bluetooth core specification). However, an external application can use the same *hci* sample application and use the same kernel messaging and format specified in GAP, GATT, L2CAP, and profile API documentation. The only difference is that the first byte of any message sent or received over UART needs to be 0x05 (`AHI_KE_MSG_TYPE` definition of the Montana Bluetooth Low Energy stack).

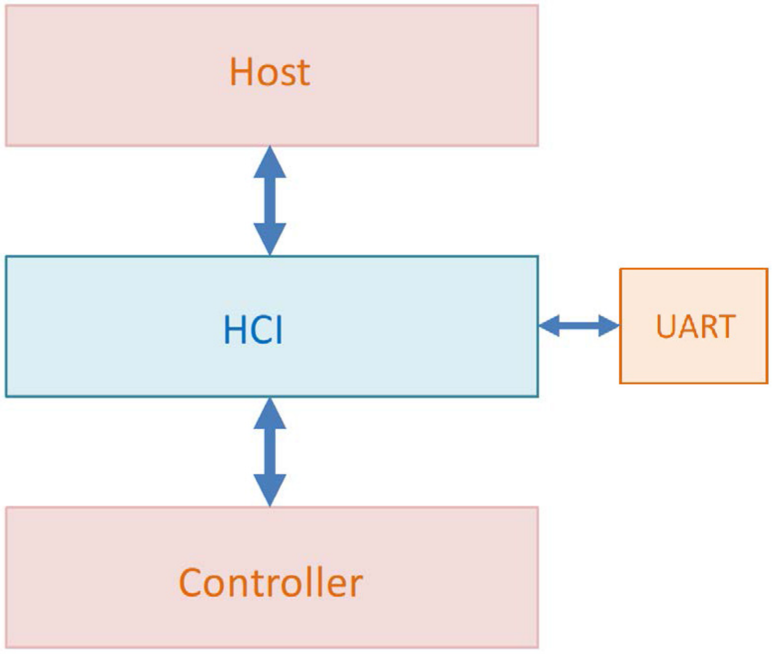


Figure 11. HCI Software Interfaces

12.3 GATT

The GATT is the gateway used by the Attribute Protocol to discover, read, write and obtain indications of the attributes present in the server attribute, and to configure the broadcasting of attributes. The GATT lies above the Attribute Protocol and communicates with the Generic Access Profile (GAP), higher layer profiles, and applications. The architecture of the GATT is shown in 12.3.

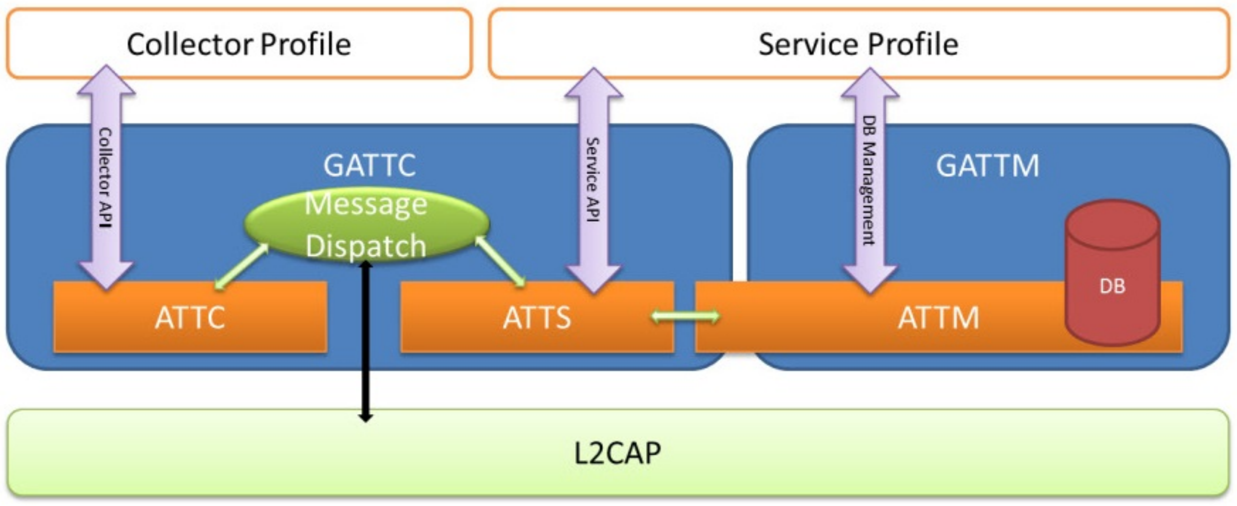


Figure 12. GATT Architecture

For more information about the GATT, and the Bluetooth Stack implementation of GATT, see the *Bluetooth Core Specification* (Volume 3, part G) and the provided CEVA *GATT Interface Specification API* document (*RW-BLE-GATT-IS.pdf*).

12.4 GAP

The Generic Access Profile (GAP) describes the generic procedures related to discovery of Bluetooth devices and link management aspects of connecting to Bluetooth devices. It also defines procedures related to use of different Bluetooth security modes.

The GAP module deals with four features:

1. Management of non-connected activities
2. Management of connected activities
3. Handling of Bluetooth Low Energy security, including pairing, bonding, encryption, and privacy.
4. Handling of life cycle of upper layer profiles

IMPORTANT: The Bluetooth standard for Bluetooth Low Energy provides several pairing schemes that can be used. Use of legacy pairing is not recommended due to known security concerns. We recommend that applications use secure connections for pairing, as per the *Bluetooth® Security and Privacy Best Practices Guide*, due to secure connection's improved overall security including substantially better MITM protection.

For more information about the GAP, and the Bluetooth Stack implementation of GAP, see the *Bluetooth Core Specification* (Volume 3, part C), the *Bluetooth® Security and Privacy Best Practices Guide*, and the provided CEVA *GAP Interface Specification API* document (*RW-BLE-GAP-IS.pdf*).

12.4.1 Non-Connected Procedures

This section describes the support provided to an application that uses non-connected procedures.

Management of these non-connected procedures is based on creation of activities representing the different available procedures. Four kind of activities can be created:

- Advertising activity
- Scanning activity
- Initiating activity
- Periodic Synchronization activity

12.4.1.1 Activity Overview

GAP API provides a set of command messages allowing to:

- Create an activity (`GAPM_ACTIVITY_CREATE_CMD`)
- Start a created activity (`GAPM_ACTIVITY_START_CMD`)
- Stop a started activity (`GAPM_ACTIVITY_STOP_CMD`)
- Delete a created activity (`GAPM_ACTIVITY_DELETE_CMD`)

A descriptions of these commands can be found in the CEVA *Gap Interface Specification*.

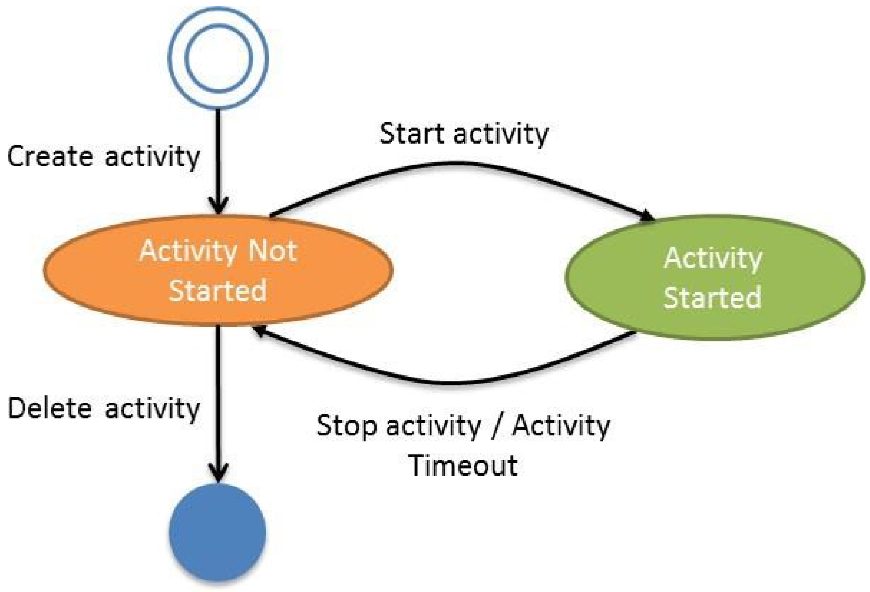


Figure 13. Activity Life Cycle

Figure 13 shows an overview of the life cycle of an activity.

- Before being usable, an activity must first be created.
- The activity must then be started.
- The activity can be considered finished after one of several events has occurred:
    - Request is received from application
    - Timeout
    - End of requested operation (after connection, synchronization,…)
- An activity can either be started again, if the application needs to perform the same procedure, or it can be deleted so that the allocated structure can be reused by another activity.

NOTE: It is not possible to directly update an activity's parameters. Instead, the application must create another activity with different parameters.

The number of activities that can be created in parallel depends on how many activities are supported by the upper layers. However, a few rules exist about the activities that can be started in parallel:

- It is possible to create and start several advertising activities in parallel.
- It is not possible to start two scanning or two initiating activities in parallel, due to HCI commands not allowing management of such operations, because management of such operations is not supported by HCI commands. However, it is possible to create two such activities in parallel.
- It is possible to start one scanning and one initiating activity in parallel with each other.

12.4.1.2 Advertising Activity

An advertising activity can be run on a device that is configured as a broadcaster device.

An advertising activity is defined by its discoverable and connectable modes, as described in Table 25.

Table 25. Advertising Activity Modes

Discoverable Mode	Connectable Mode
Non-discoverable mode: Procedure that can be limited in time. A device in this mode cannot be found by a general or limited discovery procedure. Filtering policy or targeted address can be used in this procedure. In AD_TYPE flag of advertising data, LE general and LE limited discoverable flag are set to zero.	Non-connectable mode: A device in this mode cannot be connected by a central device.
General discoverable mode: Procedure without duration limit. A device in this mode can be found by a general discovery procedure. Whitelist shall not be involved in this mode. In AD_TYPE flag of advertising data, LE general is set to 1 and LE limited discoverable flag is set to zero.	Undirected-connectable mode: A device in this mode can accept connection from any device or from device present in the whitelist.
Limited discoverable mode: Procedure with a limited duration. A device in this mode can be found either by a general or limited discovery procedure. Whitelist is not involved in this mode. In AD_TYPE flag of advertising data, LE general is set to zero and LE limited discoverable flag is set to 1.	Directed-connectable mode: A device in this mode can accept connection only by the targeted address.

In this table, the Periodic Advertising Synchronizability mode and Broadcast mode are missing.

- Periodic Advertising Synchronizability mode is neither a connectable mode nor a discoverable mode. A device in this mode sends synchronization information about periodic advertising.
- Broadcast mode is a non-connectable and non-discoverable mode.

Creation of advertising activity is possible using the `GAPM_ACTIVITY_CREATE_CMD` message, which allows creation of three different types of advertising:

- Legacy advertising activity
- Extended advertising activity
- Periodic advertising activity

NOTE: The discoverability modes such as non-discoverable mode, general discoverable mode and limited discoverable mode are considered as a property of the advertising mode.

12.4.1.2.1 Advertising Properties

The advertising properties are used to describe the content of advertising packets or behavior of the advertising activity. This section provides a small description for each of the configurable properties.

Directed

This property means that a specific device is targeted by the advertiser; the targeted device address is present in the advertising packet. For legacy advertising this applies only in connectable mode, but this is not the case for extended advertising.

High Duty Cycle

This property applies only in legacy direct advertising. The controller advertises the direct connectable packet for 1.28 s, with an advertising interval ≤ 3.75 ms.

Scannable

Advertising activity opens an RX window to receive a scan request packet, and sends in reply a scan response packet.

Connectable

Advertising activity opens an RX window to receive a connect request packet. This property is mandatory to start a connection as slave.

Anonymous

This applies only in extended advertising that is neither connectable nor scannable. The device address is not present at all in the advertised packet, but the packet can contain a targeted address.

TX Power

This applies only in extended advertising, and means that transmit power is present in an advertising packet.

Scan Request Notification

When a scan request packet is received over the air, a scan report is triggered to inform the application about observer device present over the air.

Filter Policy

This indicates if the whitelist is involved, or not to accept scan requests or connect requests. This property applies only for non-discoverable mode advertising.

12.4.1.2.2 Legacy Advertising Activity

The create legacy advertising activity operation allows an application to start legacy advertising on primary advertising channels (37, 38, and 39) at 1 Mb/s. Table 26 shows properties available for each kind of advertising mode supported for legacy advertising.

Table 26. Advertising Properties for Legacy Advertising (Primary Channel Only)

Modes/Properties **0: Must be Disabled** **1: Must be Active** **X: Can be Either Active or Disabled**	**Disc Mode**	**High Duty Cycle**	**Directed**	**Scannable**	**Connectable**	**TX Packet Type**
Non-connectable	Non-disc (Broadcaster mode)	0	0	X	0	
	Limited	0	0	X	0	
	General	0	0	X	0	
Undirected connectable	Non-disc	0	0	1	1	ADV_IND
	Limited	0	0	1	1	
	General	0	0	1	1	
Directed connectable	Non-disc	X	1	0	1	ADV_DIRECT_IND

12.4.1.2.3 Extended Advertising Activity

The create extended advertising activity operation allows an application to start extended advertising on secondary advertising channels (0 to 36) using 1 Mb/s, 2 Mb/s, or LE Coded PHY. Table 27 shows properties available for each kind of advertising mode supported for extended advertising.

NOTE: Advertising extension does not support to have both connectable and scannable modes (scannable means that advertiser replies to `AUX_SCAN_REQ`). So it is possible to set the scan response data only for non-connected modes if the scan property is set.

Table 27. Advertising Properties for Extended Advertising (Secondary Channel Usage)

Modes/Properties 0: Must be Disabled 1: Must be Active X: Can be Either Active or Disabled	Disc Mode	Anonymous	Directed	Scannable	Connectable
Non-connectable	General	0	0	X	0
	Limited	0	0	X	0
	Non-disc (Broadcaster Mode)	0	X	X	0
	Non-disc (Broadcaster Mode)	1	X	0	0
Undirected connectable	Non-disc	0	0	0	1
	General	0	0	0	1
	Limited	0	0	0	1
Directed connectable	Non-disc	0	1	0	1

12.4.1.2.4 Periodic Advertising Activity

The create periodic advertising activity operation allows an application to start a periodic advertising on secondary advertising channels (0 to 36) using 1 Mb/s, 2 Mb/s or LE Coded PHY. In order for a scanner to get information about position of the periodic advertising, a non-connectable and non-scannable extended advertising activity is started.

With this activity it is possible to set advertising data (limited to one fragment) and periodic advertising data.

Table 28 shows properties available for periodic advertising.

Table 28. Advertising Properties for Periodic Advertising (Secondary Channel Usage)

Modes/Properties 0: Must be Disabled 1: Must be Active X: Can be Either Active or Disabled	Disc Mode	Anonymous	Directed	Scannable	Connectable
Periodic Advertising Synchronizability	General	0	0	0	0
	Limited	0	0	0	0
	Non-disc	0	X	0	0

12.4.1.3 Scanning Activity

The purpose of the scanning activity is the reception of advertising packets. The Observer or Central role is mandatory for the creation of a scanning activity. Scanning activities are managed by the GAPM SCAN module.

Six scanning modes are available:

Observer Mode
: A passive or an active scan procedure with non-limited duration. In this mode, the application is notified about any received advertising data, whatever its type.

Selective Observer Mode
: A passive or active scan procedure with non-liminted duration using whitelist filtering.

General Discovery
: A passive or an active scan procedure with a limited duration. In this mode, a device is able to discover advertiser devices broadcasting data in limited or general discoverable mode. Do not use the whitelist.

Limited Discovery
: Passive or an active scan procedure with a limited duration. In this mode, a device is able to discover advertiser devices broadcasting data in limited discoverable mode. Do not use the whitelist.

General Connectable Discovery
: Discover all connectable devices.

Selective Connectable Discovery
: Discover connectable devices using whitelist filtering.

NOTE: Due to the content of HCI LE Extended Set Scan Param/Enable commands, a scanning activity cannot be started in parallel with another scanning activity.

Scan can be performed on LE 1M PHY or LE Coded PHY, or on both PHYs in parallel.

12.4.1.4 Initiating Activity

The purpose of initiating activity is the establishment of a Bluetooth Low Energy connection as master. This implies that support of the Central role is mandatory for creation of an initiating activity. Initiating activities are managed by GAPM INIT module.

Three connection modes are available:

Direct Connection Establishment
: This procedure initiates a connection with a specific device.

Automatic Connection Establishment
: This procedure makes use of the whitelist to establish a connection with known devices. The application needs to set the whitelist before starting this activity. As soon as connection with one of the whitelisted devices is established, the activity autonomously restarts the connection establishment procedure for the next device, until all desired connections have been established.

Name Discovery
: As soon as connection is established, this procedure performs a read of the device name characteristic (GATT UUID 0x2A00) and then disconnects. The device name is sent to the application by using the `GAPM_PEER_NAME_IND` message.

NOTE: Based on the content of the HCI LE Extended Creation Connection command, only one initiating activity can be started at a given time.

12.4.1.5 Periodic Synchronization Activity

The periodic synchronization activity is used to perform a periodic advertising synchronization establishment procedure. Periodic synchronization activity is managed by the `GAPM PER SYNC` module. Observer mode needs to be supported for creation of a periodic synchronization activity.

To establish a synchronization, there are two possible options: waiting information from an existing link using a periodic advertising sync transfer from a peer device, or establishing synchronization without a connection. When not using a connection, it is mandatory to start a scan activity in parallel. This scan activity can be started before or after the periodic synchronization activity.

In the case of a periodic sync transfer, activity must be started with the connection index before expecting sync information from a peer device.

Once the activity has been created, it can be started using the `GAPM_ACTIVITY_START_CMD` message to synchronize with either one specific device (general type) or any of the devices present in the periodic advertising list (selective type).

NOTE: It is not allowed for two periodic synchronization activities to be waiting for synchronization at a same time.

CHAPTER 13

Bluetooth and Kernel Library

This topic describes:

- The custom application programming interface (API) for the event kernel and Bluetooth stack library
- The Bluetooth abstraction API that provides a simplified access to the Bluetooth stack library
- Using the event kernel and Bluetooth stack in an application

13.1 USE OF THE EVENT KERNEL AND BLUETOOTH STACK

Applications using the event kernel and Bluetooth Stack make use of both the custom API, described in Section 13.2 "Baseband and Kernel Functions", and standard event kernel and Bluetooth Stack APIs, as described in the provided CEVA/RivieraWaves documentation.

IMPORTANT: To ensure Bluetooth connection stability, the Event Kernel and Bluetooth stack require that their interrupts be handled in a timely manner. For maximum Bluetooth stability, their interrupts must remain enabled throughout a connection interval and must be configured as the highest priority interrupts in the system to avoid preemption. For more information, see Section 13.2 "Baseband and Kernel Functions" on page 103.

To assist in understanding the event kernel and Bluetooth stack, the following sections outline the customary usage of these components within the context of a typical application use case.

13.1.1 The Kernel Scheduler

The kernel scheduler is responsible for constantly checking messages communicated by different tasks. Applications might use as few as one task, although more complex applications typically include several tasks. Additionally, applications can contain multiple instances of the same task.

Different application tasks control different Bluetooth Low Energy functionalities. An application sometimes needs to communicate with different tasks, such as GAPM, GAPC, GATTM, GATTC, and different standard and custom profiles (services). The kernel scheduler is responsible for handling these messages.

When the application needs to send a message, it allocates memory in the kernel buffer, fills in the message with any parameter that it wants to send, and then fills in the source address and destination address as the message identifier. The application calls a function from the kernel asking to send this message to the destination. The kernel scheduler checks that buffer. If it is filled, the scheduler sends the message to the destination task by automatically calling a pre-defined message handler. The kernel scheduler also handles all timers used by the stack, services, or application. When a timer expires, the kernel scheduler automatically calls a function or message handler allocated to the timer's identifier.

13.1.2 Message Handlers

Coordinating with the event scheduler, the message handlers manage any request or command associated with a scheduled event.

For example, any message, request or command sent from the application to the GAPM has an associated `GAPM_COMPLETE` event. This event message is sent from the stack to the application. Based on its message identifier, this message is called automatically by the kernel.

NOTE: We recommend that users read the CEVA documentation for GATT and GAP before attempting to add their own message handler for a specific API or message.

To add a message handler, you first need to add `DEFINE_MESSAGE_HANDLER` in the corresponding *.h* file, and define a function in the corresponding file, which can be *ble_standard.h*, *application.h*, or *standard_profile.h*.

To add a message handler, users must:

1. Add a task message ID for the handler to the message enumeration used by the application.
2. Add a callback function and function prototype that executes when an event with the task message ID added in step 1 occurs.
3. Register the task message ID and callback function using:

```
MsgHandler_Add(ke_msg_id_t const msg_id, MsgHandlerCallback_t callback);
```

13.1.3 Core Bluetooth Profiles

Core Bluetooth profiles provide the standard communications structures needed to communicate between devices and to organize Bluetooth data. These core profiles are listed here:

- The Generic Access Profile (GAP) contains standard compliant implementations of the broadcasting and connecting mechanisms by which a Bluetooth Low Energy device can communicate with the outside world. The GAP implementation is divided into GAPM (GAP Manager) and GAPC (GAP Controller) services, and is described in the GAP Interface Specification.
- The Generic Attribute Profile (GATT) contains rules for how attributes (data) are formatted, packaged, and sent between Bluetooth Low Energy devices once they have a dedicated connection. The GATT implementation is divided into GATTM (GATT Manager) and GATTC (GATT Controller) services, and is described in CEVA’s GATT Interface Specification.

13.1.4 Standard Profiles and Services

Standard services are pre-defined data structures and methods, which Bluetooth Low Energy devices can use to communicate Bluetooth-specific data between devices in a standardized way. Standard profiles define how a group of one or more services interact and are used.

The attributes of standard services are already recognized by the Bluetooth Low Energy stack, so you do not need to list the attributes when adding standard services to an application. Each standard service has a 16-bit UUID (unique numeric identifier).

13.1.5 Custom Profiles and Services

Custom services, like standard services, are data structures, and the methods which Bluetooth Low Energy devices can use to communicate with, and work with, your Bluetooth Low Energy applications to add functionality. Unlike standard services, custom services are not pre-defined. Users control what custom services do, creating them to meet the needs of their own applications. For custom services, a list all of the attributes needed for the service must be provided to the stack, because the stack cannot automatically tell which attributes to add. Each custom service needs a 128-bit UUID.

Custom profiles can be defined that use both standard and custom services in similar ways to standard profiles.

13.1.6 Event Kernel and Bluetooth Low Energy Library Initialization and Execution

When using the event kernel and Bluetooth stack in a user application, the application must:

- Initialize the event kernel and Bluetooth stack using the `BLE_Initialize()` function, described in Section 13.2 “Baseband and Kernel Functions”.
- Periodically execute the kernel scheduler using the `BLE_Kernel_Process()` function, described in Section 11.3 “Scheduler”.
    - For proper operation, this function must be called in the application’s main loop prior to placing the core into a state waiting for an interrupt.
    - If using low power modes, the `BLE_Baseband_Sleep()` function needs to be called after running the event scheduler before going to sleep. Only transition the device to Sleep Mode or Standby Mode if this function returns `RWIP_DEEP_SLEEP`.

NOTE: While connected to a remote device, the application must allow Bluetooth- and RF-related interrupts to be handled regularly, by not disabling and blocking interrupts for multiple Bluetooth connection periods.

13.2 Baseband and Kernel Functions

To maximize the Bluetooth Low Energy connection stability, the stack must be provided sufficient processing time. We recommended that you:

1. Set the stack interrupts to the highest priority to prevent preemp.tion or ISR processing delays.
2. Minimize the continuous time when interrupts are disabled
3. Have the application call `BLE_Kernel_Process()` at least once per Bluetooth connection interval.

In the current sample code, all Bluetooth and non-Bluetooth interrupt priorities are set to 0 (highest priority). In future SDK updates, the priority of non-Bluetooth interrupts will be lowered. We recommend that customers make this change to their own code to lower the priority of all non-Bluetooth interrupts to maximize the Bluetooth Low Energy connection stability.

Use of the event kernel and baseband are primarily supported by the functions described in the "Event Kernel Support Functions" table (Table 29). These functions initialize and execute these components, while supporting transitions between different power modes. Function prototypes for these functions are included through *ble.h*, and these prototypes are defined in *ble/rwip.h*.

Table 29. Event Kernel Support Functions

Function	Description
BLE_Initialize	Initialize the event kernel and Bluetooth baseband for use within an application.
BLE_Baseband_Sleep	Place the Bluetooth baseband to sleep if the current Bluetooth Low Energy stack and the event kernel state indicate it is safe.
BLE_Kernel_Process	Execute any pending events that have been scheduled with the event kernel.
BLE_Baseband_Is_Awake	Check if the Bluetooth baseband is awake.

13.2.1 BLE_Initialize

Prototype:

```
void BLE_Initialize(uint8_t * param_ptr)
```

Parameters:

Table 30. BLE_Initialize Parameters

Type	Parameters	Description
`uint8_t *`	`param_ptr`	Reserved for future use (input/output parameter)

Return:

None

Description:

Initializes the Bluetooth Low Energy stack and the event kernel for use within an application.

Example

```
/* Initialize the kernel and Bluetooth stack */
uint8_t param;
BLE_Initialize(&param);
```

13.2.2 BLE_Baseband_Sleep

Prototype:

```
uint8_t BLE_Baseband_Sleep(struct ble_sleep_api_param_tag *param_ptr)
```

Parameters:

Table 31. BLE_Baseband_Sleep Parameters

Type	Parameters	Description
`struct ble_sleep_api_param_tag *`	`param_ptr`	param_ptr.app_sleep_request = Application sleep request; set to 1 if the Bluetooth baseband is intended to go to sleep when possible, 0 if the system is meant to stay awake param_ptr.max_sleep_duration = Requested maximum sleep duration in multiples of 312.5 μs (if set to zero, no maximum sleep duration is used) param_ptr.min_sleep_duration = Requested minimum sleep duration in μs param_ptr.calculated_sleep_duration = Return value, in low power baseband clock cycles, providing the calculated time written to the `BB_DEEPSLWKUP` register

NOTE: Minimum sleep duration that `BLE_BASEBAND_SLEEP()` sets is the maximum value of `min_sleep_duration` and the `BB_ENBPRESET_TWOSC` bit field from the `BB_ENBPRESET` register.

Return:

Allowed sleep state.

- `RWIP_DEEP_SLEEP` if the baseband controller has been put to sleep, the baseband timer is set, and the core can switch to Sleep Mode (after saving the baseband and RF registers) or Standby Mode.
- `RWIP_CPU_SLEEP` if the core can be put to sleep, but the system is not meant to enter Sleep Mode or Standby Mode.
- `RWIP_ACTIVE` otherwise.

Description:

A user application needs to call this function when sending the Bluetooth Low Energy stack into Sleep Mode. This function checks whether it is the right time for the event kernel and Bluetooth Low Energy stack to go to sleep or not. If the return value is `RWIP_DEEP_SLEEP`, the BB controller is already in sleep and the BB timer has been started with the desired sleep duration.

Example:

```
/* Attempt to place the device to sleep */
struct ble_sleep_api_param_tag param;
param.app_sleep_request = 1;
param.max_sleep_duration = MAX_SLEEP_DURATION;
param.min_sleep_duration = MIN_SLEEP_DURATION;

switch(BLE_Baseband_Sleep(&param))
{
    case RWIP_DEEP_SLEEP:
    {
        /* Save the baseband and RF registers, then go to sleep */
        BB_Sleep(POWER_MODE);
        break;
    }
    case RWIP_CPU_SLEEP:
    {
        /* Wait for interrupt */
        __WFI();
        break;
    }
    case RWIP_ACTIVE:
    default:
    {
    }
}
```

13.2.3 BLE_Kernel_Process

Prototype:

```
void BLE_Kernel_Process(void)
```

Parameters:

None

Return:

None

Description:

Execute any pending events that have been scheduled with the event kernel.

Example:

```
/* Main application loop:
 * - Run the kernel scheduler
 * - Refresh the watchdog and wait for an interrupt before continuing */
while (1)
{
    BLE_Kernel_Process();

    /* Refresh the watchdog timer */
    SYS_WATCHDOG_REFRESH();

    /* Wait for an event before executing the scheduler again */
    __WFI();
}
```

13.2.4 BLE_Baseband_Is_Awake

Prototype:

```
bool BLE_Baseband_Is_Awake(void)
```

Parameters:

None

Return:

True if the Bluetooth baseband is awake, false otherwise.

Description:

Check if the Bluetooth baseband is awake.

Example:

```
/* Check if the Bluetooth baseband is still awake */
if (BLE_Baseband_Is_Awake())
{
    BLE_Baseband_Sleep(&param);
}
```

13.3 Managing Bluetooth Low Energy Stack RAM Usage

The Bluetooth Low Energy stack is provided as a complied library and cannot be modified by the user. However, some of the stack variables are defined at the application level, and the user has some control over the size of these variables. This is a guide to optimizing stack RAM memory usage by adjusting the application level variable defines as dependent upon the application use case.

The application level variables are defined to support the maximum number of connections and activities, which is 10 and 11 respectively. (See the CEVA documentation provided with your Montana download, in the default location *C:\Users\***<user_name>***\ON_Semiconductor\PACK\ONSemiconductor\Montana\***<version>***\documentation\ceva*.) If the application use case does not require the maximum number of connections and activities, application level variable sizes can be reduced to save memory.

NOTE: It is not recommended to adjust application level variable sizes in the HCI stack, as this can adversely affect Bluetooth certification.

The simplest method and recommended starting point to reduce stack memory usage is to reduce the value of `APP_MAX_NB_CON`. This scales most of the other application level variables as well, resulting in a reduction of memory usage. This is usually sufficient for most users. If you are an advanced user requiring further optimization, read on.

The memory allocated by the stack is divided into different parts. The first part comprises the required environment and global variables that are independent of the number of connections or activities. This part cannot be changed by developers.

The second part is heap memory, which the kernel takes as a global variable (*.bss* section), and later the kernel and the Bluetooth Low Energy stack use it as the heap memory for their procedures. It is divided into four parts:

- Environment variables
- Message heap memory
- Data base memory
- Non-retention memory

By default, all parameters are set to address the maximum number of connections and activities, which are 10 and 11 respectively. If definition `APP_HEAP_SIZE_DEFINED` is defined in the *app.h* file of any Bluetooth Low Energy application, then the heap sizes can be customized in the *ble_protocol_support.c* file (where they are defined and allocated). The following code example shows how the required memory sizes can be calculated (defined in *ble_ protocol_support.c*).

```
/// Memory allocated for environment variables
uint32_t rwip_heap_env[RWIP_CALC_HEAP_LEN(APP_RWIP_HEAP_ENV_SIZE)];
/// Memory allocated for Attribute database
uint32_t rwip_heap_db[RWIP_CALC_HEAP_LEN(APP_RWIP_HEAP_DB_SIZE)];
/// Memory allocated for kernel messages
uint32_t rwip_heap_msg[RWIP_CALC_HEAP_LEN(APP_RWIP_HEAP_MSG_SIZE)];
/// Non Retention memory block
uint32_t rwip_heap_non_ret[RWIP_CALC_HEAP_LEN(APP_RWIP_HEAP_NON_RET_SIZE)];
```

We assume:

`APP_MAX_NB_ACTIVITY` and `APP_MAX_NB_CON` are defined in the application, based on the application's use case. Please note that `PP_BLE_CONNECTION_MAX` needs to be at least one value bigger than `APP_MAX_NB_CON`.

```
APP_RWIP_HEAP_ENV_SIZE =
(600 + (APP_MAX_NB_ACTIVITY) * 230)
     + APP_MAX_NB_CON * ((sizeof(struct gapc_env_tag)
     + KE_HEAP_MEM_RESERVED)
     + (sizeof(struct gattc_env_tag)  + KE_HEAP_MEM_RESERVED)
     + (sizeof(struct l2cc_env_tag)   + KE_HEAP_MEM_RESERVED))
     + ((APP_MAX_NB_ACTIVITY ) * (sizeof(struct gapm_actv_scan_tag)
                                    + KE_HEAP_MEM_RESERVED))
```

APP_RWIP_HEAP_DB_SIZE: This depends on how much data base memory (GATT services) is added. The default value is 3072 bytes, but developers can choose a smaller value based on their use case.

```
APP_RWIP_HEAP_MSG_SIZE =
(1650 + 2 * ((16 + (APP_MAX_NB_ACTIVITY - 1) * 56)
      + (58 + (APP_MAX_NB_ACTIVITY - 1) * 26)
      + ((APP_MAX_NB_ACTIVITY) * 66)
      + ((APP_MAX_NB_ACTIVITY) * 100)
      +((APP_MAX_NB_ACTIVITY) * 12)))
      +(((BLEHL_HEAP_MSG_SIZE_PER_CON * APP_MAX_NB_CON) > BLEHL_HEAP_DATA_THP_SIZE)
                                                       ?
       (BLEHL_HEAP_MSG_SIZE_PER_CON * APP_MAX_NB_CON) : BLEHL_HEAP_DATA_THP_SIZE)
```

APP_RWIP_HEAP_NON_RET_SIZE: By default, this is set to 656 bytes. It is used for security algorithm calculations. This part of memory does not need to be in retention mode when the use case calls for keeping the Bluetooth Low Energy link active and going to sleep with VDDM in retention. If enough memory is allocated — for example, in the database — the kernel can allocate this part of memory from another heap. Developers need to make sure that this default amount of memory is available, in non-ret heap or in another heap.

In this way, users can set their own APP_MAX_NB_CON and APP_MAX_NB_ACTIVITY values, and decrease the memory size allocated by default settings.

13.3.1 Bluetooth Low Energy Link Layer Metrics

A user application can enable link metrics counters through the following API:

```
  uint8_t BLE_Link_Metrics(uint8_t en, uint8_t actidx, struct ble_link_metrics
*metricsPtr)
```

In an application, this funciton can be brought in at any time by setting en to 0x1 and then calling the API. The API is disabled by calling it with en set to zero.

The user application also needs to set the activity index associated with the desired Bluetooth Low Energy link, and to provide a pointer to the area of memory where the Bluetooth Low Energy stack holds the counter values so that the application can read them at any time.

We recommend that the API be activated at link establishment when GAPC_CONNECTION_REQ_IND is received, and disabled at disconnection when GAPC_DISCONNECT_IND is received.

13.3.1.1 BLE_Link_Metrics

```
  uint8_t BLE_Link_Metrics(uint8_t en, uint8_t actidx, struct ble_link_metrics
*metricsPtr)
```

Location: lld_con.c: 3643

This function enables/disables Bluetooth low energy metrics counters.

Returns:

Return value from this function indicates if the requested action (enable/disable) done successfully by the application.

Example code for BLE_Link_Metrics:

```
BLE_Link_Metrics(1, activityIndex, &metricsPtr);
```

Parameters:

Direction	*Name*	Description
in	*en*	Enable value (1=enable, 0=disable)
in	*actidx*	Bluetooth Low Energy link Activity identifier
in	*metricsPtr*	Pointer to ble_link_metrics to hold counter values
out	*Return value*	Return status and indicates if requested action (enable/disable) done successfully by the application • Zero (successful) • Not zero otherwise

13.3.2 Bluetooth Stack Supporting API Functions

13.3.2.1 Device_BLE_Param_Get

Prototype:

```
uint8_t Device_BLE_Param_Get(uint8_t param_id, uint8_t *lengthPtr, uint8_t *buf)
```

This is a callback function that is called by the Bluetooth stack whenever the associated data is required. If the application returns `PARAM_OK`, it means that requested parameter is provided by the application; if it returns `PARAM_FAIL`, it means the stack can use the default settings/parameters.

In this function there is a switch/case, described below:

- `PARAM_ID_BD_ADDRESS`: the global public address of the device
- `PARAM_ID_LPCLK_DRIFT`: the clock accuracy in ppm for the low power clock. There are three clock sources: XTAL32, RC OSC 32, and clock input from a GPIO[0- 3]. By default it is 500 ppm, but the application can provide another value based on the clock accuracy used. For the central role it is not expected to be bigger than 500.
- `PARAM_ID_ACTCLK_DRIFT`: the clock accuracy of XTAL 48 MHz in ppm that depends on the XTAL component used in PCB. If it is not provided by application maximum, 50 ppm will be used.
- `PARAM_ID_OSC_WAKEUP_TIME`: the time that, after wakeup from a low power mode, XTAL48 and other blocks of need time to power up and be ready. The value is in µs.
- `PARAM_ID_CH_ASS_EN`: can be used in a central role to indicate if the channel assessment mechanism needs to be executed by the stack or not. It has True and False values.

* `PARAM_ID_LPCLK_NO_XTAL32K`: determines if a XTAL 32768 Hz is used as the low power clock source or another option of clock source/value is used. If 32768 Hz with maximum 500 ppm is not used, then the actual clock value can be provided by calling `LPCLK_PeriodValue_Set(LPCLK_PERIOD_VALUE)`; refer to sample application code.
* `PARAM_ID_LE_PRIVATE_KEY_P256` and `PARAM_ID_LE_PUBLIC_KEY_P256`: if ECC public and private keys are provided by the application, this can be used for debugging purposes or for use cases where an application needs to generate the keys by itself (using the Arm CryptoCell-312 hardware accelerator or any other way) rather than having them generated by the Bluetooth stack.
* `PARAM_ID_LE_DBG_FIXED_P256_KEY`: values are True or False, indicating if the stack should use forced keys provided by application or not.
* `PARAM_ID_CH_ASSESS_PARAMS`: when a channel assessment algorithm is enabled by the application, the related parameters can be provided through this case that need to follow `struct channel_map_assess_tag`.
* `PARAM_ID_DTM_ANT_ID_TO_PTRN_PARAMS`: for antenna-switching use cases, the antennal pattern table can be provided through this case.
* `PARAM_ID_CUSTOMIZED_HEAP_SIZE`: if an application needs to control and define the memory size used by the Bluetooth stack (stack heap), the four heap sizes values can be provided; otherwise the stack will use default sizes. (Refer to the *Firmware Reference* for details).

13.3.2.2 platform_reset

Prototype:

```
void platform_reset(uint32_t error)
```

This function is called whenever the stack memory is full, so that the application can be notified of the situation and handle it accordingly.

13.3.2.3 srand_func

Prototype:

```
void srand_func(uint32_t seed)
```

This provides a seed value for the `rand()` function used in the stack.

13.3.2.4 rand_func

Prototype:

```
int rand_func(void)
```

Instead of the C `rand` function, the application can provide its own implementation or use the TRNG accelerator from the Arm CryptoCell-312.

13.3.2.5 Device_RF_RSSI_Convert

Prototype:

```
int8_t Device_RF_RSSI_Convert(uint8_t rssi_reg)
```

This function can be used by application to convert or readthe RF RSSI register to an actual dBm value. A general function is provided, but RSSI needs calibration for use cases that need an more accurate RSSI.

13.3.2.6 Device_RF_TxPwr_Get_dBm

Prototype:

```
int8_t Device_RF_TxPwr_Get_dBm(uint8_t txpwr_idx)
```

This callback function is called whenever the stack needs to convert the `RF0_REG1A_PA_PWR_PA_PWR_BYTE` field of register `RF0_REG1A` to a dBm value.

13.3.2.7 Device_RF_TxPwr_Get_Idx

Prototype:

```
uint8_t Device_RF_TxPwr_Get_Idx(int8_t txpwr_dbm)
```

This callback function is used whenever the stack is converting output power in dBm to a value used to set the `RF0_REG1A_PA_PWR_PA_PWR_BYTE` field of register `RF0_REG1A`.

13.3.2.8 BLE_Set_RxStatusCallBack

Prototype:

```
void App_RxStatus_Callback(uint8_t *actidx, uint16_t *status, uint8_t *rssi, uint8_t
*chnl, uint16_t *length)
```

Parameters:

- *actidx
- *status
- *rssi
- *chnl
- *length

Return:

The returned RSSI is a raw RSSI register read value and needs to be converted to dBm.

Description:

This API can register an application callback function, so that whenever a packet is received, the Bluetooth Low Energy stack calls that callback function and returns the listed parameters as arguments of that function.

Examples:

```
/* registering a callback function any time in application after Bluetooth Low Energy
initialization */

BLE_Set_RxStatusCallBack(App_RxStatus_Callback);

/*callback function:*/
```

```
 void App_RxStatus_Callback(uint8_t *actidx, uint16_t *status, uint8_t *rssi, uint8_t
*chnl, uint16_t *length)
 {

 int8_t calculated_rssi = ((0.328 * (*rssi)) - 108);

     if(*status == 0)

 {         swmTrace_printf("\n\r actidx = %d, status = %d, rssi = %d, chnl= %d, length =
%d", *actidx, *status, calculated_rssi, *chnl, *length);    }

 }

 /* When this functionality is required another API needs to be called before BLE_
Initialize() is called, as shown below: */

 BLE_Set_RFOffSeqMode(1);
 uint8_t param_ptr;

 BLE_Initialize(&param_ptr);
```

This information is only valid if the status of the received packet is zero (no error). The callback function does not consume a great deal of processing time, as it is called in the Bluetooth Low Energy RX ISR.

A rough estimation formula can be used as shown below, but in general the RSSI needs to be calibrated to take into account sample-to-sample variation, and channel variation that depends on PCB, XTAL48 trimming, and antenna/matching circuit design. Changes in temperature can also cause variation.

13.3.2.9 BLE_Set_ScanConIndStatusCallBack

Prototype:

```
 BLE_Set_ScanConIndStatusCallBack(App_ScanConIndStatus_Callback);
```

Parameters:

none

Return:

The returned RSSI is a raw RSSI register read value and needs to be converted to dBm.

Description:

Through this function, it is possible to register an application callback function to access the channel number and RSSI for any successfully received scan request and connection indication. It requires that `BLE_Set_RFOffSeqMode (1)` be called in advance.

Example:

```
 BLE_Set_ScanConIndStatusCallBack(App_ScanConIndStatus_Callback);

 void App_ScanConIndStatus_Callback(uint8_t *type, uint8_t *rssi, uint8_t *chnl)

 {
    int8_t calculated_rssi = ((0.328 * (*rssi)) - 108);
    if(*type == BLE_SCAN_REQ)

 {        swmTrace_printf("\n\r  SCAN_REQ, rssi = %d, chnl= %d", calculated_rssi,
*chnl);       }

     else if(*type == BLE_CONNECT_IND)

 {          swmTrace_printf("\n\r  CONNECT_IND, rssi = %d, chnl= %d", calculated_rssi,
*chnl);       }

 }
```

13.3.2.10 Hci_Vs_Cmd_App_Func

Prototype:

```
 uint8_t Hci_Vs_Cmd_App_Func(uint8_t cmd_code,
uint8_t length, uint8_t *data_buf,
uint8_t *result_length, uint8_t *result_data)
```

Parameters:

- cmd_code
- length
- *data_buf
- *result_length
- *result_data

Return:

Status.

Description:

When running an hci application that uses the HCI variant of the Bluetooth Low Energy stack to support HCI commands over UART, `HCI_DBG_VS_APP_CMD_OPCODE` needs to be implemented in the application. This allows users to develop any desired vendor-specific command. The format of the message is handled by the stack, and the application needs to have a function implemented such that it can interpret the command code, input parameters length, and data, and sends a response with the status, length and data. Some of the command codes are used by RF tools, so users can add unused command codes.

Example:

```
uint8_t Hci_Vs_Cmd_App_Func(uint8_t cmd_code,
uint8_t length, uint8_t *data_buf,
uint8_t *result_length, uint8_t *result_data)
{
uint8_t status = CO_ERROR_NO_ERROR;
*result_length = 0;
uint16_t freq_Mhz;
int8_t pwr_dBm, txOrRx; /* 0: Tx, 1:Tx */

switch(cmd_code)
```

```
 { case HCI_VS_RF_CW_ENABLE_CMD_CODE: txOrRx = data_buf[0]; freq_Mhz = (data_buf[1] +
(data_buf[2] << 8)); //TODO break;
 case HCI_VS_RF_CW_DISABLE_CMD_CODE: //TODO break; case HCI_VS_RF_OUTPUT_PWR_CMD_CODE:
pwr_dBm =
 (int8_t)data_buf[0]; //TODO To be replace with HAL RF set output power function break;
default: status =
 CO_ERROR_INVALID_HCI_PARAM; break; }

 return(status);

 }
```

The above commands are implemented and can be found in the *hci* sample code.

13.4 Bluetooth Low Energy Abstraction

The Bluetooth Low Energy abstraction is a wrapper for the Bluetooth APIs provided by CEVA for the RSL15 device. This provides a simplified API to access the Bluetooth components, which can be included through the *ble_ abstraction.h* header. Component wrappers include:

Generic Access Profile (GAP)

This profile defines the generic procedures related to discovery of Bluetooth devices (idle mode procedures), link management aspects of connecting to Bluetooth devices (connection mode procedures), and management of security procedures.

As defined in *ble_gap.h* / *ble_gap.c*, this portion of the Bluetooth abstraction API extends the GAP support described in *RW-BLE-GAP-IS.pdf.*

Generic Attribute Profile (GATT)

This profile provides a service framework using the Bluetooth attribute protocol for discovering services, and for reading and writing characteristic values on a peer device.

As defined in *ble_gatt.h* / *ble_gatt.c*, this portion of the Bluetooth abstraction API extends the GATT support described in *RW-BLE-GATT-IS.pdf.*

Protocol Support

As defined in *ble_protocol_support.h* / *ble_protocol_support.*c, this portion of the Bluetooth abstraction API provides access to support components that are used with the Bluetooth stack. This includes:

- The Bluetooth address
- The RF front-end transmit power configuration and received signal strength indication (RSSI)
- Bluetooth parameters
- Random number generation

Bondlist

Support for managing bonding information used by the Bluetooth GAP interface. This includes storage and access to the device addresses and privacy-related identity resolving keys (IRKs) for any devices bonded to this RSL15 device.

This support is defined in *bondlist.h* / *bondlist.c*, and is used with the Bluetooth GAP support.

Message Handler

This block extends the event kernel message handling, described in Section 11.2 “Messages” on page 74. The Bluetooth abstraction API for this support is defined in *msg_handler.h* / *msg_handler.c*.

The Bluetooth Low Energy abstraction is configured by adding the `ble_abstraction` to an application using the *RTE_Device.h* file, as described in Section 6.3 “Configuring The Driver Run-time Environment” on page 59.

CHAPTER 14

Supplemental Calibration Library

The supplemental calibration library provides a set of functions and symbols to calibrate various power blocks and the internal RC oscillators. The library is distributed in source-code and pre-compiled form as part of the Montana SDK. All library functions are blocking and interruptible. The library does not use any timer interrupts or DMA channels.

14.1 USAGE

Each Montana device contains trim settings in NVR7, which are calibrated with the nominal settings for the items that the library calibrates. These values are trimmed and flashed for each device during production. The values in NVR7 are accurate enough for most applications that use the nominal settings. The default trim settings can be loaded via the Hardware Abstraction Library macro `SYS_TRIM_LOAD_DEFAULT`.

These trim settings can be supplemented by additional calibration values. These supplemental values are calibrated using calibration procedures that execute without any external connections, as they utilize the internal analog test bus (AOUT), the crystal oscillators, and the Asynchronous Clock Counter.

IMPORTANT: Before you begin any supplemental calibration, we recommend that you load the default trim settings such that the trim values from NVR7 using `SYS_TRIM_LOAD_DEFAULT` are loaded, as this puts the system in its nominal state and ensures that critical values, such as the bandgap voltage, are as accurate as reasonably possible. The accuracy of the bandgap is critical for the trimming of regulators and for accurate LSAD measurements.

To use the calibration library functions, include *calibrate.h* in the application source and in the application link against the *libcalibratelib.a* library object. For an RTE project, the calibration library can be linked against by selecting **Calibrate** under **Device** > **Libraries**. The calibration library must then be initialized by a call to either `Calibrate_Clock_Initialize` or `Calibrate_Power_Initialize` before calling functions from *calibrate_clock.c* or *calibrate_power.c*, respectively.

14.2 CLOCK CALIBRATION NOTES

The theoretical accuracy of the 32 kHz RC oscillator calibration is correlated with the system clock frequency. As a result, we recommend calibrating while using the RFCLK as the SYSCLK source when calibrating this clock.

The 3 MHz RC oscillator is calibrated relative to the 32768 Hz crystal. If this crystal is not present, this calibration hangs until the watchdog resets the device.

For the complete calibration library API, refer to Chapter 19 "Supplemental Calibration Library Reference" on page 289.

CHAPTER 15

swmTrace Library

The swmTrace library is a complement to the RTT Viewer plugin described in the Diagnostic Strategies chapter of the *RSL15 Developer's Guide*. The swmTrace library is a logging utility, intended to provide debugging capabilities through an application running on the Arm Cortex-M33 core. When using the swmTrace library's functions with the RTT viewer, logging information is color-coded depending on the level of logging used.

15.1 INTRODUCTION

Although this library's expected primary use case is with the SEGGER RTT technology through the SDK's RTT Viewer, you can also use the library with UART simply by linking to a different library variant. There are four library variants that can be selected between without changing the underlying code:

- RTT variants include both blocking and non-blocking debug.
- UART variants include support for DMA and user defined pins for UART, TX and RX, where both variants are non-blocking.

15.2 USAGE

Various levels of logging are available. See *swmTrace_api.h* for full details, but examples include `SWM_LOG_LEVEL_VERBOSE` and `SWM_LOG_TEST_PASS`. All levels of logging use the `SWM_LOG_` prefix.

Sample applications are available specifically to illustrate the usage of the swmTrace library. See *swmTrace_logger* under the sample application folder. Other sample applications also make use of the swmTrace library.

CHAPTER 16

CMSIS Reference

Hardware register abstraction layer for the SOC.

16.1 SUMMARY

Variables

- MONTANA_Sys_Version : Montana firmware version (variable)
- __Heap_Begin__ : Start location for the heap.
- __Heap_Limit__ : Top limit for the heap.
- __stack_limit : Bottom limit for the stack.
- __stack : Start location for the stack.
- __data_init__ : Pointer to the data to used to initialize volatile memory.
- __data_start__ : Start address of the initialized data area in volatile memory.
- __data_end__ : End address of the initialized data area in volatile memory.
- __bss_start__ : Start address of the cleared data area in volatile memory.
- __bss_end__ : End address of the cleared data area in volatile memory.
- __preinit_array_start__ : Weakly defined function list pointer for pre-initialization functions.
- __preinit_array_end__ : Weakly defined pointer to the end of the pre-initialization function list.
- __init_array_start__ : Weakly defined function list pointer for initialization functions.
- __init_array_end__ : Weakly defined pointer to the end of the initialization function list.
- flash_layout : Flash layout for the Montana device.
- SystemCoreClock : Contains the current SYS_CLK frequency, in Hz.

Data Structures

- flash_region : Structure used to define flash regions.

Macros

- MONTANA_SYS_VER_MAJOR : Montana header file major version.
- MONTANA_SYS_VER_MINOR : Montana header file minor version.
- MONTANA_SYS_VER_REVISION : Montana header file revision version.
- MONTANA_SYS_VER : Montana firmware version.
- __ARMv8MML_REV : Arm v8 architecture revision.
- __CM33_REV : Core revision r0p4.
- __FPU_PRESENT : FPU present.
- __DSP_PRESENT : DSP extension present.
- __SAUREGION_PRESENT : SAU regions present.
- __MPU_PRESENT : MPU present.
- __VTOR_PRESENT : VTOR present.
- __NVIC_PRIO_BITS : 3 bits used for interrupt priority levels
- __Vendor_SysTickConfig : Standard SysTick configuration is used.
- I2C_REF_VALID : Validation of I2C register block pointer reference for assert statements.

- SPI_REF_VALID : Validation of SPI register block pointer reference for assert statements.
- UART_REF_VALID : Validation of UART register block pointer reference for assert statements.
- TIMER_REF_VALID : Validation of TIMER register block pointer reference for assert statements.
- DMA_REF_VALID : Validation of DMA register block pointer reference for assert statements.
- FLASH_REF_VALID : Validation of FLASH register block pointer reference for assert statements.
- GPIO_PAD_COUNT : GPIO peripheral definitions.
- GPIO_GROUP_LOW_PAD_RANGE : Number of GPIO pads in the lowest group (all)
- GPIO_EVENT_CHANNEL_COUNT : Number of available GPIO interrupts.
- GPIO_CLK_DIV_COUNT : GPIO clock divisors.
- GPIO0 : GPIO pads definitions
- GPIO1 : GPIO 1.
- GPIO2 : GPIO 2.
- GPIO3 : GPIO 3.
- GPIO4 : GPIO 4.
- GPIO5 : GPIO 5.
- GPIO6 : GPIO 6.
- GPIO7 : GPIO 7.
- GPIO8 : GPIO 8.
- GPIO9 : GPIO 9.
- GPIO10 : GPIO 10.
- GPIO11 : GPIO 11.
- GPIO12 : GPIO 12.
- GPIO13 : GPIO 13.
- GPIO14 : GPIO 14.
- GPIO15 : GPIO 15.
- SYS_DUMMY_READ : Register that always reads back as 0x00000000.
- SYS_DUMMY_WRITE : Register to which writes are ineffective.
- ERRNO_NO_ERROR : No error.
- ERRNO_GENERAL_FAILURE : General error.
- DEFAULT_FREQ : High speed main RC oscillator default frequency set by boot ROM application Default value is 3 MHz uncalibrated.
- STANDBYCLK_DEFAULT_FREQ : Low speed standby RC oscillator default frequency.
- RFCLK_BASE_FREQ : Frequency of the 48 MHz crystal used for the RF front-end.
- EXTCLK_MAX_FREQ : Maximum frequency supported by using an external clock.
- JTCK_MAX_FREQ : Maximum frequency supported by the JTAG interface.
- RCOSC_MAX_FREQ : Maximum frequency supported by the internal RC oscillator.

Functions

- _start : Initialize the application data and start execution with main.
- _sbrk : Increment (or decrement) the top of the heap.
- SystemInit : Initializes the system by clearing and disabling interrupts, updating the SystemCoreClock variable and updating flash timing registers based on the read SYS_CLK.
- SystemCoreClockUpdate : Reads system registers to determine the current system clock frequency, update the SystemCoreClock variable and update the flash timing registers accordingly.

16.2 CMSIS Reference Variable Documentation

16.2.1 MONTANA_Sys_Version

```
const short MONTANA_Sys_Version
```

Location: montana.h:57

Montana firmware version (variable)

16.2.2 __Heap_Begin__

```
uint8_t __Heap_Begin__
```

Location: montana_start.h:46

Start location for the heap.

16.2.3 __Heap_Limit__

```
uint8_t __Heap_Limit__
```

Location: montana_start.h:47

Top limit for the heap.

16.2.4 __stack_limit

```
uint32_t __stack_limit
```

Location: montana_start.h:52

Bottom limit for the stack.

16.2.5 __stack

```
uint32_t __stack
```

Location: montana_start.h:53

Start location for the stack.

16.2.6 __data_init__

```
uint32_t __data_init__
```

Location: montana_start.h:58

Pointer to the data to used to initialize volatile memory.

16.2.7 __data_start__

```
uint32_t __data_start__
```

Location: montana_start.h:59

Start address of the initialized data area in volatile memory.

16.2.8 __data_end__

```
uint32_t __data_end__
```

Location: montana_start.h:60

End address of the initialized data area in volatile memory.

16.2.9 __bss_start__

```
uint32_t __bss_start__
```

Location: montana_start.h:62

Start address of the cleared data area in volatile memory.

16.2.10 __bss_end__

```
uint32_t __bss_end__
```

Location: montana_start.h:63

End address of the cleared data area in volatile memory.

16.2.11 __preinit_array_start__

```
void(* __preinit_array_start__[])(void)
```

Location: montana_start.h:69

Weakly defined function list pointer for pre-initialization functions.

16.2.12 __preinit_array_end__

```
void(* __preinit_array_end__[])(void)
```

Location: montana_start.h:72

Weakly defined pointer to the end of the pre-initialization function list.

16.2.13 __init_array_start__

```
void(* __init_array_start__[])(void)
```

Location: montana_start.h:75

Weakly defined function list pointer for initialization functions.

16.2.14 __init_array_end__

```
void(* __init_array_end__[])(void)
```

Location: montana_start.h:78

Weakly defined pointer to the end of the initialization function list.

16.2.15 flash_layout

```
 const struct flash_region flash_layout[] =
{
    {
        .start = FLASH0_CODE_BASE,
        .end = FLASH0_CODE_TOP,
        .flash = FLASH0,
        .IRQn = FLASH0_COPY_IRQn,
    },
    {
        .start = FLASH1_CODE_BASE,
        .end = FLASH1_CODE_TOP,
        .flash = FLASH1,
        .IRQn = FLASH1_COPY_IRQn,
    },
    {
        .start = FLASH0_DATA_BASE,
        .end = FLASH0_DATA_TOP,
        .flash = FLASH0,
        .IRQn = FLASH0_COPY_IRQn,
    },
    {
        .start = FLASH1_DATA_BASE,
        .end = FLASH1_DATA_TOP,
        .flash = FLASH1,
        .IRQn = FLASH1_COPY_IRQn,
    },
}
```

Location: montana_start.h:106

Flash layout for the Montana device.

16.2.16 SystemCoreClock

```
 uint32_t SystemCoreClock
```

Location: system_montana.h:72

Contains the current SYS_CLK frequency, in Hz.

16.3 CMSIS Reference Data Structures Type Documentation

16.3.1 flash_region

Location: montana_start.h:112

Structure used to define flash regions.

Data Fields

Type	*Name*	Description
`uint32_t`	*start*	First address in the flash region.
`uint32_t`	*end*	Last address in the flash region.
`FLASH_Type *`	*flash*	Flash interface for this region.
`uint32_t`	*IRQn*	Interrupt supporting this flash region.

16.4 CMSIS Reference Macro Definition Documentation

16.4.1 MONTANA_SYS_VER_MAJOR

```
#define MONTANA_SYS_VER_MAJOR 0x02
```

Location: montana.h:43

Montana header file major version.

16.4.2 MONTANA_SYS_VER_MINOR

```
#define MONTANA_SYS_VER_MINOR 0x01
```

Location: montana.h:46

Montana header file minor version.

16.4.3 MONTANA_SYS_VER_REVISION

```
#define MONTANA_SYS_VER_REVISION 0x00
```

Location: montana.h:49

Montana header file revision version.

16.4.4 MONTANA_SYS_VER

```
#define MONTANA_SYS_VER ((MONTANA_SYS_VER_MAJOR << 12) | \
                         (MONTANA_SYS_VER_MINOR << 8)   | \
                         (MONTANA_SYS_VER_REVISION))
```

Location: montana.h:52

Montana firmware version.

16.4.5 __ARMv8MML_REV

```
#define __ARMv8MML_REV 0x0000U
```

Location: montana.h:96

Arm v8 architecture revision.

16.4.6 __CM33_REV

```
#define __CM33_REV 0x0000U
```

Location: montana.h:97

Core revision r0p4.

16.4.7 __FPU_PRESENT

```
#define __FPU_PRESENT 1U
```

Location: montana.h:98

FPU present.

16.4.8 __DSP_PRESENT

```
#define __DSP_PRESENT 1U
```

Location: montana.h:99

DSP extension present.

16.4.9 __SAUREGION_PRESENT

```
#define __SAUREGION_PRESENT 1U
```

Location: montana.h:100

SAU regions present.

16.4.10 __MPU_PRESENT

```
#define __MPU_PRESENT 1U
```

Location: montana.h:101

MPU present.

16.4.11 __VTOR_PRESENT

```
#define __VTOR_PRESENT 1U
```

Location: montana.h:102

VTOR present.

16.4.12 __NVIC_PRIO_BITS

```
#define __NVIC_PRIO_BITS 3U
```

Location: montana.h:103

3 bits used for interrupt priority levels

16.4.13 __Vendor_SysTickConfig

```
#define __Vendor_SysTickConfig 0U
```

Location: montana.h:104

Standard SysTick configuration is used.

16.4.14 I2C_REF_VALID

```
#define I2C_REF_VALID (((uint32_t)(ref) == (uint32_t)I2C) | \
                                        ((uint32_t)(ref) == (uint32_t)I2C0))
```

Location: montana.h:164

Validation of I2C register block pointer reference for assert statements.

16.4.15 SPI_REF_VALID

```
#define SPI_REF_VALID (((uint32_t)(ref) == (uint32_t)SPI) | \
                                        ((uint32_t)(ref) == (uint32_t)SPI0))
```

Location: montana.h:168

Validation of SPI register block pointer reference for assert statements.

16.4.16 UART_REF_VALID

```
#define UART_REF_VALID (((uint32_t)(ref) == (uint32_t)UART) | \
                                        ((uint32_t)(ref) == (uint32_t)UART0))
```

Location: montana.h:172

Validation of UART register block pointer reference for assert statements.

16.4.17 TIMER_REF_VALID

```
#define TIMER_REF_VALID (((uint32_t)(ref) == (uint32_t)TIMER) | \
                                        ((uint32_t)(ref) == (uint32_t)TIMER0) | \
                                                  ((uint32_t)(ref) == (uint32_t)TIMER1) | \
                                                  ((uint32_t)(ref) == (uint32_t)TIMER2) | \
                                                  ((uint32_t)(ref) == (uint32_t)TIMER3))
```

Location: montana.h:177

Validation of TIMER register block pointer reference for assert statements.

16.4.18 DMA_REF_VALID

```
#define DMA_REF_VALID (((uint32_t)(ref) == (uint32_t)DMA) | \
                                                  ((uint32_t)(ref) == (uint32_t)DMA0) | \
                                                  ((uint32_t)(ref) == (uint32_t)DMA1) | \
                                                  ((uint32_t)(ref) == (uint32_t)DMA2) | \
                                                  ((uint32_t)(ref) == (uint32_t)DMA3))
```

Location: montana.h:184

Validation of DMA register block pointer reference for assert statements.

16.4.19 FLASH_REF_VALID

```
#define FLASH_REF_VALID (((uint32_t)(ref) == (uint32_t)FLASH) | \
                                                  ((uint32_t)(ref) == (uint32_t)FLASH0) | \
                                                  ((uint32_t)(ref) == (uint32_t)FLASH1))
```

Location: montana.h:191

Validation of FLASH register block pointer reference for assert statements.

16.4.20 GPIO_PAD_COUNT

```
#define GPIO_PAD_COUNT 16
```

Location: montana.h:205

GPIO peripheral definitions.

Maximum number of GPIO pads

16.4.21 GPIO_GROUP_LOW_PAD_RANGE

```
#define GPIO_GROUP_LOW_PAD_RANGE 16
```

Location: montana.h:206

Number of GPIO pads in the lowest group (all)

16.4.22 GPIO_EVENT_CHANNEL_COUNT

```
#define GPIO_EVENT_CHANNEL_COUNT 8
```

Location: montana.h:207

Number of available GPIO interrupts.

16.4.23 GPIO_CLK_DIV_COUNT

```
#define GPIO_CLK_DIV_COUNT 0
```

Location: montana.h:208

GPIO clock divisors.

16.4.24 GPIO0

```
#define GPIO0 0
```

Location: montana.h:211

GPIO pads definitions

GPIO 0

16.4.25 GPIO1

```
#define GPIO1 1
```

Location: montana.h:212

GPIO 1.

16.4.26 GPIO2

```
#define GPIO2 2
```

Location: montana.h:213

GPIO 2.

16.4.27 GPIO3

```
#define GPIO3 3
```

Location: montana.h:214

GPIO 3.

16.4.28 GPIO4

```
#define GPIO4 4
```

Location: montana.h:215

GPIO 4.

16.4.29 GPIO5

```
#define GPIO5 5
```

Location: montana.h:216

GPIO 5.

16.4.30 GPIO6

```
#define GPIO6 6
```

Location: montana.h:217

GPIO 6.

16.4.31 GPIO7

```
#define GPIO7 7
```

Location: montana.h:218

GPIO 7.

16.4.32 GPIO8

```
#define GPIO8 8
```

Location: montana.h:219

GPIO 8.

16.4.33 GPIO9

```
#define GPIO9 9
```

Location: montana.h:220

GPIO 9.

16.4.34 GPIO10

```
#define GPIO10 10
```

Location: montana.h:221

GPIO 10.

16.4.35 GPIO11

```
#define GPIO11 11
```

Location: montana.h:222

GPIO 11.

16.4.36 GPIO12

```
#define GPIO12 12
```

Location: montana.h:223

GPIO 12.

16.4.37 GPIO13

```
#define GPIO13 13
```

Location: montana.h:224

GPIO 13.

16.4.38 GPIO14

```
#define GPIO14 14
```

Location: montana.h:225

GPIO 14.

16.4.39 GPIO15

```
#define GPIO15 15
```

Location: montana.h:226

GPIO 15.

16.4.40 SYS_DUMMY_READ

```
#define SYS_DUMMY_READ SYSCTRL->PROD_STATUS
```

Location: montana.h:239

Register that always reads back as 0x00000000.

16.4.41 SYS_DUMMY_WRITE

```
#define SYS_DUMMY_WRITE SYSCTRL->CC_DCU_EN0
```

Location: montana.h:242

Register to which writes are ineffective.

16.4.42 ERRNO_NO_ERROR

```
#define ERRNO_NO_ERROR 0x0000
```

Location: montana.h:288

No error.

16.4.43 ERRNO_GENERAL_FAILURE

```
#define ERRNO_GENERAL_FAILURE 0x0001
```

Location: montana.h:291

General error.

16.4.44 DEFAULT_FREQ

```
#define DEFAULT_FREQ 5000000
```

Location: system_montana.h:57

High speed main RC oscillator default frequency set by boot ROM application Default value is 3 MHz uncalibrated.

Assuming a worse case of 5 MHz

16.4.45 STANDBYCLK_DEFAULT_FREQ

```
#define STANDBYCLK_DEFAULT_FREQ 32768
```

Location: system_montana.h:60

Low speed standby RC oscillator default frequency.

16.4.46 RFCLK_BASE_FREQ

```
#define RFCLK_BASE_FREQ 48000000
```

Location: system_montana.h:63

Frequency of the 48 MHz crystal used for the RF front-end.

16.4.47 EXTCLK_MAX_FREQ

```
#define EXTCLK_MAX_FREQ 48000000
```

Location: system_montana.h:65

Maximum frequency supported by using an external clock.

16.4.48 JTCK_MAX_FREQ

```
#define JTCK_MAX_FREQ 48000000
```

Location: system_montana.h:67

Maximum frequency supported by the JTAG interface.

16.4.49 RCOSC_MAX_FREQ

```
#define RCOSC_MAX_FREQ 12000000
```

Location: system_montana.h:69

Maximum frequency supported by the internal RC oscillator.

16.5 CMSIS Reference Function Documentation

16.5.1 _start

```
void _start()
```

Location: montana_start.h:94

Initialize the application data and start execution with main.

Should be called from the reset vector.
Returns:
None

Assumptions

The symbols **data_init**, **data_start**, **data_end**, **bss_start**, **bss_end**, and **stack_limit** are defined when the application is linked.

Assumptions

The symbol flash_layout exists, and is an array of structures containing the start, end, and FLASH_Type* of all the banks of flash, in ascending memory address order, that the data region could be present in.

Returns:

None

Assumptions

The symbols **data_init**, **data_start**, **data_end**, **bss_start**, **bss_end**, and **stack_limit** are defined when the application is linked.

Assumptions

The symbol flash_layout exists, and is an array of structures containing the start, end, and FLASH_Type* of all the banks of flash, in ascending memory address order, that the data region could be present in.

16.5.2 _sbrk

```
int8_t * _sbrk(int increment)
```

Location: montana_start.h:106

Increment (or decrement) the top of the heap.

Returns:

The prior value of the heap top (points to the base of the newly allocated data if the heap was incremented); returns -1 if the function was unable to allocate the requested memory

Assumptions

The symbols **Heap_Begin**, **Heap_Limit** are defined when the application is linked.

Parameters

Direction	Name	Description
`in`	*increment*	Increment to be applied to the top of the heap

16.5.3 SystemInit

```
void SystemInit()
```

Location: system_montana.h:49

Initializes the system by clearing and disabling interrupts, updating the SystemCoreClock variable and updating flash timing registers based on the read SYS_CLK.

16.5.4 SystemCoreClockUpdate

```
void SystemCoreClockUpdate()
```

Location: system_montana.h:79

Reads system registers to determine the current system clock frequency, update the SystemCoreClock variable and update the flash timing registers accordingly.

CHAPTER 17

Hardware Abstraction Layer Reference

Simple hardware abstraction layer library reference.

17.1 Summary

17.2 Activity Counter

Activity counter hardware abstraction layer.

17.2.1 Summary

Functions

- Sys_ACNT_Start : Start activity counter.
- Sys_ACNT_Stop : Stop activity counter.
- Sys_ACNT_Clear : Clear activity counter values.

17.2.2 Activity Counter Function Documentation

17.2.2.1 Sys_ACNT_Start

```
void Sys_ACNT_Start()
```

Location: acnt.h:42

Start activity counter.

Example Code for Sys_ACNT_Start

```
    // Start the activity counter
    Sys_ACNT_Start();
```

17.2.2.2 Sys_ACNT_Stop

```
void Sys_ACNT_Stop()
```

Location: acnt.h:52

Stop activity counter.

Example Code for Sys_ACNT_Stop

```
    // Stop the activity counter
    Sys_ACNT_Stop();
```

17.2.2.3 Sys_ACNT_Clear

```
void Sys_ACNT_Clear()
```

Location: acnt.h:62

Clear activity counter values.

Example Code for Sys_ACNT_Clear

```
    // Clear the current counter values in the activity counters.
    Sys_ACNT_Clear();
```

17.3 BASEBAND INTERFACE

Baseband interface hardware abstraction layer.

17.3.1 Summary

Macros

- BLE_NONE : No Bluetooth Low Energy event.
- BLE_RISING_EDGE : Bluetooth Low Energy rising edge event.
- BLE_FALLING_EDGE : Bluetooth Low Energy falling edge event.
- BLE_TRANSITION : Bluetooth Low Energy transition event.

Functions

- Sys_BBIF_CoexIntConfig : Configure the coexistence interrupts to monitor for Bluetooth and other RF activity.

17.3.2 Baseband Interface Macro Definition Documentation

17.3.2.1 BLE_NONE

```
#define BLE_NONE 0x0U
```

Location: bbif.h:38

No Bluetooth Low Energy event.

17.3.2.2 BLE_RISING_EDGE

```
#define BLE_RISING_EDGE 0x1U
```

Location: bbif.h:41

Bluetooth Low Energy rising edge event.

17.3.2.3 BLE_FALLING_EDGE

```
#define BLE_FALLING_EDGE 0x2U
```

Location: bbif.h:44

Bluetooth Low Energy falling edge event.

17.3.2.4 BLE_TRANSITION

```
#define BLE_TRANSITION 0x3U
```

Location: bbif.h:47

Bluetooth Low Energy transition event.

17.3.3 Baseband Interface Function Documentation

17.3.3.1 Sys_BBIF_CoexIntConfig

```
 void Sys_BBIF_CoexIntConfig(uint32_t edge, uint32_t types, uint32_t cf_ant_
delay, uint32_t cf_powerup)
```

Location: bbif.h:66

Configure the coexistence interrupts to monitor for Bluetooth and other RF activity.

Example Code for Sys_BBIF_CoexIntConfig

```
    // Enable co-exsistence interrupts on BLE rising edge events and while
    // receiving on the BLE communication
    Sys_BBIF_CoexIntConfig(BLE_RISING_EDGE, BLE_RX_BUSY, 0, 0);
```

Parameters

Direction	Name	Description
in	*edge*	The edge used for all coexistence interrupts; use BLE_[NONE | RISING_EDGE | FALLING_EDGE | TRANSITION]
in	*types*	The types of coexistence interrupts that are relevant; use BLE_RX_BUSY, BLE_TX_BUSY, BLE_IN_PROCESS, and/or EVENT_IN_PROCESS
in	*cf_ant_delay*	Correction factor to be applied to account for antenna delays; used to advance the coexistence interrupts by the specified number of uS.
in	*cf_powerup*	Correction factor to be applied to account for power-up delays on TX, RX; used to advance the coexistence interrupts by the specified number of uS.

17.4 Clock Configuration

Clock Hardware Abstraction Layer.

17.4.1 Summary

Variables

- SystemCoreClock : CMSIS required system core clock variable.

Functions

- Sys_Clocks_RCSystemClkConfig : Configure the RC oscillator and system clock.
- Sys_Clocks_SystemClkConfig : Configure the system clock.
- Sys_Clocks_XTALClkConfig : Configure the 48 MHz XTAL Oscillator.
- Sys_Clocks_DividerConfig : Configure the clock divisors for a standard configuration:

17.4.2 Clock Configuration Variable Documentation

17.4.2.1 SystemCoreClock

```
uint32_t SystemCoreClock
```

Location: clock.h:40

CMSIS required system core clock variable.

17.4.3 Clock Configuration Function Documentation

17.4.3.1 Sys_Clocks_RCSystemClkConfig

```
void Sys_Clocks_RCSystemClkConfig(uint32_t cfg, uint32_t rc_cfg)
```

Location: clock.h:51

Configure the RC oscillator and system clock.

Example Code for Sys_Clocks_RCSystemClkConfig

```
    // Enable RC oscillators, set 12 MHz multiplier for start RC oscillator
    // Set nominal trim settings for RC oscillators
    Sys_Clocks_RCSystemClkConfig(SYSCLK_CLKSRC_RCCLK, RC_OSC_12MHZ |
                                 RC_OSC_ENABLE   |
                                 RC32_OSC_NOM    |
                                 RC_OSC_NOM);
```

Parameters

Direction	Name	Description
`in`	*cfg*	Configuration of the system clock source and prescale value; use SYSCLK_CLKSRC_[RCCLK | STANDBYCLK | RFCLK | JTCK], and JTCK_PRESCALE_*
`in`	*rc_cfg*	Configuration for the RC oscillator

17.4.3.2 Sys_Clocks_SystemClkConfig

```
void Sys_Clocks_SystemClkConfig(uint32_t cfg)
```

Location: clock.h:78

Configure the system clock.

Assumptions

The flash delay configuration is correct for the previously selected system clock source and frequency; if also changing the RC oscillator frequency, use Sys_Clocks_RCSystemClkConfig()

Example Code for Sys_Clocks_SystemClkConfig

```
    // Set SYSCLK source to the RF clock.
    Sys_Clocks_SystemClkConfig_Example(SYSCLK_CLKSRC_RFCLK)
```

Parameters

Direction	Name	Description
`in`	*cfg*	Configuration of the system clock source and prescale value; use SYSCLK_CLKSRC_[RCCLK | STANDBYCLK | RFCLK | JTCK], and JTCK_PRESCALE_*

17.4.3.3 Sys_Clocks_XTALClkConfig

```
void Sys_Clocks_XTALClkConfig(uint32_t xtal_prescaler)
```

Location: clock.h:91

Configure the 48 MHz XTAL Oscillator.

Example Code for Sys_Clocks_XTALClkConfig

```
    // Configure the RFCLK to 8 MHz, using the 48 MHz external crystal.
    Sys_Clocks_XTALClkConfig(CK_DIV_1_6_PRESCALE_6)
```

Parameters

Direction	Name	Description
`in`	*xtal_prescaler*	Configuration of the 48MHz XTAL Oscillator as clock and it's prescale value; use CK_DIV_1_6_PRESCALE_[NO_CLOCK_BYTE | 1_BYTE | 2_BYTE | 3_BYTE | 4_BYTE | 5_BYTE | 6_BYTE],

17.4.3.4 Sys_Clocks_DividerConfig

```
 void Sys_Clocks_DividerConfig(uint32_t uartclk_freq, uint32_t sensorclk_freq,
uint32_t userclk_freq)
```

Location: clock.h:140

Configure the clock divisors for a standard configuration:

- SLOWCLK (1 MHz)
- BBCLK (8 MHz)
- DCCLK (4 MHz)
- CPCLK (166 kHz)
- UARTCLK as per uartclk_freq
- SENSOR_CLK as per sensorclk_freq
- USERCLK as per userclk_freq If an exact configuration cannot be found for the desired frequency, the clock divisor will be set to ensure the divided clock does not exceed the specified target frequency.

Note: If SENSOR_CLK is configured to be derived from the RTC timer clock the SENSOR_CLK divisor set by this function is not used.

Assumptions

The system clock has previously been configured.

Example Code for Sys_Clocks_DividerConfig

```
    // Set UARTCLK to 115200 Hz
    // Set SENSORCLK to 1000 Hz
    // Set USERCLK to 1000000 Hz
    // Set other clocks to typical values:
    // SLOWCLK set to 1 MHz
    // BBCLK set to 8 MHz
    // DCCLK set to 4 MHz
    // CPCLK set to 125 kHz
    Sys_Clocks_DividerConfig(115200, 1000, 1000000)
```

Parameters

Direction	Name	Description
in	*uartclk_freq*	Target frequency for the UART clock
in	*sensorclk_freq*	Target frequency for the sensor clock
in	*userclk_freq*	Target frequency for user clock; if the target frequency exceeds the system clock frequency and the RF clock is available, USERCLK will be sourced from RF clock.

17.5 Cyclic Redundancy Check

Cyclic Redundancy Check (CRC) hardware abstraction layer.

17.5.1 Summary

Macros

- SYS_CRC_CONFIG : Macro wrapper for Sys_Set_CRC_Config() Configure the CRC generator type, endianness of the input data, and standard vs non-standard CRC behavior.
- SYS_CRC_32INITVALUE : Macro wrapper for Sys_CRC_32InitValue() Initialize CRC for CRC-32.
- SYS_CRC_CCITTINITVALUE : Macro wrapperr for Sys_CRC_CCITTInitValue() Initialize CRC for CRC-CCITT.
- SYS_CRC_GETCURRENTVALUE : Macro wrapper for Sys_CRC_GetCurrentValue() Initialize CRC for CRC-CCITT.
- SYS_CRC_GETFINALVALUE : Macro wrapper for Sys_CRC_GetFinalValue() Initialize final CRC value.
- SYS_CRC_ADD : Macro wrapper for Sys_CRC_Add() Add data to the current CRC calculation, based on size.

Functions

- Sys_Set_CRC_Config : Configure the CRC generator type, endianness of the input data, and standard vs non-standard CRC behavior.
- Sys_CRC_32InitValue : Initialize CRC for CRC-32.
- Sys_CRC_CCITTInitValue : Initialize CRC for CRC-CCITT.
- Sys_CRC_GetCurrentValue : Retrieve current value from CRC.
- Sys_CRC_GetFinalValue : Initialize final CRC value.
- Sys_CRC_Add : Add data to the current CRC calculation, based on size.

17.5.2 Cyclic Redundancy Check Macro Definition Documentation

17.5.2.1 SYS_CRC_CONFIG

```
#define SYS_CRC_CONFIG Sys_Set_CRC_Config(CRC, (config))
```

Location: crc.h:164

Macro wrapper for Sys_Set_CRC_Config() Configure the CRC generator type, endianness of the input data, and standard vs non-standard CRC behavior.

Assumptions

Note that D_CRC supports only CRC_CCITT mode, hence no configuration is applied for this instance.

Example Code for SYS_CRC_CONFIG

```
    // Configure the default CRC block to CRC-32 (IEEE 802.3) algorithm,
    // using little endian and non-standard (opposite) bit order
    SYS_CRC_CONFIG(CRC_32 | CRC_LITTLE_ENDIAN |
                   CRC_BIT_ORDER_NON_STANDARD);
```

Parameters

Direction	*Name*	Description
in	*config*	CRC generator configuration; use CRC_[CCITT | 32], CRC_[BIG | LITTLE]_ENDIAN, CRC_BIT_ORDER_[STANDARD | NON_STANDARD], CRC_FINAL_REVERSE_[STANDARD | NON_STANDARD], and CRC_FINAL_XOR_[STANDARD | NON_STANDARD]

17.5.2.2 SYS_CRC_32INITVALUE

```
#define SYS_CRC_32INITVALUE Sys_CRC_32InitValue(CRC)
```

Location: crc.h:173

Macro wrapper for Sys_CRC_32InitValue() Initialize CRC for CRC-32.

Assumptions

Note that D_CRC supports only CRC_CCITT mode, hence no configuration is applied for this instance.

Example Code for SYS_CRC_32INITVALUE

```
    //Initialize the default CRC block for CRC-32
    SYS_CRC_32INITVALUE();
```

17.5.2.3 SYS_CRC_CCITTINITVALUE

```
#define SYS_CRC_CCITTINITVALUE Sys_CRC_CCITTInitValue(CRC)
```

Location: crc.h:181

Macro wrapperr for Sys_CRC_CCITTInitValue() Initialize CRC for CRC-CCITT.

Assumptions

CRC is configured to work in CRC-CCITT mode.

Example Code for SYS_CRC_CCITTINITVALUE

```
//Initialize the default CRC block for CRC-CCITT
SYS_CRC_CCITTINITVALUE();
```

17.5.2.4 SYS_CRC_GETCURRENTVALUE

```
#define SYS_CRC_GETCURRENTVALUE Sys_CRC_GetCurrentValue(CRC)
```

Location: crc.h:188

Macro wrapper for Sys_CRC_GetCurrentValue() Initialize CRC for CRC-CCITT.

Example Code for SYS_CRC_GETCURRENTVALUE

```
// Retrieve current value from the default CRC block
SYS_CRC_GETCURRENTVALUE();
```

17.5.2.5 SYS_CRC_GETFINALVALUE

```
#define SYS_CRC_GETFINALVALUE Sys_CRC_GetFinalValue(CRC)
```

Location: crc.h:200

Macro wrapper for Sys_CRC_GetFinalValue() Initialize final CRC value.

Returns:

CRC final value.

Assumptions

D_I2C only supports CRC-CCITT mode. Use Sys_CRC_GetCurrentValue instead. Returns initial value of CRC if D_CRC is passed or any other unknown instance.

Example Code for SYS_CRC_GETFINALVALUE

```
    // Retrieve final value from the default CRC block
    SYS_CRC_GETFINALVALUE();
```

17.5.2.6 SYS_CRC_ADD

```
#define SYS_CRC_ADD Sys_CRC_Add(CRC, (data), (size))
```

Location: crc.h:209

Macro wrapper for Sys_CRC_Add() Add data to the current CRC calculation, based on size.

Example Code for SYS_CRC_ADD

```
    // Add 8 bits of data to the default CRC block
    SYS_CRC_ADD(0xF1, 8);
```

Parameters

Direction	Name	Description
in	*data*	Data to add
in	*size*	Size of data to add, 1, 8, 16, 24, 32 are valid.

17.5.3 Cyclic Redundancy Check Function Documentation

17.5.3.1 Sys_Set_CRC_Config

```
void Sys_Set_CRC_Config(CRC_Type * crc, uint32_t config)
```

Location: crc.h:52

Configure the CRC generator type, endianness of the input data, and standard vs non-standard CRC behavior.

Assumptions

Note that D_CRC supports only CRC_CCITT mode, hence no configuration is applied for this instance.

Example Code for Sys_Set_CRC_Config

```
    // Configure CRC block to CRC-32 (IEEE 802.3) algorithm,
    // using little endian and non-standard (opposite) bit order
    Sys_Set_CRC_Config(CRC, CRC_32 | CRC_LITTLE_ENDIAN |
                                CRC_BIT_ORDER_NON_STANDARD);
```

Parameters

Direction	Name	Description
in	*crc*	Pointer to the CRC instance
in	*config*	CRC generator configuration; use CRC_[CCITT | 32], CRC_[BIG | LITTLE]_ENDIAN, CRC_BIT_ORDER_[STANDARD | NON_STANDARD], CRC_FINAL_REVERSE_[STANDARD | NON_STANDARD], and CRC_FINAL_XOR_[STANDARD | NON_STANDARD]

17.5.3.2 Sys_CRC_32InitValue

```
void Sys_CRC_32InitValue(CRC_Type * crc)
```

Location: crc.h:68

Initialize CRC for CRC-32.

Assumptions

Note that D_CRC supports only CRC_CCITT mode, hence no configuration is applied for this instance.

Example Code for Sys_CRC_32InitValue

```
    // Initialize the CRC block for CRC-32
    Sys_CRC_32InitValue(CRC);
```

Parameters

Direction	Name	Description
in	*crc*	Pointer to the CRC instance

17.5.3.3 Sys_CRC_CCITTInitValue

```
void Sys_CRC_CCITTInitValue(CRC_Type * crc)
```

Location: crc.h:83

Initialize CRC for CRC-CCITT.

Assumptions

CRC is configured to work in CRC-CCITT mode.

Example Code for Sys_CRC_CCITTInitValue

```
    // Initialize the CRC block for CRC-CCITT
    Sys_CRC_CCITTInitValue(CRC);
```

Parameters

Direction	Name	Description
in	*crc*	Pointer to the CRC instance

17.5.3.4 Sys_CRC_GetCurrentValue

```
uint32_t Sys_CRC_GetCurrentValue(const CRC_Type * crc)
```

Location: crc.h:95

Retrieve current value from CRC.

Returns:

Current CRC value.

Example Code for Sys_CRC_GetCurrentValue

```
    // Retrieve current value from the CRC block
    value = Sys_CRC_GetCurrentValue(CRC);
```

Parameters

Direction	Name	Description
in	*crc*	Pointer to the CRC instance

17.5.3.5 Sys_CRC_GetFinalValue

```
uint32_t Sys_CRC_GetFinalValue(const CRC_Type * crc)
```

Location: crc.h:111

Initialize final CRC value.

Returns:

CRC final value.

Assumptions

D_CRC only supports CRC-CCITT mode. Use Sys_CRC_GetCurrentValue instead. Returns initial value of CRC if D_CRC is passed or any other unknown instance.

Example Code for Sys_CRC_GetFinalValue

```
    // Retrieve final value from the CRC block
    Sys_CRC_GetFinalValue(CRC);
```

Parameters

Direction	Name	Description
in	*crc*	Pointer to the CRC instance

17.5.3.6 Sys_CRC_Add

```
void Sys_CRC_Add(CRC_Type * crc, uint32_t data, uint32_t size)
```

Location: crc.h:138

Add data to the current CRC calculation, based on size.

Example Code for Sys_CRC_Add

```
    // Add 8 bits of data to the current CRC block
    Sys_CRC_Add(CRC, 0xF1, 8);
```

Parameters

Direction	Name	Description
in	*crc*	Pointer to the CRC instance
in	*data*	Data to add
in	*size*	Size of data to add, 1, 8, 16, 24, 32 are valid.

17.6 Direct Memory Access

Direct Memory Access (DMA) hardware abstraction layer.

17.6.1 Summary

Macros

- SYS_DMA_CHANNELCONFIG : Macro wrapper for Sys_DMA_ChannelConfig() Configure the DMA channels for a data transfer.
- SYS_DMA_MODE_ENABLE : Macro wrapper for Sys_DMA_Mode_Enable() Configure the DMA channels for a data transfer.

Functions

- Sys_DMA_ChannelConfig : Configure the DMA channels for a data transfer.
- Sys_DMA_Mode_Enable : Configure the DMA channels for a data transfer.
- Sys_DMA_Get_Status : Get the status register of the DMA instance.
- Sys_DMA_Clear_Status : Writes to the CNT_INT_CLEAR, COMPLETE_INT_CLEAR, or SRC_BUFFER_FILL_LVL_WR.
- Sys_DMA_Set_Ctrl : Sets the DMA_CTRL of the DMA instance.

17.6.2 Direct Memory Access Macro Definition Documentation

17.6.2.1 SYS_DMA_CHANNELCONFIG

```
 #define SYS_DMA_CHANNELCONFIG Sys_DMA_ChannelConfig(DMA, cfg, \
                                                              transferlength, counterInt,
srcAddr, \
                                                              destAddr)
```

Location: dma.h:174

Macro wrapper for Sys_DMA_ChannelConfig() Configure the DMA channels for a data transfer.

Example Code for SYS_DMA_CHANNELCONFIG

```
    // Configure the default DMA channels for data transfer using a transfer length of
 4
    // and interrupting at the beginning of the transfer
    SYS_DMA_CHANNELCONFIG(DMA_BIG_ENDIAN | DEST_TRANS_LENGTH_SEL |
                          DMA_PRIORITY_1 | DMA_CNT_INT_ENABLE, 4, 0,
                          (uint32_t)0x2001ffc8, (uint32_t)0x2001ffb8);
```

Parameters

Direction	*Name*	Description
in	*cfg*	Configuration of the DMA transfer behavior; use DMA_[LITTLE | BIG]_ENDIAN, [DEST | SRC]_TRANS_LENGTH_SEL, DMA_PRIORITY_[0 | 1 | 2 | 3], DMA_SRC_* DMA_DEST_* WORD_SIZE_*, DMA_SRC_ADDR_*, DMA_DEST_ADDR_*, DMA_SRC_ADDR_LSB_TOGGLE_[DISABLE | ENABLE], DMA_CNT_INT_[DISABLE | ENABLE], DMA_COMPLETE_INT_[DISABLE | ENABLE]
in	*transferlength*	Configuration of the DMA transfer length
in	*counterInt*	Configuration of when the counter interrupt will occur during the transfer
in	*srcAddr*	Base source address for the DMA transfer
in	*destAddr*	Base destination address for the DMA transfer

17.6.2.2 SYS_DMA_MODE_ENABLE

```
 #define SYS_DMA_MODE_ENABLE Sys_DMA_Mode_Enable(DMA, (mode))
```

Location: dma.h:189

Macro wrapper for Sys_DMA_Mode_Enable() Configure the DMA channels for a data transfer.

Example Code for SYS_DMA_MODE_ENABLE

```
    // Enable the default DMA block
    SYS_DMA_MODE_ENABLE(DMA_ENABLE);
```

Parameters

Direction	*Name*	Description
in	*mode*	Enable mode of operation of the DMA Channel; use DMA_[DISABLE | ENABLE | DMA_ENABLE_WRAP | DMA_ENABLE_WRAP_RESTART | DMA_TRIGGER DMA_TRIGGER_WRAP | DMA_TRIGGER_WRAP_RESTART]

17.6.3 Direct Memory Access Function Documentation

17.6.3.1 Sys_DMA_ChannelConfig

```
 void Sys_DMA_ChannelConfig(DMA_Type * dma, uint32_t cfg, uint32_t
transferLength, uint32_t counterInt, uint32_t srcAddr, uint32_t destAddr)
```

Location: dma.h:62

Configure the DMA channels for a data transfer.

Example Code for Sys_DMA_ChannelConfig

```
    // Configure the DMA1 channels for data transfer using a transfer length of 4
    // and interrupting at the beginning of the transfer
    uint32_t srcAddr = (uint32_t)0x2001ffc8;
    uint32_t destAddr = (uint32_t)0x2001ffb8;
    Sys_DMA_ChannelConfig(DMA1, DMA_BIG_ENDIAN | DEST_TRANS_LENGTH_SEL |
                          DMA_PRIORITY_1 | DMA_CNT_INT_ENABLE, 4, 0,
                          srcAddr, destAddr);
```

Parameters

Direction	Name	Description
`in`	*dma*	Pointer to the DMA instance
`in`	*cfg*	Configuration of the DMA transfer behavior; use DMA_[LITTLE | BIG]_ENDIAN, [DEST | SRC]_TRANS_LENGTH_SEL, DMA_PRIORITY_[0 | 1 | 2 | 3], DMA_SRC_* DMA_DEST_* WORD_SIZE_*, DMA_SRC_ADDR_*, DMA_DEST_ADDR_*, DMA_SRC_ADDR_LSB_TOGGLE_[DISABLE | ENABLE], DMA_CNT_INT_[DISABLE | ENABLE], DMA_COMPLETE_INT_[DISABLE | ENABLE]
`in`	*transferLength*	Configuration of the DMA transfer length
`in`	*counterInt*	Configuration of when the counter interrupt will occur during the transfer
`in`	*srcAddr*	Base source address for the DMA transfer
`in`	*destAddr*	Base destination address for the DMA transfer

17.6.3.2 Sys_DMA_Mode_Enable

```
void Sys_DMA_Mode_Enable(DMA_Type * dma, uint32_t mode)
```

Location: dma.h:107

Configure the DMA channels for a data transfer.

Example Code for Sys_DMA_Mode_Enable

```
    // Enable DMA1
    Sys_DMA_Mode_Enable(DMA1, DMA_ENABLE);
```

Parameters

Direction	Name	Description
in	*dma*	Pointer to the DMA instance
in	*mode*	Enable mode of operation of the DMA Channel; use DMA_[DISABLE | ENABLE | DMA_ENABLE_WRAP | DMA_ENABLE_WRAP_RESTART | DMA_TRIGGER DMA_TRIGGER_WRAP | DMA_TRIGGER_WRAP_RESTART]

17.6.3.3 Sys_DMA_Get_Status

```
uint32_t Sys_DMA_Get_Status(DMA_Type * dma)
```

Location: dma.h:119

Get the status register of the DMA instance.

Returns:

The DMA_STATUS of the DMA instance.

Parameters

Direction	Name	Description
in	*dma*	Pointer to the DMA instance

17.6.3.4 Sys_DMA_Clear_Status

```
void Sys_DMA_Clear_Status(DMA_Type * dma, uint32_t ctrl)
```

Location: dma.h:132

Writes to the CNT_INT_CLEAR, COMPLETE_INT_CLEAR, or SRC_BUFFER_FILL_LVL_WR.

Returns:

The DMA_STATUS of the DMA instance

Parameters

Direction	Name	Description
`in`	*dma*	Pointer to the DMA instance
`in`	*ctrl*	Data to be written to the DMA_SATUS register

17.6.3.5 Sys_DMA_Set_Ctrl

```
void Sys_DMA_Set_Ctrl(DMA_Type * dma, uint32_t ctrl)
```

Location: dma.h:143

Sets the DMA_CTRL of the DMA instance.

Parameters

Direction	Name	Description
`in`	*dma*	Pointer to the DMA instance
`in`	*ctrl*	Data to be written to the DMA_CTRL register

17.7 Flash Copier

Flash copier hardware abstraction layer.

17.7.1 Summary

Functions

- Sys_Flash_Copy : Copy data from the flash memory to a RAM memory instance.
- Sys_Flash_Compare : Compare data in the flash to a pre-specified value.
- Sys_Flash_CalculateCRC : Calculate CRC of words in flash using flash copier.

17.7.2 Flash Copier Function Documentation

17.7.2.1 Sys_Flash_Copy

```
 void Sys_Flash_Copy(FLASH_Type * flash, uint32_t src_addr, uint32_t dest_addr,
uint32_t length, uint32_t cpy_dest)
```

Location: flash_copier.h:54

Copy data from the flash memory to a RAM memory instance.

Assumptions

src_addr points to an address in flash memory dest_addr points to an address in RAM memory The flash copy does not need to be complete before returning If CRC is selected as the destination, dest_addr is ignored and 32-bit copy mode is selected automatically.

Parameters

Direction	Name	Description
`in`	*flash*	Pointer to the flash instance
`in`	*src_addr*	Source address in flash to copy data from
`in`	*dest_addr*	Destination address in RAM to copy data to
`in`	*length*	Number of words to copy
`in`	*cpy_dest*	Destination copier is CRC or memories; use COPY_TO_[CRC | MEM], and COPY_TO_[32 | 40]_BIT

17.7.2.2 Sys_Flash_Compare

```
 uint32_t Sys_Flash_Compare(FLASH_Type * flash, uint32_t cfg, uint32_t addr,
uint32_t length, uint32_t value, uint32_t value_ecc)
```

Location: flash_copier.h:74

Compare data in the flash to a pre-specified value.

Returns:

0 if comparison succeeded, 1 if the comparison failed.

Assumptions

addr points to an address in flash memory

Parameters

Direction	Name	Description
`in`	*flash*	Pointer to the flash instance
`in`	*cfg*	Flash comparator configuration; use COMP_MODE_[CONSTANT | CHBK]_BYTE, COMP_ADDR_[DOWN | UP]_BYTE, and COMP_ADDR_STEP_*_BYTE
`in`	*addr*	Base address of the area to verify
`in`	*length*	Number of words to verify
`in`	*value*	Value that the words read from flash will be compared against
`in`	*value_ecc*	Value that the error-correction coding bits from the extended words read from flash will be compared against

17.7.2.3 Sys_Flash_CalculateCRC

```
uint32_t Sys_Flash_CalculateCRC(FLASH_Type * flash, uint32_t addr, uint32_t
length, uint32_t * crc)
```

Location: flash_copier.h:88

Calculate CRC of words in flash using flash copier.

Returns:

0 if calculation has succeeded, 1 if the calculation has failed.

NOTE: Flash copier and CRC register are modified

Parameters

Direction	Name	Description
`in`	*flash*	Flash instance being used
`in`	*addr*	Address belonging to the flash instance
`in`	*length*	Total number of words used in CRC calculation
`out`	*crc*	CRC value calculated using CRC peripheral

17.8 General-Purpose I/O Interface

General-Purpose I/O (GPIO) Interface hardware abstraction layer.

17.8.1 Summary

Macros

- GPIO_LEVEL1_DRIVE : 1st level GPIO drive strength
- GPIO_LEVEL2_DRIVE : 2nd level GPIO drive strength
- GPIO_LEVEL3_DRIVE : 3rd level GPIO drive strength
- GPIO_LEVEL4_DRIVE : 4th level GPIO drive strength
- SYS_GPIO_CONFIG : Configure the specified digital I/O.

Functions

- Sys_GPIO_NMIConfig : Configure a source for NMI input selection.
- Sys_GPIO_IntConfig : Configure a GPIO interrupt source.
- Sys_GPIO_CM33JTAGConfig : Configure Arm Cortex-M3 SWJ-DP.
- Sys_GPIO_Set_High : Set the specified GPIO output value to high.
- Sys_GPIO_Set_Low : Set the specified GPIO output value to low.
- Sys_GPIO_Toggle : Toggle the current value of the specified GPIO output.
- Sys_GPIO_Read : Read the specified GPIO value.
- Sys_GPIO_Write : Write the specified GPIO value.
- Sys_GPIO_Set_Direction : Set the input/output direction for any GPIOs configured as GPIOs.

17.8.2 General-Purpose I/O Interface Macro Definition Documentation

17.8.2.1 GPIO_LEVEL1_DRIVE

```
#define GPIO_LEVEL1_DRIVE GPIO_2X_DRIVE
```

Location: gpio.h:45

1st level GPIO drive strength

17.8.2.2 GPIO_LEVEL2_DRIVE

```
#define GPIO_LEVEL2_DRIVE GPIO_3X_DRIVE
```

Location: gpio.h:48

2nd level GPIO drive strength

17.8.2.3 GPIO_LEVEL3_DRIVE

```
#define GPIO_LEVEL3_DRIVE GPIO_5X_DRIVE
```

Location: gpio.h:51

3rd level GPIO drive strength

17.8.2.4 GPIO_LEVEL4_DRIVE

```
#define GPIO_LEVEL4_DRIVE GPIO_6X_DRIVE
```

Location: gpio.h:54

4th level GPIO drive strength

17.8.2.5 SYS_GPIO_CONFIG

```
#define SYS_GPIO_CONFIG SYS_ASSERT(pad < GPIO_PAD_COUNT); \
    GPIO->CFG[(pad)] = (config)
```

Location: gpio.h:69

Configure the specified digital I/O.

Example Code for SYS_GPIO_CONFIG

```
    // Enable GPIO 5 as GPIO input using no pull-up resistor
    SYS_GPIO_CONFIG(GPIO5, GPIO_MODE_GPIO_IN | GPIO_NO_PULL);
```

Parameters

Direction	*Name*	Description
in	*pad*	Digital I/O pad to configure; use a constant
in	*config*	I/O configuration; use GPIO_*X_DRIVE, GPIO_LPF_[ENABLE | DISABLE], GPIO_*_PULL, and GPIO_MODE_*

17.8.3 General-Purpose I/O Interface Function Documentation

17.8.3.1 Sys_GPIO_NMIConfig

```
void Sys_GPIO_NMIConfig(uint32_t config, uint32_t source, uint32_t polarity)
```

Location: gpio.h:80

Configure a source for NMI input selection.

Example Code for Sys_GPIO_NMIConfig

```
    // Configure NMI input using GPIO 13 as active-low source without low-pass filter
    Sys_GPIO_NMIConfig(GPIO_LPF_ENABLE, NMI_SRC_GPIO_13, NMI_ACTIVE_LOW);
```

Parameters

Direction	Name	Description
in	*config*	GPIO pin configuration if NMI is pad
in	*source*	NMI source; use NMI_SRC_*
in	*polarity*	NMI polarity; use NMI_ACTIVE_[LOW | HIGH]

17.8.3.2 Sys_GPIO_IntConfig

```
 void Sys_GPIO_IntConfig(uint32_t index, uint32_t config, uint32_t dbnc_clk,
uint32_t dbnc_cnt)
```

Location: gpio.h:105

Configure a GPIO interrupt source.

Example Code for Sys_GPIO_IntConfig

```
    // configure GPIO interrupt channel 7 to use GPIO 11 as interrupt source, with
    // 1 debounce filter count.
    Sys_GPIO_IntConfig(7, (GPIO_INT_DEBOUNCE_ENABLE | GPIO_INT_SRC_GPIO_11),
                       GPIO_DEBOUNCE_SLOWCLK_DIV1024, 1);
```

Parameters

Direction	Name	Description
in	*index*	GPIO interrupt source to configure; use a constant
in	*config*	GPIO interrupt configuration; use GPIO__INT_DEBOUNCE_[DISABLE | ENABLE], GPIO_INT_SRC_*, and GPIO_INT_EVENT_[NONE | HIGH_LEVEL | LOW_LEVEL | RISING_EDGE | FALLING_EDGE | TRANSITION]
in	*dbnc_clk*	Interrupt button debounce filter clock; use GPIO_DEBOUNCE_SLOWCLK_DIV[32 | 1024]
in	*dbnc_cnt*	Interrupt button debounce filter count

17.8.3.3 Sys_GPIO_CM33JTAGConfig

```
void Sys_GPIO_CM33JTAGConfig(uint32_t config, uint8_t mode)
```

Location: gpio.h:130

Configure Arm Cortex-M3 SWJ-DP.

Example Code for Sys_GPIO_CM33JTAGConfig

```
    // Enable Arm Cortex-M3 SWJ-DP port and JTAG
    Sys_GPIO_CM33JTAGConfig(true, true);
```

Parameters

Direction	Name	Description
in	*config*	JTAG pad configuration; use JTMS_[NO_PULL | WEAK_PULL_UP | WEAK_PULL_DOWN | STRONG_PULL_UP], JTMS_[2X | 3X | 5X | 6X]_DRIVE, JTCK_[NO_PULL | WEAK_PULL_UP | WEAK_PULL_DOWN | STRONG_PULL_UP], JTMS_LPF_[DISABLED | ENABLED], and JTCK_LPF_[DISABLED | ENABLED]
in	*mode*	Enable one of the two JTAG modes or SW mode; use 0 for SW mode, 1 for JTAG with reset enabled, and 2 for JTAG with reset disabled

17.8.3.4 Sys_GPIO_Set_High

```
void Sys_GPIO_Set_High(uint32_t pad)
```

Location: gpio.h:159

Set the specified GPIO output value to high.

Example Code for Sys_GPIO_Set_High

```
    // Set GPIO2 high
    Sys_GPIO_Set_High(GPIO2);
```

Parameters

Direction	Name	Description
in	*pad*	GPIO pad to set high

17.8.3.5 Sys_GPIO_Set_Low

```
void Sys_GPIO_Set_Low(uint32_t pad)
```

Location: gpio.h:170

Set the specified GPIO output value to low.

Example Code for Sys_GPIO_Set_Low

```
    // Set GPIO2 low
    Sys_GPIO_Set_Low(GPIO2);
```

Parameters

Direction	Name	Description
in	*pad*	GPIO pad to set low

17.8.3.6 Sys_GPIO_Toggle

```
void Sys_GPIO_Toggle(uint32_t pad)
```

Location: gpio.h:181

Toggle the current value of the specified GPIO output.

Example Code for Sys_GPIO_Toggle

```
    // Toggle the state of the GPIO2 pin
    Sys_GPIO_Toggle(GPIO2);
```

Parameters

Direction	Name	Description
`in`	*pad*	GPIO pad to toggle

17.8.3.7 Sys_GPIO_Read

```
uint32_t Sys_GPIO_Read(uint32_t pad)
```

Location: gpio.h:193

Read the specified GPIO value.

Returns:

uint32_t pin value; 0 or 1

Example Code for Sys_GPIO_Read

```
    // Read the current value of the GPIO2 pin
    Sys_GPIO_Read(GPIO2);
```

Parameters

Direction	Name	Description
in	*pad*	GPIO pad to set low

17.8.3.8 Sys_GPIO_Write

```
void Sys_GPIO_Write(uint32_t pad, bool value)
```

Location: gpio.h:205

Write the specified GPIO value.

Example Code for Sys_GPIO_Write

```
    // Write a 1 to GPIO2
    Sys_GPIO_Write(GPIO2, 1);
```

Parameters

Direction	Name	Description
in	*pad*	GPIO pad to write value
in	*value*	1 or 0 written to GPIO

17.8.3.9 Sys_GPIO_Set_Direction

```
void Sys_GPIO_Set_Direction(uint32_t dir)
```

Location: gpio.h:220

Set the input/output direction for any GPIOs configured as GPIOs.

Example Code for Sys_GPIO_Set_Direction

```
    // Set GPIO2 as an output
    Sys_GPIO_Set_Direction(GPIO2_OUTPUT);
```

Parameters

Direction	Name	Description
in	*dir*	Input/output configuration settings for any GPIOs from 0 to 15 that are configured as GPIO pads; use GPIO*_INPUT, and GPIO*_OUTPUT

17.9 I2C

Inter-Integrated Circuit (I2C) hardware abstraction layer.

17.9.1 Summary

Macros

- I2C_CONFIG_MASK : Mask for the I2C_CFG register.
- I2C_PADS_NUM : Number of pads used for the I2C interface, for a single instance.
- SYS_I2C_GPIOCONFIG : Macro wrapper for Sys_I2C_GPIOConfig() Configure two GPIOs for the specified I2C interface.
- SYS_I2C_CONFIG : Macro wrapper for Sys_I2C_Config() Apply I2C Master mode related configuration.
- SYS_I2C_STARTREAD : Macro wrapper for Sys_I2C_StartRead() Send slave address on the bus with a read request.
- SYS_I2C_STARTWRITE : Macro wrapper for Sys_I2C_StartWrite() Send slave address on the bus with a write request.
- SYS_I2C_ACK : Macro wrapper for Sys_I2C_ACK() Issue an ACK on the I2C interface.
- SYS_I2C_NACK : Macro wrapper for Sys_I2C_NACK() Issue a NACK on the I2C interface.
- SYS_I2C_LASTDATA : Macro wrapper for Sys_I2C_LastData() Indicate that the current data is the last byte.
- SYS_I2C_RESET : Macro wrapper for Sys_I2C_Reset() Reset the I2C interface.
- SYS_I2C_NACKANDSTOP : Macro wrapper for Sys_I2C_NackAndStop() Issue a NACK followed by a Stop condition on I2C bus.

Functions

- Sys_I2C_GPIOConfig : Configure two GPIOs for the specified I2C interface.
- Sys_I2C_Config : Apply I2C Master mode related configuration.
- Sys_I2C_StartRead : Send slave address on the bus with a read request.
- Sys_I2C_StartWrite : Send slave address on the bus with a write request.

- Sys_I2C_ACK : Issue a ACK on the I2C interface.
- Sys_I2C_NACK : Issue a NACK on the I2C interface.
- Sys_I2C_LastData : Indicate that the current data is the last byte.
- Sys_I2C_Reset : Reset the I2C interface.
- Sys_I2C_NackAndStop : Issue a NACK followed by a Stop condition on I2C bus.

17.9.2 I2C Macro Definition Documentation

17.9.2.1 I2C_CONFIG_MASK

```
#define I2C_CONFIG_MASK (((uint32_t)1U << I2C_CFG_CONNECT_IN_STANDBY_Pos) | \
    ((uint32_t)1U << I2C_CFG_TX_DMA_ENABLE_Pos) | \
    ((uint32_t)1U << I2C_CFG_RX_DMA_ENABLE_Pos) | \
    ((uint32_t)1U << I2C_CFG_TX_INT_ENABLE_Pos) | \
    ((uint32_t)1U << I2C_CFG_RX_INT_ENABLE_Pos) | \
    ((uint32_t)1U << I2C_CFG_BUS_ERROR_INT_ENABLE_Pos) | \
    (1U << I2C_CFG_OVERRUN_INT_ENABLE_Pos) | \
    (1U << I2C_CFG_STOP_INT_ENABLE_Pos) | \
    (1U << I2C_CFG_AUTO_ACK_ENABLE_Pos) | \
    I2C_CFG_SLAVE_PRESCALE_Mask | \
    I2C_CFG_MASTER_PRESCALE_Mask | \
    I2C_CFG_SLAVE_ADDRESS_Mask | \
    (1U << I2C_CFG_SLAVE_Pos))
```

Location: i2c.h:41

Mask for the I2C_CFG register.

17.9.2.2 I2C_PADS_NUM

```
#define I2C_PADS_NUM 2U
```

Location: i2c.h:57

Number of pads used for the I2C interface, for a single instance.

17.9.2.3 SYS_I2C_GPIOCONFIG

```
#define SYS_I2C_GPIOCONFIG Sys_I2C_GPIOConfig(I2C, (config), (scl), (sda))
```

Location: i2c.h:206

Macro wrapper for Sys_I2C_GPIOConfig() Configure two GPIOs for the specified I2C interface.

Example Code for SYS_I2C_GPIOCONFIG

```
    // Configure GPIO3 and GPIO4 as SCL and SDA, enable 1 kOhm pull-up resistors,
    // and disable low-pass filter for default I2C interface
    SYS_I2C_GPIOCONFIG((GPIO_LPF_DISABLE | GPIO_1K_PULL_UP),
                       GPIO3, GPIO4);
```

Parameters

Direction	*Name*	Description
in	*config*	GPIO pin configuration for the I2C pads
in	*scl*	GPIO to use as the I2C transmit pad; use an integer
in	*sda*	GPIO to use as the I2C receive pad; use an integer

17.9.2.4 SYS_I2C_CONFIG

```
#define SYS_I2C_CONFIG Sys_I2C_Config(I2C, (config))
```

Location: i2c.h:229

Macro wrapper for Sys_I2C_Config() Apply I2C Master mode related configuration.

Example Code for SYS_I2C_CONFIG

```
    // Apply I2C Master mode related configuration for default I2C interface
    // Set up prescaler, auto acknowledge, interrupt, and slave enable
    SYS_I2C_CONFIG((I2C_SLAVE_PRESCALE_4      |
                    I2C_AUTO_ACK_ENABLE       |
                    I2C_RX_INT_ENABLE         |
                    I2C_TX_INT_ENABLE         |
                    I2C_STOP_INT_ENABLE       |
                    I2C_OVERRUN_INT_ENABLE    |
                    I2C_BUS_ERROR_INT_ENABLE |
                    I2C_SLAVE_ENABLE |
                    (64 << I2C_CFG_SLAVE_ADDRESS_Pos)));
```

Parameters

Direction	*Name*	Description
in	*config*	I2C configurations for master mode; use I2C_[CONNECT | DISCONNECT]_IN_STANDBY, I2C_TX_DMA_[ENABLE | DISABLE] I2C_RX_DMA_[ENABLE | DISABLE] I2C_TX_INT_[ENABLE | DISABLE] I2C_RX_INT_[ENABLE | DISABLE] I2C_BUS_ERROR_INT_[ENABLE | DISABLE] I2C_OVERRUN_INT_[ENABLE | DISABLE] I2C_STOP_INT_[ENABLE | DISABLE] I2C_AUTO_ACK_[ENABLE | DISABLE] I2C_MASTER_PRESCALE_*, I2C_SLAVE_PRESCALE_*, a slave address constant shifted to I2C_CTRL0_SLAVE_ADDRESS_Pos, I2C_SLAVE_[ENABLE | DISABLE]

17.9.2.5 SYS_I2C_STARTREAD

```
#define SYS_I2C_STARTREAD Sys_I2C_StartRead(I2C, (addr))
```

Location: i2c.h:237

Macro wrapper for Sys_I2C_StartRead() Send slave address on the bus with a read request.

Example Code for SYS_I2C_STARTREAD

```
    // Send slave address 64 on the default I2C bus with a read request
    SYS_I2C_STARTREAD(64);
```

Parameters

Direction	*Name*	Description
in	*addr*	I2C address to use for write transaction

17.9.2.6 SYS_I2C_STARTWRITE

```
#define SYS_I2C_STARTWRITE Sys_I2C_StartWrite(I2C, (addr))
```

Location: i2c.h:245

Macro wrapper for Sys_I2C_StartWrite() Send slave address on the bus with a write request.

Example Code for SYS_I2C_STARTWRITE

```
    // Send slave address 64 on the default I2C bus with a write request
    SYS_I2C_STARTWRITE(64);
```

Parameters

Direction	*Name*	Description
in	*addr*	I2C address to use for write transaction

17.9.2.7 SYS_I2C_ACK

```
#define SYS_I2C_ACK Sys_I2C_ACK(I2C)
```

Location: i2c.h:252

Macro wrapper for Sys_I2C_ACK() Issue an ACK on the I2C interface.

Example Code for SYS_I2C_ACK

```
    // Issue a ACK on the default I2C interface
    SYS_I2C_ACK();
```

17.9.2.8 SYS_I2C_NACK

```
#define SYS_I2C_NACK Sys_I2C_NACK(I2C)
```

Location: i2c.h:259

Macro wrapper for Sys_I2C_NACK() Issue a NACK on the I2C interface.

Example Code for SYS_I2C_NACK

```
// Issue a NACK on the default I2C interface
SYS_I2C_NACK();
```

17.9.2.9 SYS_I2C_LASTDATA

```
#define SYS_I2C_LASTDATA Sys_I2C_LastData(I2C)
```

Location: i2c.h:266

Macro wrapper for Sys_I2C_LastData() Indicate that the current data is the last byte.

Example Code for SYS_I2C_LASTDATA

```
//Indicate that the current data is the last byte
SYS_I2C_LASTDATA();
```

17.9.2.10 SYS_I2C_RESET

```
#define SYS_I2C_RESET Sys_I2C_Reset(I2C)
```

Location: i2c.h:273

Macro wrapper for Sys_I2C_Reset() Reset the I2C interface.

Example Code for SYS_I2C_RESET

```
// Reset the default I2C interface
SYS_I2C_RESET();
```

17.9.2.11 SYS_I2C_NACKANDSTOP

```
#define SYS_I2C_NACKANDSTOP Sys_I2C_NackAndStop(I2C)
```

Location: i2c.h:280

Macro wrapper for Sys_I2C_NackAndStop() Issue a NACK followed by a Stop condition on I2C bus.

Example Code for SYS_I2C_NACKANDSTOP

```
    // Issue a NACK followed by a Stop condition on default I2C bus
    SYS_I2C_NACKANDSTOP();
```

17.9.3 I2C Function Documentation

17.9.3.1 Sys_I2C_GPIOConfig

```
void Sys_I2C_GPIOConfig(const I2C_Type * i2c, uint32_t config, uint32_t scl,
uint32_t sda)
```

Location: i2c.h:68

Configure two GPIOs for the specified I2C interface.

Example Code for Sys_I2C_GPIOConfig

```
    // Configure GPIO3 and GPIO4 as SCL and SDA, enable 1 kOhm pull-up resistors,
    // and disable low-pass filter for I2C interface
    Sys_I2C_GPIOConfig(I2C, (GPIO_LPF_DISABLE | GPIO_1K_PULL_UP),
                       GPIO3, GPIO4);
```

Parameters

Direction	Name	Description
in	*i2c*	Pointer to the I2C instance
in	*config*	GPIO pin configuration for the I2C pads
in	*scl*	GPIO to use as the I2C transmit pad; use an integer
in	*sda*	GPIO to use as the I2C receive pad; use an integer

17.9.3.2 Sys_I2C_Config

```
void Sys_I2C_Config(I2C_Type * i2c, uint32_t config)
```

Location: i2c.h:110

Apply I2C Master mode related configuration.

Example Code for Sys_I2C_Config

```
    // Apply I2C Master mode related configuration for I2C interface
    // Set up prescaler, auto acknowledge, interrupt, and slave enable
    Sys_I2C_Config(I2C, (I2C_SLAVE_PRESCALE_4 |
                   I2C_AUTO_ACK_ENABLE        |
                   I2C_RX_INT_ENABLE          |
                   I2C_TX_INT_ENABLE          |
                   I2C_STOP_INT_ENABLE        |
                   I2C_OVERRUN_INT_ENABLE     |
                   I2C_BUS_ERROR_INT_ENABLE   |
                   I2C_SLAVE_ENABLE           |
                   (64 << I2C_CFG_SLAVE_ADDRESS_Pos)));
```

Parameters

Direction	Name	Description
`in`	*i2c*	Pointer to the I2C instance
`in`	*config*	I2C configurations for master mode; use I2C_[CONNECT | DISCONNECT]_IN_STANDBY, I2C_TX_DMA_[ENABLE | DISABLE] I2C_RX_DMA_[ENABLE | DISABLE] I2C_TX_INT_[ENABLE | DISABLE] I2C_RX_INT_[ENABLE | DISABLE] I2C_BUS_ERROR_INT_[ENABLE | DISABLE] I2C_OVERRUN_INT_[ENABLE | DISABLE] I2C_STOP_INT_[ENABLE | DISABLE] I2C_AUTO_ACK_[ENABLE | DISABLE] I2C_MASTER_PRESCALE_*, I2C_SLAVE_PRESCALE_*, a slave address constant shifted to I2C_CTRL0_SLAVE_ADDRESS_Pos, I2C_SLAVE_[ENABLE | DISABLE]

17.9.3.3 Sys_I2C_StartRead

```
void Sys_I2C_StartRead(I2C_Type * i2c, uint32_t addr)
```

Location: i2c.h:122

Send slave address on the bus with a read request.

Example Code for Sys_I2C_StartRead

```
    // Send slave address 64 on the I2C bus with a read request
    Sys_I2C_StartRead(I2C, 64);
```

Parameters

Direction	Name	Description
`in`	*i2c*	Pointer to the I2C instance
`in`	*addr*	I2C address to use for write transaction

17.9.3.4 Sys_I2C_StartWrite

```
void Sys_I2C_StartWrite(I2C_Type * i2c, uint32_t addr)
```

Location: i2c.h:136

Send slave address on the bus with a write request.

Example Code for Sys_I2C_StartWrite

```
    // Send slave address 64 on the I2C bus with a write request
    Sys_I2C_StartWrite(I2C, 64);
```

Parameters

Direction	Name	Description
in	*i2c*	I2C instance number
in	*addr*	I2C address to use for write transaction

17.9.3.5 Sys_I2C_ACK

```
void Sys_I2C_ACK(I2C_Type * i2c)
```

Location: i2c.h:148

Issue a ACK on the I2C interface.

Example Code for Sys_I2C_ACK

```
    // Issue a ACK on the I2C interface
    Sys_I2C_ACK(I2C);
```

Parameters

Direction	Name	Description
in	*i2c*	Pointer to the I2C instance

17.9.3.6 Sys_I2C_NACK

```
void Sys_I2C_NACK(I2C_Type * i2c)
```

Location: i2c.h:159

Issue a NACK on the I2C interface.

Example Code for Sys_I2C_NACK

```
    // Issue a NACK on the I2C interface
    Sys_I2C_NACK(I2C);
```

Parameters

Direction	Name	Description
in	*i2c*	Pointer to the I2C instance

17.9.3.7 Sys_I2C_LastData

```
void Sys_I2C_LastData(I2C_Type * i2c)
```

Location: i2c.h:170

Indicate that the current data is the last byte.

Example Code for Sys_I2C_LastData

```
    //Indicate that the current data is the last byte
    Sys_I2C_LastData(I2C);
```

Parameters

Direction	Name	Description
in	*i2c*	Pointer to the I2C instance

17.9.3.8 Sys_I2C_Reset

```
void Sys_I2C_Reset(I2C_Type * i2c)
```

Location: i2c.h:181

Reset the I2C interface.

Example Code for Sys_I2C_Reset

```
    // Reset the I2C interface
    Sys_I2C_Reset(I2C);
```

Parameters

Direction	Name	Description
in	*i2c*	Pointer to the I2C instance

17.9.3.9 Sys_I2C_NackAndStop

```
void Sys_I2C_NackAndStop(I2C_Type * i2c)
```

Location: i2c.h:192

Issue a NACK followed by a Stop condition on I2C bus.

Example Code for Sys_I2C_NackAndStop

```
    // Issue a NACK followed by a Stop condition on I2C bus
    Sys_I2C_NackAndStop(I2C);
```

Parameters

Direction	Name	Description
in	*i2c*	Pointer to the I2C instance

17.10 LSAD

LSAD hardware abstraction layer.

17.10.1 Summary

Data Structures

- F_LSAD_TRIM : LSAD structure for offset/gain conversion function.

Macros

- LSAD_OFFSET_ERROR_CONV_QUOTIENT : LSAD offset conversion quotient.
- LSAD_GAIN_ERROR_CONV_QUOTIENT : LSAD gain conversion quotient.
- ERROR_LSAD_INPUT_CFG : Error code for failed configuration.
- PRE_SEL_SIZE : Value size of pre-select inputs in bits.

Functions

- Sys_LSAD_Gain_Offset : Convert a gain and offset value from NVR in integer format to float format.
- Sys_LSAD_TempSensor_Gain_Offset : Convert a gain and offset value from NVR in integer format to float format for the temperature sensor.
- Sys_LSAD_InputConfig : Configure LSAD input channel.

17.10.2 LSAD Data Structures Type Documentation

17.10.2.1 F_LSAD_TRIM

Location: lsad.h:53

LSAD structure for offset/gain conversion function.

Data Fields

Type	*Name*	Description
`float`	*lf_offset*	Low frequency LSAD offset.
`float`	*hf_offset*	High frequency LSAD offset.
`float`	*lf_gain*	Low frequency LSAD gain.
`float`	*hf_gain*	High frequency LSAD gain.

17.10.3 LSAD Macro Definition Documentation

17.10.3.1 LSAD_OFFSET_ERROR_CONV_QUOTIENT

```
#define LSAD_OFFSET_ERROR_CONV_QUOTIENT 32768.0f;
```

Location: lsad.h:41

LSAD offset conversion quotient.

17.10.3.2 LSAD_GAIN_ERROR_CONV_QUOTIENT

```
#define LSAD_GAIN_ERROR_CONV_QUOTIENT 65536.0f;
```

Location: lsad.h:44

LSAD gain conversion quotient.

17.10.3.3 ERROR_LSAD_INPUT_CFG

```
#define ERROR_LSAD_INPUT_CFG 0x80
```

Location: lsad.h:47

Error code for failed configuration.

17.10.3.4 PRE_SEL_SIZE

```
#define PRE_SEL_SIZE 4
```

Location: lsad.h:50

Value size of pre-select inputs in bits.

17.10.4 LSAD Function Documentation

17.10.4.1 Sys_LSAD_Gain_Offset

```
void Sys_LSAD_Gain_Offset(const volatile struct LSAD_TRIM * i_gain_offset,
struct F_LSAD_TRIM * f_gain_offset)
```

Location: lsad.h:79

Convert a gain and offset value from NVR in integer format to float format.

Parameters

Direction	Name	Description
in	*i_gain_offset*	Gain and offset error from NVR, raw integer form
out	*f_gain_offset*	Gain and offset error converted to floating point.

17.10.4.2 Sys_LSAD_TempSensor_Gain_Offset

```
void Sys_LSAD_TempSensor_Gain_Offset(const volatile struct TEMP_SENSOR_TRIM * i_
gain_offset, struct F_LSAD_TRIM * f_gain_offset)
```

Location: lsad.h:100

Convert a gain and offset value from NVR in integer format to float format for the temperature sensor.

Parameters

Direction	Name	Description
in	*i_gain_offset*	Gain and offset error from NVR, raw integer form
out	*f_gain_offset*	Gain and offset error converted to floating point.

17.10.4.3 Sys_LSAD_InputConfig

`uint32_t Sys_LSAD_InputConfig(uint32_t num, uint32_t source, uint32_t pos_gpio, uint32_t neg_gpio)`

Location: lsad.h:124

Configure LSAD input channel.

Returns:

Returns success/fail result of configuration. 0 = success.

Example Code for Sys_LSAD_InputConfig

```
// Use LSAD channel 0
// Set LSAD positive source to gpio selection source 1 and negative source to
// ground
// Connect GPIO 7 to the LSAD
// No GPIO used for negative input (-1)
Sys_LSAD_InputConfig(0, LSAD_POS_INPUT_SEL1 | LSAD_NEG_INPUT_GND,
                     GPIO7, -1);
```

Parameters

Direction	Name	Description
in	*num*	Channel number; use an integer
in	*source*	Select ADC channel input source use LSAD_[POS | NEG]_INPUT_[SEL* | AOUT | VBAT | TEMP | GND].
in	*pos_gpio*	Select GPIO to use for positive input(if applicable); use an integer. If no GPIO used, use a negative integer.

Direction	Name	Description
`in`	*neg_gpio*	Select GPIO to use for negative input(if applicable); use an integer. If no GPIO used, use a negative integer.

17.11 Nested Vectored Interrupt Controller

Nested Vectored Interrupt Controller (NVIC) hardware abstraction layer.

17.11.1 Summary

Functions

- Sys_NVIC_DisableAllInt : Disable all external interrupts.
- Sys_NVIC_ClearAllPendingInt : Clear the pending status for all external interrupts.

17.11.2 Nested Vectored Interrupt Controller Function Documentation

17.11.2.1 Sys_NVIC_DisableAllInt

```
__STATIC_INLINE void Sys_NVIC_DisableAllInt()
```

Location: nvic.h:42

Disable all external interrupts.

Example Code for Sys_NVIC_DisableAllInt

```
    // Disable all external interrupts
    Sys_NVIC_DisableAllInt();
```

17.11.2.2 Sys_NVIC_ClearAllPendingInt

```
__STATIC_INLINE void Sys_NVIC_ClearAllPendingInt()
```

Location: nvic.h:53

Clear the pending status for all external interrupts.

Example Code for Sys_NVIC_ClearAllPendingInt

```
    // Clear all pending external interrupts
    Sys_NVIC_ClearAllPendingInt();
```

17.12 POWER

Power Supply Hardware Abstraction Layer.

17.12.1 Summary

Functions

- Sys_Power_RFEnable : RF Power Switch and Isolation.
- Sys_Power_CC312AO_Enable : CryptoCell312AO Power Switch and Isolation.
- Sys_Power_CC312AO_Disable : CryptoCell312AO Power Switch and Isolation.

17.12.2 POWER Function Documentation

17.12.2.1 Sys_Power_RFEnable

```
void Sys_Power_RFEnable()
```

Location: power.h:41

RF Power Switch and Isolation.

Example Code for Sys_Power_RFEnable

```
    // Enable RF power switches, remove RF Isolation
    Sys_Power_RFEnable();
```

17.12.2.2 Sys_Power_CC312AO_Enable

```
void Sys_Power_CC312AO_Enable()
```

Location: power.h:58

CryptoCell312AO Power Switch and Isolation.

Example Code for Sys_Power_CC312AO_Enable

```
    // Enable the CryptoCell block and remove its power isolation
    Sys_Power_CC312AO_Enable();
```

17.12.2.3 Sys_Power_CC312AO_Disable

```
void Sys_Power_CC312AO_Disable()
```

Location: power.h:77

CryptoCell312AO Power Switch and Isolation.

Example Code for Sys_Power_CC312AO_Disable

```
    // Disable and isolate power from the CryptoCell block
    Sys_Power_CC312AO_Disable();
```

17.13 RFFE Radio Frequency Front End

Radio Frequency Front End (RFFE) hardware abstraction layer.

17.13.1 Summary

Macros

- STABILIZATION_DELAY : External include files.
- MEASUREMENT_DELAY : Corresponds to sample rate of the LSAD as configured (625 Hz)
- V_TO_MV : Factor for converting back and forth from mV to V.
- V_TO_MV_F : Float iteration of factor for converting back and forth from mV to V.
- DEF_CHANNEL : Default LSAD channel used to measure voltage rails.
- MAX_LSAD_CHANNEL : Maximum number of LSAD channels in the design.
- VDDPA_EN : VDDPA enable selection.
- VDDPA_DIS : VDDPA disable selection.
- VCC_VDDRF_MARGIN : We strongly recommended having VCC at least 50mV higher than VDDRF.
- TRIM_MARGIN : Trim margin.
- MV_PER_DBM_VDDPA : Estimated voltage increase per 1dBm increased TX power.
- MV_PER_DBM_VDDRF : Estimated voltage increase per 1dBm increased TX power.
- STEPS_PER_DBM_VDDRF : Estimated trim steps per 1dBm increased TX power.
- STEPS_PER_DBM_VDDPA : Estimated trim steps per 1dBm increased TX power.
- RF_MAX_POWER : Maximum RF output power possible.
- RF_MAX_POWER_NO_VDDPA : Maximum RF output power possible without using VDDPA.
- RF_NO_VDDPA_TYPICAL_POWER : Typical RF output power when VDDPA is not used, with default trims.
- RF_DEFAULT_POWER : RF output power used by default.
- RF_MIN_POWER : Minimum possible RF output power.
- PA_PWR_BYTE_0DBM : RF Output Power code for 0dBm.
- PA_ENABLE_BIAS_SETTING : Power amplifier bias enable.
- PA_DISABLE_BIAS_SETTING : Power amplifier bias disable.
- SW_CTRL_DELAY_3_BYTE : Switch control delay.
- RAMPUP_DELAY_3_BYTE : Ramp-up delay.
- DISABLE_DELAY_3_BYTE : Disable delay.
- ERRNO_TX_POWER_MARKER : Error marker for RFFE errors.
- ERRNO_NO_TRIMS : No trims found when attempting to adjust voltage rails.
- ERRNO_RFFE_MISSINGSETTING_ERROR : Setting does not exist.
- ERRNO_RFFE_INVALIDSETTING_ERROR : Setting is not possible.
- ERRNO_RFFE_VCC_INSUFFICIENT : VCC is too low to increase VDDRF sufficiently to suppor the requested RF output power.
- WARNING_RFFE_VLOW_POWER_STATE : Warning that the device is in a very low RF output power state.
- WARNING_RFFE_PA_ENABLED_STATE : Warning that the device has the power amplifier enabled.
- CONVERT : Converts an ADC code to a voltage, calculated as follows voltage = adc_code * (2 V * 1000 [mV]/1 V / 2^14 steps).
- SWAP : Swap the values in variables a and b.
- SYS_RFFE_SETTXPOWER : Set the TX Power according to the desired target value with an accuracy of +/-1 dBm for +6 dBm to -17 dBm.

Functions

- Sys_RFFE_GetTXPower : Retrieve the current setting for RF output power by using the values retrieved from the appropriate registers.
- Sys_RFFE_SetTXPower : Set the TX Power according to the desired target value with an accuracy of +/-1 dBm for +6 dBm to -17 dBm.

17.13.2 RFFE Radio Frequency Front End Macro Definition Documentation

17.13.2.1 STABILIZATION_DELAY

```
#define STABILIZATION_DELAY (SystemCoreClock * 3 / 625)
```

Location: rffe.h:46

External include files.

Three times the length of time corresponding to the minimum sample rate, which is deemed sufficient to allow the LSAD to stabilize

17.13.2.2 MEASUREMENT_DELAY

```
#define MEASUREMENT_DELAY (SystemCoreClock / 625)
```

Location: rffe.h:49

Corresponds to sample rate of the LSAD as configured (625 Hz)

17.13.2.3 V_TO_MV

```
#define V_TO_MV 1000
```

Location: rffe.h:52

Factor for converting back and forth from mV to V.

17.13.2.4 V_TO_MV_F

```
#define V_TO_MV_F 1000.0f
```

Location: rffe.h:55

Float iteration of factor for converting back and forth from mV to V.

17.13.2.5 DEF_CHANNEL

```
#define DEF_CHANNEL 6
```

Location: rffe.h:58

Default LSAD channel used to measure voltage rails.

17.13.2.6 MAX_LSAD_CHANNEL

```
#define MAX_LSAD_CHANNEL 7
```

Location: rffe.h:61

Maximum number of LSAD channels in the design.

17.13.2.7 VDDPA_EN

```
#define VDDPA_EN 1
```

Location: rffe.h:64

VDDPA enable selection.

17.13.2.8 VDDPA_DIS

```
#define VDDPA_DIS 0
```

Location: rffe.h:67

VDDPA disable selection.

17.13.2.9 VCC_VDDRF_MARGIN

```
#define VCC_VDDRF_MARGIN 50
```

Location: rffe.h:70

We strongly recommended having VCC at least 50mV higher than VDDRF.

17.13.2.10 TRIM_MARGIN

```
#define TRIM_MARGIN 10    /*mv*/
```

Location: rffe.h:74

Trim margin.

The granularity of trims for VDDRF allow for a certain error above or below the set value

17.13.2.11 MV_PER_DBM_VDDPA

```
#define MV_PER_DBM_VDDPA 100
```

Location: rffe.h:77

Estimated voltage increase per 1dBm increased TX power.

17.13.2.12 MV_PER_DBM_VDDRF

```
#define MV_PER_DBM_VDDRF 60
```

Location: rffe.h:80

Estimated voltage increase per 1dBm increased TX power.

17.13.2.13 STEPS_PER_DBM_VDDRF

```
#define STEPS_PER_DBM_VDDRF 6
```

Location: rffe.h:83

Estimated trim steps per 1dBm increased TX power.

17.13.2.14 STEPS_PER_DBM_VDDPA

```
#define STEPS_PER_DBM_VDDPA 10
```

Location: rffe.h:86

Estimated trim steps per 1dBm increased TX power.

17.13.2.15 RF_MAX_POWER

```
#define RF_MAX_POWER 6
```

Location: rffe.h:89

Maximum RF output power possible.

17.13.2.16 RF_MAX_POWER_NO_VDDPA

```
#define RF_MAX_POWER_NO_VDDPA 2
```

Location: rffe.h:92

Maximum RF output power possible without using VDDPA.

17.13.2.17 RF_NO_VDDPA_TYPICAL_POWER

```
#define RF_NO_VDDPA_TYPICAL_POWER 0
```

Location: rffe.h:95

Typical RF output power when VDDPA is not used, with default trims.

17.13.2.18 RF_DEFAULT_POWER

```
#define RF_DEFAULT_POWER 0
```

Location: rffe.h:98

RF output power used by default.

17.13.2.19 RF_MIN_POWER

```
#define RF_MIN_POWER -17
```

Location: rffe.h:101

Minimum possible RF output power.

17.13.2.20 PA_PWR_BYTE_0DBM

```
#define PA_PWR_BYTE_0DBM 0x0C
```

Location: rffe.h:104

RF Output Power code for 0dBm.

17.13.2.21 PA_ENABLE_BIAS_SETTING

```
#define PA_ENABLE_BIAS_SETTING 0xF3
```

Location: rffe.h:107

Power amplifier bias enable.

17.13.2.22 PA_DISABLE_BIAS_SETTING

```
#define PA_DISABLE_BIAS_SETTING 0x73
```

Location: rffe.h:110

Power amplifier bias disable.

17.13.2.23 SW_CTRL_DELAY_3_BYTE

```
#define SW_CTRL_DELAY_3_BYTE ((uint8_t)0x2U)
```

Location: rffe.h:114

Switch control delay.

17.13.2.24 RAMPUP_DELAY_3_BYTE

```
#define RAMPUP_DELAY_3_BYTE ((uint8_t)0x2U)
```

Location: rffe.h:117

Ramp-up delay.

17.13.2.25 DISABLE_DELAY_3_BYTE

```
#define DISABLE_DELAY_3_BYTE ((uint8_t)0x2U)
```

Location: rffe.h:120

Disable delay.

17.13.2.26 ERRNO_TX_POWER_MARKER

```
#define ERRNO_TX_POWER_MARKER 0x30
```

Location: rffe.h:124

Error marker for RFFE errors.

17.13.2.27 ERRNO_NO_TRIMS

```
#define ERRNO_NO_TRIMS (0x01 | ERRNO_TX_POWER_MARKER)
```

Location: rffe.h:127

No trims found when attempting to adjust voltage rails.

17.13.2.28 ERRNO_RFFE_MISSINGSETTING_ERROR

```
#define ERRNO_RFFE_MISSINGSETTING_ERROR (0x02 | ERRNO_TX_POWER_MARKER)
```

Location: rffe.h:130

Setting does not exist.

17.13.2.29 ERRNO_RFFE_INVALIDSETTING_ERROR

```
#define ERRNO_RFFE_INVALIDSETTING_ERROR (0x03 | ERRNO_TX_POWER_MARKER)
```

Location: rffe.h:133

Setting is not possible.

17.13.2.30 ERRNO_RFFE_VCC_INSUFFICIENT

```
#define ERRNO_RFFE_VCC_INSUFFICIENT (0x04 | ERRNO_TX_POWER_MARKER)
```

Location: rffe.h:136

VCC is too low to increase VDDRF sufficiently to suppor the requested RF output power.

17.13.2.31 WARNING_RFFE_VLOW_POWER_STATE

```
#define WARNING_RFFE_VLOW_POWER_STATE (0x05 | ERRNO_TX_POWER_MARKER)
```

Location: rffe.h:140

Warning that the device is in a very low RF output power state.

17.13.2.32 WARNING_RFFE_PA_ENABLED_STATE

```
#define WARNING_RFFE_PA_ENABLED_STATE (0x06 | ERRNO_TX_POWER_MARKER)
```

Location: rffe.h:143

Warning that the device has the power amplifier enabled.

17.13.2.33 CONVERT

```
#define CONVERT ((uint32_t)((x * 1000) >> 13))
```

Location: rffe.h:154

Converts an ADC code to a voltage, calculated as follows voltage = adc_code * (2 V * 1000 [mV]/1 V / 2^14 steps).

Returns:

The voltage output in mV

Assumptions

Low frequency mode for the ADC is used, meaning that the resolution of the ADC is 14-bits. CONVERT provides voltage level as a milliVolt value based on the input ADC code.

Parameters

Direction	*Name*	Description
in	*x*	the ADC code input

17.13.2.34 SWAP

```
#define SWAP ((t) = (a), (a) = (b), (b) = (t))
```

Location: rffe.h:164

Swap the values in variables a and b.

Returns:

a holds the value previously in b

Returns:

b holds the value previously in a

Parameters

Direction	Name	Description
in	*a*	holds the value that must go to b
in	*b*	holds the value that must go to a
in	*t*	a temporary buffer for the swap

17.13.2.35 SYS_RFFE_SETTXPOWER

```
#define SYS_RFFE_SETTXPOWER Sys_RFFE_SetTXPower(target, DEF_CHANNEL, VDDPA_DIS)
```

Location: rffe.h:208

Set the TX Power according to the desired target value with an accuracy of +/-1 dBm for +6 dBm to -17 dBm.

This function sets VDDRF, VDDPA, and PA_PWR (RF_REG1A) when applicable. Note: This function provides RF TX power configurations that match the requested levels, without considering the potential for increased power consumption due to the use of VDDPA. target Target transmission power in the range from -17 to +6 dBm in 1 dBm increments.

Returns:

Return value error value, if any

Example Code for SYS_RFFE_SETTXPOWER

```
    // Reads the current register settings and measures VDDRF, if requried, to
    // determine the current TX power setting.
    result = SYS_RFFE_SETTXPOWER(0);
```

Returns:

Return value error value, if any

Example Code for SYS_RFFE_SETTXPOWER

```
    // Reads the current register settings and measures VDDRF, if requried, to
    // determine the current TX power setting.
    result = SYS_RFFE_SETTXPOWER(0);
```

Parameters

Direction	*Name*	Description
in	*target*	Target transmission power in the range from -17 to +6 dBm in 1 dBm increments.

17.13.3 RFFE Radio Frequency Front End Function Documentation

17.13.3.1 Sys_RFFE_GetTXPower

```
int8_t Sys_RFFE_GetTXPower(uint32_t lsad_channel)
```

Location: rffe.h:173

Retrieve the current setting for RF output power by using the values retrieved from the appropriate registers.

Returns:

The currently set TX output power. Returns -100 in error state.

Example Code for Sys_RFFE_GetTXPower

```
    // Use LSAD channel 0 to measure VDDRF if neccessary to determine the currently
    // set TX output power.
    result = Sys_RFFE_GetTXPower(0);
```

Parameters

Direction	Name	Description
in	*lsad_channel*	The LSAD channel used for measuring VDDRF

17.13.3.2 Sys_RFFE_SetTXPower

```
uint32_t Sys_RFFE_SetTXPower(int8_t target, uint8_t lsad_channel, bool pa_en)
```

Location: rffe.h:193

Set the TX Power according to the desired target value with an accuracy of +/-1 dBm for +6 dBm to -17 dBm.

This function sets VDDRF, VDDPA, and PA_PWR (RF_REG1A) when applicable. Note: This function provides RF TX power configurations that match the requested levels, without considering the potential for increased power consumption due to the use of VDDPA. target Target transmission power in the range from -17 to +6 dBm in 1 dBm increments. lsad_channel LSAD channel to use to measure rails, if necessary. pa_en If 1, the power amplifier will be enabled, otherwise, it will be disabled. The power amplifier will be enabled regardless if 'target' is greater than 2.
Returns:
Return value error value, if any

Example Code for Sys_RFFE_SetTXPower

```
// Set the TX power for the device to 6 dBm, uses LSAD channel 0 to measure
// (unused in the case of 6 dBm), and enables VDDPA.
result = Sys_RFFE_SetTXPower(6, 0, VDDPA_EN);
```

Returns:

Return value error value, if any

Example Code for Sys_RFFE_SetTXPower

```
// Set the TX power for the device to 6 dBm, uses LSAD channel 0 to measure
// (unused in the case of 6 dBm), and enables VDDPA.
result = Sys_RFFE_SetTXPower(6, 0, VDDPA_EN);
```

Parameters

Direction	Name	Description
`in`	*target*	Target transmission power in the range from -17 to +6 dBm in 1 dBm increments.
`in`	*lsad_channel*	LSAD channel to use to measure rails, if necessary.
`in`	*pa_en*	If 1, the power amplifier will be enabled, otherwise, it will be disabled. The power amplifier will be enabled regardless if 'target' is greater than 2.

17.14 RTC

Real Time Clock Hardware Abstraction Layer.

17.14.1 Summary

Functions

- Sys_RTC_Config : Configure RTC block.
- Sys_RTC_Value : Read the current value of the RTC timer.

17.14.2 RTC Function Documentation

17.14.2.1 Sys_RTC_Config

```
void Sys_RTC_Config(uint32_t start_value, uint32_t rtc_ctrl)
```

Location: rtc.h:48

Configure RTC block.

Example Code for Sys_RTC_Config

```
    // Configure and enable the real time clock to count down from 32767, use the
    // external 32kHz crystal as the clock, fire an alarm once a second.
    Sys_RTC_Config(RTC_CNT_START_32767, RTC_CLK_SRC_XTAL32K |
                   RTC_ALARM_1S |
                   RTC_ENABLE);
```

Parameters

Direction	Name	Description
in	*start_value*	Start value for the RTC timer counter; use a 32 bit value
in	*rtc_ctrl*	RTC control register; use RTC_FORCE_CLOCK, RTC_RESET, RTC_ALARM_*, RTC_CLK_SRC_*, and RTC_[DISABLE | ENABLE]

17.14.2.2 Sys_RTC_Value

```
uint32_t Sys_RTC_Value()
```

Location: rtc.h:60

Read the current value of the RTC timer.

Returns:

RTC timer counter current value

Example Code for Sys_RTC_Value

```
    // Return the current value of the RTC counter
    result = Sys_RTC_Value();
```

17.15 Simple Assertions

Simple Assertion (SASSERT) hardware abstraction layer.

17.15.1 Summary

Macros

- SYS_ASSERT : Assertion handler; default behavior is no operation.

17.15.2 Simple Assertions Macro Definition Documentation

17.15.2.1 SYS_ASSERT

```
#define SYS_ASSERT False
```

Location: sassert.h:47

Assertion handler; default behavior is no operation.

17.16 SENSOR API

Sensor Hardware Abstraction Layer.

17.16.1 Summary

Functions

- Sys_Sensor_ADCConfig : Configure the sensor interface, ensuring the sensor is powered if possible.
- Sys_Sensor_IntConfig : Configure sensor integration states.
- Sys_Sensor_IdleConfig : Configure sensor idle time state.
- Sys_Sensor_TimerConfig : Configure sensor timer settings.
- Sys_Sensor_StorageConfig : Configure data storage settings.
- Sys_Sensor_DelayConfig : Configure sensor delay clocks and length.
- Sys_Sensor_Enable : Enable the sensor.
- Sys_Sensor_Disable : Disable the sensor.
- Sys_Sensor_TimerReset : The sensor timer counter, the sensor timer enable and the ADC counter are reset.
- Sys_Sensor_CurrentState : Read the current delay state of the sensor interface.
- Sys_Sensor_CurrentCountValue : Read the current value of the sensor's main counter.

17.16.2 Sensor API Function Documentation

17.16.2.1 Sys_Sensor_ADCConfig

```
 void Sys_Sensor_ADCConfig(uint32_t if_cfg, uint32_t wedac_high, uint32_t wedac_
low, uint32_t clk_cfg)
```

Location: sensor.h:48

Configure the sensor interface, ensuring the sensor is powered if possible.

Example Code for Sys_Sensor_ADCConfig

```
     // Configure and enable the sensor interface with the following configuration:
     //   - RE connected to VSSA (ground)
     //   - Interface measurement range set to 50 nA
     //   - Default interface current offset set to 20 nA
     //   - Enable internal automatic calibration of the sensor ADC
     //   - Enable the sensor amplifier
     //   - Enable the guard buffer
     //   - Set high and low states WEDAC voltage to 600 mV
     //   - Set sensor interface clock source to the RTC
     Sys_Sensor_ADCConfig(SENSOR_ENABLED |
                          SENSOR_RE_VSSA         |
                          SENSOR_IRANGE_50NA     |
                          SENSOR_IOFFSET_20NA    |
                          SENSOR_CALIB_ENABLED   |
                          SENSOR_AMP_ENABLED     |
                          SENSOR_GUARD_ENABLED, SENSOR_WEDAC_HIGH_0600, SENSOR_WEDAC_
LOW_0600,
                          SENSOR_CLK_RTC);
```

Parameters

Direction	Name	Description
in	*if_cfg*	interface configuration
in	*wedac_high*	WEDAC output voltage during pulse (16mV steps) (except 0x1F -> 0x20 = 8mV step.)
in	*wedac_low*	WEDAC output voltage not during pulse (16mV steps) (except 0x1F -> 0x20 = 8mV step.)
in	*clk_cfg*	select interface clock source

17.16.2.2 Sys_Sensor_IntConfig

```
void Sys_Sensor_IntConfig(uint32_t pulse_count, uint32_t pre_count)
```

Location: sensor.h:79

Configure sensor integration states.

Example Code for Sys_Sensor_IntConfig

```
    // Configure the duration of sensor integration states, pulse count and
    // pre count. Both states are set to 2 periods of the SENSOR_CLK.
    Sys_Sensor_IntConfig(2,2);
```

Parameters

Direction	Name	Description
`in`	*pulse_count*	Duration of "Pulse Count Sample" state
`in`	*pre_count*	Absolute Value of main counter to trigger the change of "Pre Count Sample" state

17.16.2.3 Sys_Sensor_IdleConfig

```
void Sys_Sensor_IdleConfig(uint32_t idle_count)
```

Location: sensor.h:96

Configure sensor idle time state.

Example Code for Sys_Sensor_IdleConfig

```
    // Set the duration of the idle state of the sensor interface to 2 periods
    // of the SENSOR_CLK
    Sys_Sensor_IdleConfig(2);
```

Parameters

Direction	Name	Description
`in`	*idle_count*	Absolute Value of main counter to trigger the change of "Idle Time" state

17.16.2.4 Sys_Sensor_TimerConfig

```
void Sys_Sensor_TimerConfig(uint8_t cfg, uint8_t re_idle_connect)
```

Location: sensor.h:111

Configure sensor timer settings.

Example Code for Sys_Sensor_TimerConfig

```
    // Set the duration of the idle state of the sensor interface to 2 periods
    // of the SENSOR_CLK
    Sys_Sensor_IdleConfig(2);
```

Parameters

Direction	Name	Description
`in`	*cfg*	timer configuration
`in`	*re_idle_connect*	RE pad connected during idle state switch

17.16.2.5 Sys_Sensor_StorageConfig

```
 void Sys_Sensor_StorageConfig(uint32_t diff_mode, uint32_t sum_en, uint32_t nbr_
samples, uint32_t threshold, uint32_t store_en, uint32_t fifo_size)
```

Location: sensor.h:136

Configure data storage settings.

Example Code for Sys_Sensor_StorageConfig

```
    // Configure the way sensor samples are acquired and stored:
    //  - Enabling storing of values that are a difference of a pair of samples
    //  - Enabling of summation of values
    //  - Use 255 samples when summing values and for the sensor wakeup threshold
    //  - Raw sensor data needs to be higher than 1024 in order to increment wakeup
    //    counter
    //  - Enabling storing of values in a fifo, with a size of 8 samples (wakeup
    //    after 8 samples)
    Sys_Sensor_StorageConfig(SENSOR_DIFF_MODE_ENABLED, SENSOR_SUMMATION_ENABLED,
                             SENSOR_NBR_SAMPLES_255, 1024,
                             SENSOR_FIFO_STORE_ENABLED, SENSOR_FIFO_SIZE8);
```

Parameters

Direction	Name	Description
`in`	*diff_mode*	enable differential storage mode
`in`	*sum_en*	enable summation mode
`in`	*nbr_samples*	number of samples to store before wakeup in sensor detect mode or for impedance measurement
`in`	*threshold*	Sensor data level threshold for wakeup
`in`	*store_en*	Enable storing samples in FIFO
`in`	*fifo_size*	Number of samples to store in FIFO before wakeup of core.

17.16.2.6 Sys_Sensor_DelayConfig

```
 void Sys_Sensor_DelayConfig(uint32_t clk_1l, uint32_t clk_2l, uint32_t clk_1h,
uint32_t clk_2h, uint32_t len_1l, uint32_t len_2l, uint32_t len_1h, uint32_t len_
2h)
```

Location: sensor.h:177

Configure sensor delay clocks and length.

Use sub-register defines.
NOTE: Clocks can be 32kHz(0) or 1kHz(1-default)
clk_1l clock for delay state "Delay 1 WE_L" clk_2l clock for delay state "Delay 2 WE_L" clk_1h clock for delay state "Delay 1 WE_H" clk_2h clock for delay state "Delay 2 WE_H" len_1l number of periods for delay state "Delay 1 WE_L" len_2l number of periods for delay state "Delay 2 WE_L" len_1h number of periods for delay state "Delay 1 WE_H" len_2h number of periods for delay state "Delay 2 WE_H"

Example Code for Sys_Sensor_DelayConfig

```
    // Configure the length of the various delay states. Select 1 kHz clock source
    // for all delay states, and set each delay states to 255 periods of that clock.
    Sys_Sensor_DelayConfig(DLY1_WE_L_DIV_ENABLED_SHORT, DLY2_WE_L_DIV_ENABLED_SHORT,
                           DLY1_WE_H_DIV_ENABLED_SHORT, DLY2_WE_H_DIV_ENABLED_SHORT,
                           DLY1_WE_L_255_SHORT, DLY2_WE_L_255_SHORT,
                           DLY1_WE_H_255_SHORT, DLY2_WE_H_255_SHORT);
```

NOTE: Clocks can be 32kHz(0) or 1kHz(1-default)

Example Code for Sys_Sensor_DelayConfig

```
    // Configure the length of the various delay states. Select 1 kHz clock source
    // for all delay states, and set each delay states to 255 periods of that clock.
    Sys_Sensor_DelayConfig(DLY1_WE_L_DIV_ENABLED_SHORT, DLY2_WE_L_DIV_ENABLED_SHORT,
                           DLY1_WE_H_DIV_ENABLED_SHORT, DLY2_WE_H_DIV_ENABLED_SHORT,
                           DLY1_WE_L_255_SHORT, DLY2_WE_L_255_SHORT,
                           DLY1_WE_H_255_SHORT, DLY2_WE_H_255_SHORT);
```

Parameters

Direction	Name	Description
`in`	*clk_1l*	clock for delay state "Delay 1 WE_L"
`in`	*clk_2l*	clock for delay state "Delay 2 WE_L"
`in`	*clk_1h*	clock for delay state "Delay 1 WE_H"
`in`	*clk_2h*	clock for delay state "Delay 2 WE_H"
`in`	*len_1l*	number of periods for delay state "Delay 1 WE_L"
`in`	*len_2l*	number of periods for delay state "Delay 2 WE_L"
`in`	*len_1h*	number of periods for delay state "Delay 1 WE_H"
`in`	*len_2h*	number of periods for delay state "Delay 2 WE_H"

17.16.2.7 Sys_Sensor_Enable

```
void Sys_Sensor_Enable()
```

Location: sensor.h:198

Enable the sensor.

In secure mode, also enable power to the sensor.

Example Code for Sys_Sensor_Enable

```
    // Enable the ULP power data acquisition (sensor) subsystem
    Sys_Sensor_Enable()
```

Example Code for Sys_Sensor_Enable

```
    // Enable the ULP power data acquisition (sensor) subsystem
    Sys_Sensor_Enable()
```

17.16.2.8 Sys_Sensor_Disable

```
void Sys_Sensor_Disable()
```

Location: sensor.h:217

Disable the sensor.

In secure mode, also disable power to the sensor.

Example Code for Sys_Sensor_Disable

```
    // Disable the ULP power data acquisition (sensor) subsystem
    Sys_Sensor_Disable()
```

Example Code for Sys_Sensor_Disable

```
    // Disable the ULP power data acquisition (sensor) subsystem
    Sys_Sensor_Disable()
```

17.16.2.9 Sys_Sensor_TimerReset

```
void Sys_Sensor_TimerReset()
```

Location: sensor.h:237

The sensor timer counter, the sensor timer enable and the ADC counter are reset.

Example Code for Sys_Sensor_TimerReset

```
// Reset the sensor timer interface, including the timer counter, the enable
// bit and the ADC counter.
Sys_Sensor_TimerReset();
```

17.16.2.10 Sys_Sensor_CurrentState

```
uint8_t Sys_Sensor_CurrentState()
```

Location: sensor.h:249

Read the current delay state of the sensor interface.

Returns:

current state of the sensor interface.

Example Code for Sys_Sensor_CurrentState

```
// Returns the current state of the sensor interface
state = Sys_Sensor_CurrentState();
```

17.16.2.11 Sys_Sensor_CurrentCountValue

```
uint32_t Sys_Sensor_CurrentCountValue()
```

Location: sensor.h:261

Read the current value of the sensor's main counter.

Returns:

current value of the main counter.

Example Code for Sys_Sensor_CurrentCountValue

```
    // Returns the current value of the sensor's main counter.
    result = Sys_Sensor_CurrentCountValue();
```

17.17 SPI

Serial Peripheral Interface (SPI) hardware abstraction layer.

17.17.1 Summary

Macros

- SPI_CONFIG_MASK : Mask for the SPI_CFG register.
- SPI_PADS_NUM : Number of pads used for the SPI interface, for a single instance.
- SYS_SPI_CONFIG : Configure the specified SPI interface's operation and controller information.
- SYS_SPI_TRANSFERCONFIG : Configure the SPI transfer information for the specified SPI instance.
- SYS_SPI_READ : Generate clock and CS to read data from SPI interface.
- SYS_SPI_WRITE : Generate clock and CS to write data to SPI interface.
- SYS_SPI_GPIOCONFIG : Configure four GPIOs for the SPI0 interface.
- SYS_DSPI_GPIOCONFIG : Configure four GPIOs for the SPI0 interface for DSPI.
- SYS_QSPI_GPIOCONFIG : Configure six GPIOs for the SPI0 interface for QSPI cfg GPIO pin configuration for the SPI pads clk GPIO to use as the QSPI clock pad cs GPIO to use as the QSPI chip select pad io0 GPIO to use as the QSPI io0 io1 GPIO to use as the QSPI io1 io2 GPIO to use as the QSPI io2 io3 GPIO to use as the QSPI io3.

Functions

- Sys_SPI_Config : Configure the specified SPI interface's operation and controller information.
- Sys_SPI_TransferConfig : Configure the SPI transfer information for the specified SPI instance.
- Sys_SPI_Read : Generate clock and CS to read data from SPI interface.
- Sys_SPI_Write : Generate clock and CS to write data to SPI interface.
- Sys_SPI_GPIOConfig : Configure four GPIOs for the specified SPI interface.
- Sys_DSPI_GPIOConfig : Configure four GPIOs for the specified SPI interface.
- Sys_QSPI_GPIOConfig : Configure six GPIOs for the specified SPI interface.

17.17.2 SPI Macro Definition Documentation

17.17.2.1 SPI_CONFIG_MASK

```
#define SPI_CONFIG_MASK ((1 << SPI_CFG_TX_DMA_ENABLE_Pos)            | \
    (1 << SPI_CFG_RX_DMA_ENABLE_Pos)            | \
    (1 << SPI_CFG_TX_END_INT_ENABLE_Pos)        | \
    (1 << SPI_CFG_TX_START_INT_ENABLE_Pos)      | \
    (1 << SPI_CFG_RX_INT_ENABLE_Pos)            | \
    (1 << SPI_CFG_CS_RISE_INT_ENABLE_Pos)       | \
    (1 << SPI_CFG_OVERRUN_INT_ENABLE_Pos)       | \
    (1 << SPI_CFG_UNDERRUN_INT_ENABLE_Pos)      | \
    (1 << SPI_CFG_MODE_Pos)                     | \
    SPI_CFG_MODE_Mask                           | \
    (1 << SPI_CFG_WORD_SIZE_Pos)                | \
    SPI_CFG_WORD_SIZE_Mask                      | \
    (1 << SPI_CFG_PRESCALE_Pos)                 | \
    SPI_CFG_PRESCALE_Mask                       | \
    (1 << SPI_CFG_CLK_POLARITY_Pos)             | \
    (1 << SPI_CFG_SLAVE_Pos))
```

Location: spi.h:61

Mask for the SPI_CFG register.

17.17.2.2 SPI_PADS_NUM

```
#define SPI_PADS_NUM 6
```

Location: spi.h:81

Number of pads used for the SPI interface, for a single instance.

17.17.2.3 SYS_SPI_CONFIG

```
#define SYS_SPI_CONFIG Sys_SPI_Config(SPI, (config))
```

Location: spi.h:304

Configure the specified SPI interface's operation and controller information.

Example Code for SYS_SPI_CONFIG

```
    // Configure the default SPI interface's operation and controller information:
    //  - Master mode
    //  - Use 8-bit words
    //  - Select interrupts enabled
    //  - Prescale the SPI interface clock by 32
    SYS_SPI_CONFIG((SPI_MODE_QSPI | SPI_WORD_SIZE_8 | SPI_TX_START_INT_ENABLE |
                   SPI_RX_INT_ENABLE | SPI_OVERRUN_INT_ENABLE |
                   SPI_PRESCALE_32 | SPI_UNDERRUN_INT_ENABLE));
```

Parameters

Direction	*Name*	Description
in	*config*	Interface operation configuration; use SPI_SELECT_[MASTER | SLAVE], SPI_CLK_POLARITY_[NORMAL | INVERSE], SPI_PRESCALE_*, SPI_WORD_SIZE_*, SPI_MODE_[SPI | DSPI | QSPI] SPI_UNDERRUN_INT_[ENABLE | DISABLE], SPI_OVERRRUN_INT_[ENABLE | DISABLE], SPI_CS_RISE_INT_[ENABLE | DISABLE], SPI_RX_INT_[ENABLE | DISABLE] SPI_TX_INT_[ENABLE | DISABLE] SPI_RX_DMA_[ENABLE | DISABLE] SPI_TX_DMA_[ENABLE | DISABLE]

17.17.2.4 SYS_SPI_TRANSFERCONFIG

```
#define SYS_SPI_TRANSFERCONFIG Sys_SPI_TransferConfig(SPI, (config))
```

Location: spi.h:322

Configure the SPI transfer information for the specified SPI instance.

Example Code for SYS_SPI_TRANSFERCONFIG

```
    // Enable and configure default SPI interface to read operation mode
    SYS_SPI_TRANSFERCONFIG(SPI_ENABLE | SPI_MODE_READ);
```

Parameters

Direction	*Name*	Description
in	*config*	Interface transfer configuration; use SPI_ENABLE, SPI_DISABLE, SPI_RESET, SPI_START, SPI_MODE_READ_WRITE, SPI_MODE_READ, SPI_MODE_WRITE, SPI_MODE_NOP, SPI_CS_0, SPI_CS_1,

17.17.2.5 SYS_SPI_READ

```
#define SYS_SPI_READ Sys_SPI_Read(SPI)
```

Location: spi.h:332

Generate clock and CS to read data from SPI interface.

Returns:

Data read from the SPI interface

Assumptions

SPI is configured as master mode and transfer operation mode is SPI_MODE_READ_WRITE(full duplex) or (SPI_MODE_READ) half duplex

Example Code for SYS_SPI_READ

```
    // Generate clock and CS to read data from the default SPI interface
    SYS_SPI_READ();
```

17.17.2.6 SYS_SPI_WRITE

```
#define SYS_SPI_WRITE Sys_SPI_Write(SPI, (data))
```

Location: spi.h:342

Generate clock and CS to write data to SPI interface.

Assumptions

SPI is configured as master mode and transfer operation mode is SPI_MODE_READ_WRITE(full duplex) or (SPI_MODE_WRITE) half duplex

Example Code for SYS_SPI_WRITE

```
    // Generate clock and CS to write data to the default SPI interface
    SYS_SPI_WRITE(0xFF);
```

Parameters

Direction	*Name*	Description
in	*data*	Data to be sent over SPI

17.17.2.7 SYS_SPI_GPIOCONFIG

```
#define SYS_SPI_GPIOCONFIG Sys_SPI_GPIOConfig(SPI, (slave), (cfg), (clk), (cs), (seri),
(sero))
```

Location: spi.h:355

Configure four GPIOs for the SPI0 interface.

Example Code for SYS_SPI_GPIOCONFIG

```
    // Configure GPIOs 0, 1, 2, and 3 for the default SPI interface with
    // low-pass filter disabled, 8X drive-strength, and 1 kOhm pull-up resistors
    SYS_SPI_GPIOCONFIG(GPIO0, (GPIO_LPF_DISABLE |GPIO_8X_DRIVE |
                      GPIO_1K_PULL_UP), GPIO0, GPIO1, GPIO2, GPIO3);
```

Parameters

Direction	Name	Description
in	*slave*	SPI master/slave configuration; use SPI*_SELECT_ [MASTER | SLAVE]
in	*cfg*	GPIO pin configuration for the SPI pads
in	*clk*	GPIO to use as the SPI clock pad
in	*cs*	GPIO to use as the SPI chip select pad
in	*seri*	GPIO to use as the SPI serial input pad
in	*sero*	GPIO to use as the SPI serial output pad

17.17.2.8 SYS_DSPI_GPIOCONFIG

```
#define SYS_DSPI_GPIOCONFIG Sys_DSPI_GPIOConfig(SPI, (cfg), (clk), (cs), (io0), (io1))
```

Location: spi.h:367

Configure four GPIOs for the SPI0 interface for DSPI.

Example Code for SYS_DSPI_GPIOCONFIG

```
    // Configure GPIOs 0, 1, 2, and 3 for the default DSPI interface with
    // low-pass filter disabled, 8X drive-strength, and 1 kOhm pull-up resistors
    SYS_DSPI_GPIOCONFIG((GPIO_LPF_DISABLE | GPIO_8X_DRIVE |
                      GPIO_1K_PULL_UP), GPIO0, GPIO1, GPIO2, GPIO3);
```

Parameters

Direction	*Name*	Description
in	*cfg*	GPIO pin configuration for the SPI pads
in	*clk*	GPIO to use as the DSPI clock pad
in	*cs*	GPIO to use as the DSPI chip select pad
in	*io0*	GPIO to use as the DSPI io0
in	*io1*	GPIO to use as the DSPI io1

17.17.2.9 SYS_QSPI_GPIOCONFIG

```
#define SYS_QSPI_GPIOCONFIG Sys_QSPI_GPIOConfig(SPI, (cfg), (clk), (cs), (io0), (io1), \
                        (io2), (io3))
```

Location: spi.h:381

Configure six GPIOs for the SPI0 interface for QSPI cfg GPIO pin configuration for the SPI pads clk GPIO to use as the QSPI clock pad cs GPIO to use as the QSPI chip select pad io0 GPIO to use as the QSPI io0 io1 GPIO to use as the QSPI io1 io2 GPIO to use as the QSPI io2 io3 GPIO to use as the QSPI io3.

Example Code for SYS_QSPI_GPIOCONFIG

```
    // Configure GPIOs 0, 1, 2, 3, 4, and 5 for the default QSPI interface with
    // low-pass filter disabled, 8X drive-strength, and 1 kOhm pull-up resistors
    SYS_QSPI_GPIOCONFIG((GPIO_LPF_DISABLE | GPIO_8X_DRIVE | GPIO_1K_PULL_UP),
                        GPIO0, GPIO1, GPIO2, GPIO3, GPIO4, GPIO5);
```

17.17.3 SPI Function Documentation

17.17.3.1 Sys_SPI_Config

```
void Sys_SPI_Config(SPI_Type * spi, uint32_t config)
```

Location: spi.h:102

Configure the specified SPI interface's operation and controller information.

Example Code for Sys_SPI_Config

```
    // Configure the SPI interface's operation and controller information:
    //  - Master mode
    //  - Use 8-bit words
    //  - Select interrupts enabled
    //  - Prescale the SPI interface clock by 32
    Sys_SPI_Config(SPI, SPI_SELECT_MASTER | SPI_WORD_SIZE_8 | SPI_TX_START_INT_ENABLE |
                   SPI_RX_INT_ENABLE | SPI_OVERRUN_INT_ENABLE |
                   SPI_PRESCALE_32 | SPI_UNDERRUN_INT_ENABLE);
```

Parameters

Direction	Name	Description
in	*spi*	Pointer to the SPI instance
in	*config*	Interface operation configuration; use SPI_SELECT_[MASTER | SLAVE], SPI_CLK_POLARITY_[NORMAL | INVERSE], SPI_PRESCALE_*, SPI_WORD_SIZE_*, SPI_MODE_[SPI | DSPI | QSPI] SPI_UNDERRUN_INT_[ENABLE | DISABLE], SPI_OVERRRUN_INT_[ENABLE | DISABLE], SPI_CS_RISE_INT_[ENABLE | DISABLE], SPI_RX_INT_[ENABLE | DISABLE] SPI_TX_INT_[ENABLE | DISABLE] SPI_RX_DMA_[ENABLE | DISABLE] SPI_TX_DMA_[ENABLE | DISABLE]

17.17.3.2 Sys_SPI_TransferConfig

```
void Sys_SPI_TransferConfig(SPI_Type * spi, uint32_t config)
```

Location: spi.h:125

Configure the SPI transfer information for the specified SPI instance.

Example Code for Sys_SPI_TransferConfig

```
    // Enable and configure the SPI interface to read operation mode
    Sys_SPI_TransferConfig(SPI, SPI_ENABLE | SPI_MODE_READ);
```

Parameters

Direction	Name	Description
`in`	*spi*	Pointer to the SPI instance
`in`	*config*	Interface transfer configuration; use SPI_ENABLE, SPI_DISABLE, SPI_RESET, SPI_START, SPI_MODE_READ_WRITE, SPI_MODE_READ, SPI_MODE_WRITE, SPI_MODE_NOP, SPI_CS_0, SPI_CS_1,

17.17.3.3 Sys_SPI_Read

```
uint32_t Sys_SPI_Read(const SPI_Type * spi)
```

Location: spi.h:140

Generate clock and CS to read data from SPI interface.

Returns:

data Data read from the SPI interface

Assumptions

SPI is configured as master mode and transfer operation mode is SPI_MODE_READ_WRITE(full duplex) or (SPI_MODE_READ) half duplex

Example Code for Sys_SPI_Read

```
    // Generate clock and CS to read data from SPI interface
    Sys_SPI_Read(SPI);
```

Parameters

Direction	Name	Description
in	*spi*	Pointer to the SPI instance

17.17.3.4 Sys_SPI_Write

```
void Sys_SPI_Write(SPI_Type * spi, uint32_t data)
```

Location: spi.h:155

Generate clock and CS to write data to SPI interface.

Assumptions

SPI is configured as master mode and transfer operation mode is SPI_MODE_READ_WRITE(full duplex) or (SPI_MODE_WRITE) half duplex

Example Code for Sys_SPI_Write

```
    // Generate clock and CS to write data to SPI interface
    Sys_SPI_Write(SPI, 0xFF);
```

Parameters

Direction	Name	Description
in	*spi*	Pointer to the SPI instance
in	*data*	Data to be sent over SPI

17.17.3.5 Sys_SPI_GPIOConfig

```
void Sys_SPI_GPIOConfig(const SPI_Type * spi, uint32_t slave, uint32_t cfg,
uint32_t clk, uint32_t cs, uint32_t seri, uint32_t sero)
```

Location: spi.h:174

Configure four GPIOs for the specified SPI interface.

Example Code for Sys_SPI_GPIOConfig

```
    // Configure GPIOs 0, 1, 2, and 3 for the SPI0 interface with
    // low-pass filter disabled, 8X drive-strength, and 1 kOhm pull-up resistors
    Sys_SPI_GPIOConfig(SPI, GPIO0, (GPIO_LPF_DISABLE | GPIO_8X_DRIVE |
                        GPIO_1K_PULL_UP), GPIO0, GPIO1, GPIO2, GPIO3);
```

Parameters

Direction	Name	Description
in	*spi*	Pointer to the SPI instance
in	*slave*	SPI master/slave configuration; use SPI*_SELECT_ [MASTER | SLAVE]
in	*cfg*	GPIO pin configuration for the SPI pads
in	*clk*	GPIO to use as the SPI clock pad
in	*cs*	GPIO to use as the SPI chip select pad
in	*seri*	GPIO to use as the SPI serial input pad
in	*sero*	GPIO to use as the SPI serial output pad

17.17.3.6 Sys_DSPI_GPIOConfig

```
 void Sys_DSPI_GPIOConfig(const SPI_Type * spi, uint32_t cfg, uint32_t clk,
uint32_t cs, uint32_t io0, uint32_t io1)
```

Location: spi.h:228

Configure four GPIOs for the specified SPI interface.

Example Code for Sys_DSPI_GPIOConfig

```
    // Configure GPIOs 0, 1, 2, and 3 for the DSPI0 interface with
    // low-pass filter disabled, 8X drive-strength, and 1 kOhm pull-up resistors
    Sys_DSPI_GPIOConfig(SPI, (GPIO_LPF_DISABLE | GPIO_8X_DRIVE |
                       GPIO_1K_PULL_UP), GPIO0, GPIO1, GPIO2, GPIO3);
```

Parameters

Direction	Name	Description
in	*spi*	Pointer to the SPI instance
in	*cfg*	GPIO pin configuration for the SPI pads
in	*clk*	GPIO to use as the DSPI clock pad
in	*cs*	GPIO to use as the DSPI chip select pad
in	*io0*	GPIO to use as the DSPI io0
in	*io1*	GPIO to use as the DSPI io1

17.17.3.7 Sys_QSPI_GPIOConfig

```
 void Sys_QSPI_GPIOConfig(const SPI_Type * spi, uint32_t cfg, uint32_t clk,
uint32_t cs, uint32_t io0, uint32_t io1, uint32_t io2, uint32_t io3)
```

Location: spi.h:260

Configure six GPIOs for the specified SPI interface.

Example Code for Sys_QSPI_GPIOConfig

```
    // Configure GPIOs 0, 1, 2, 3, 4, and 5 for the QSPI0 interface with
    // low-pass filter disabled, 8X drive-strength, and 1 kOhm pull-up resistors
    Sys_QSPI_GPIOConfig(SPI, (GPIO_LPF_DISABLE | GPIO_8X_DRIVE | GPIO_1K_PULL_UP),
                       GPIO0, GPIO1, GPIO2, GPIO3, GPIO4, GPIO5);
```

Parameters

Direction	Name	Description
`in`	*spi*	Pointer to the SPI instance
`in`	*cfg*	GPIO pin configuration for the SPI pads
`in`	*clk*	GPIO to use as the QSPI clock pad
`in`	*cs*	GPIO to use as the QSPI chip select pad
`in`	*io0*	GPIO to use as the QSPI io0
`in`	*io1*	GPIO to use as the QSPI io1
`in`	*io2*	GPIO to use as the QSPI io2
`in`	*io3*	GPIO to use as the QSPI io3

17.18 General-Purpose Timer

General-purpose timer hardware abstraction layer.

17.18.1 Summary

Macros

- SYS_TIMER_CONFIG : Configures the default timer instance.
- SYS_TIMER_START : Starts the default timer instance.
- SYS_TIMER_STOP : Stops the default timer instance.

Functions

- Sys_Timer_Config : Configure timer instance.
- Sys_Timer_Start : Start or restart timer instance.
- Sys_Timer_Stop : Stop the timer instance.

17.18.2 General-Purpose Timer Macro Definition Documentation

17.18.2.1 SYS_TIMER_CONFIG

```
#define SYS_TIMER_CONFIG Sys_Timer_Config(TIMER, (cfg0), \
                                                           (cfg1), (timeout))
```

Location: timer.h:83

Configures the default timer instance.

17.18.2.2 SYS_TIMER_START

```
#define SYS_TIMER_START Sys_Timer_Start(TIMER)
```

Location: timer.h:87

Starts the default timer instance.

17.18.2.3 SYS_TIMER_STOP

```
#define SYS_TIMER_STOP Sys_Timer_Stop(TIMER)
```

Location: timer.h:90

Stops the default timer instance.

17.18.3 General-Purpose Timer Function Documentation

17.18.3.1 Sys_Timer_Config

```
void Sys_Timer_Config(TIMER_Type * timer, uint32_t cfg0, uint32_t cfg1, uint32_t
timeout)
```

Location: timer.h:51

Configure timer instance.

Example Code for Sys_Timer_Config

```
    // Configure TIMER2 instance:
    //  - Divide the input clock frequency by 2
    //  - Stop on 2nd Time-out occurrence and issue an interrupt
    //  - Select the GPIO interrupt defined in GPIO_INT_CFG3
    //  - GPIO interrupt single capture mode
    //  - Free-run mode
    //  - Long timeout
    Sys_Timer_Config(TIMER2, TIMER_PRESCALE_2,
                     TIMER_MULTI_COUNT_2 |
                     TIMER_SRC_GPIO_INT3   |
                     TIMER_GPIO_INT_SINGLE |
                     TIMER_FREE_RUN, 0xFFFF);
```

Parameters

Direction	Name	Description
`in`	*timer*	Pointer to the timer instance
`in`	*cfg0*	Timer configuration 0; use TIMER_PRESCALE_*,
`in`	*cfg1*	Timer configuration 1; use TIMER_MULTI_COUNT_* [TIMER_SHOT_MODE | TIMER_FREE_RUN]
`in`	*timeout*	number of timer clock cycles before a timeout would occur

17.18.3.2 Sys_Timer_Start

```
void Sys_Timer_Start(TIMER_Type * timer)
```

Location: timer.h:65

Start or restart timer instance.

Example Code for Sys_Timer_Start

```
    // Start or restart TIMER instance
    Sys_Timer_Start(TIMER);
```

Parameters

Direction	Name	Description
in	*timer*	Pointer to the timer instance

17.18.3.3 Sys_Timer_Stop

```
void Sys_Timer_Stop(TIMER_Type * timer)
```

Location: timer.h:76

Stop the timer instance.

Example Code for Sys_Timer_Stop

```
    // Stop the TIMER3 instance
    Sys_Timer_Stop(TIMER3);
```

Parameters

Direction	Name	Description
in	*timer*	Pointer to the timer instance

17.19 Time of Flight

Time of Flight (TOF) hardware abstraction layer.

17.19.1 Summary

Functions

- Sys_TOF_Config : Configure TOF module.
- Sys_TOF_Start : Start the time of flight module.
- Sys_TOF_Stop : Stop the time of flight module.

17.19.2 Time of Flight Function Documentation

17.19.2.1 Sys_TOF_Config

```
void Sys_TOF_Config(uint32_t cfg)
```

Location: tof.h:53

Configure TOF module.

Example Code for Sys_TOF_Config

```
    // Configure the time-of-flight module:
    //  - Enable the error, overrun and average data complete interrupts
    //  - Average data interrupt is triggered after 16 samples
    Sys_TOF_Config(TOF_ERROR_INT_ENABLE     |
                   TOF_OVERRUN_INT_ENABLE   |
                   TOF_AVG_DATA_INT_ENABLE  |
                   TOF_DATA_INT_ENABLE      |
                   TOF_AVG_DATA_16);
```

Parameters

Direction	Name	Description
in	*cfg*	Time of flight configuration; use [TOF_ERROR_INT_ENABLE | TOF_ERROR_INT_DISABLE], [TOF_OVERRUN_INT_ENABLE | TOF_OVERRUN_INT_DISABLE], [TOF_AVG_DATA_INT_ENABLE | TOF_AVG_DATA_INT_DISABLE], [TOF_DATA_INT_ENABLE | TOF_DATA_INT_DISABLE], [TOF_AVG_DATA_DMA_ENABLE | TOF_AVG_DATA_DMA_DISABLE], [TOF_DATA_DMA_ENABLE | TOF_DATA_DMA_DISABLE], TOF_AVG_DATA_*, TOF_STOP_SRC_*, TOF_START_SRC_*, TOF_CLK_PRESCALE_*

17.19.2.2 Sys_TOF_Start

```
void Sys_TOF_Start()
```

Location: tof.h:63

Start the time of flight module.

Example Code for Sys_TOF_Start

```
    // Start the time-of-flight counter.
    Sys_TOF_Start();
```

17.19.2.3 Sys_TOF_Stop

```
void Sys_TOF_Stop()
```

Location: tof.h:73

Stop the time of flight module.

Example Code for Sys_TOF_Stop

```
    // Stop the time-of-flight counter.
    Sys_TOF_Stop();
```

17.20 TRIMMING SUPPORT

Power, clock, and sensor component trimming hardware abstraction layer.

17.20.1 Summary

Variables

- trim_args1 : Trim targets for items needing one trim.
- trim_args2 : Trim targets for items needing two trims.

Enumerations

- TrimTarget_t : Default trim targets present in NVR7.
- TrimName_t : Voltage rail and oscillator names.

Macros

- NULL_POINTER : NULL pointer.
- MIN_32_BIT : Minimum 32-bit value.
- MAX_32_BIT : Maximum 32-bit value.
- MIN_18_BIT : Minimum 18-bit value.
- MAX_18_BIT : Maximum 18-bit value.
- MIN_16_BIT : Minimum 16-bit value.
- MAX_16_BIT : Maximum 16-bit value.
- MIN_8_BIT : Minimum 8-bit value.
- MAX_8_BIT : Maximum 8-bit value.
- MAX_4_BIT : Maximum 4-bit value.
- ERROR_NO_ERROR
- ERROR_NULL : Null pointer error.
- ERROR_NO_TRIM_FOUND : Target trim value not found.
- ERROR_INVALID_TRIM : Trims in region specified are not valid.
- ERROR_INVALID_CRC : Trim region CRC has failed.
- ERROR_BG_INVALID : Bandgap target value is invalid.
- ERROR_BG_V_INVALID : Bandgap voltage trim is invalid.
- ERROR_BG_I_INVALID : Bandgap current trim is invalid.
- ERROR_DCDC_INVALID : DCDC trim is invalid.
- ERROR_VDDC_INVALID : VDDC trim is invalid.
- ERROR_VDDC_STBY_INVALID : VDCC standby trim is invalid.
- ERROR_VDDM_INVALID : VDDM trim is invalid.
- ERROR_VDDM_STBY_INVALID : VDCM standby trim is invalid.
- ERROR_VDDRF_INVALID : VDDRF trim is invalid.
- ERROR_VDDPA_INVALID : VDDPA trim is invalid.
- ERROR_VDDPA_MIN_INVALID : VDDPA minimum trim is invalid.
- ERROR_VDDIF_INVALID : VDDIF trim is invalid.
- ERROR_VDDFLASH_INVALID : VDDFLASH trim is invalid.
- ERROR_RCOSC_INVALID : RC start oscillator trim is invalid.
- ERROR_RCOSC32_INVALID : RC standby oscillator trim is invalid.
- ERROR_LSAD_INVALID : LSAD gain or offset is invalid.
- ERROR_TEMPERATURE_INVALID : Temperature sensor gain or offset is invalid.
- ERROR_THERMISTOR_INVALID : Thermistor gain or offset is invalid.
- ERROR_MEASURED_INVALID : Measured reference temperature is invalid.
- ERROR_TRIM_CUSTOM_SIGNATURE_INVALID : Custom signature check is invalid.
- ERROR_TRIM_CUSTOM_ICH_INVALID : Custom ICH trim value is invalid.
- ERROR_TRIM_CUSTOM_XTAL_INVALID : Custom Xtal trim value is invalid.
- TR_REG_TRIM_MASK : Temperature record 18-bit trim value mask.
- TRIM_8_BIT_TRIM_MASK : 8-bit trim value mask
- TRIM_16_BIT_TRIM_MASK : 16-bit trim value mask
- LSAD_HF : LSAD high frequency compensation values.
- LSAD_LF : LSAD low frequency compensation values.
- LSAD_OFFSET : LSAD offset compensation address offset.
- LSAD_OFFSET_MASK : LSAD offset compensation mask.
- LSAD_GAIN : LSAD gain compensation address offset.
- LSAD_GAIN_MASK : LSAD gain compensation mask.

- TRIM : Default trim instance, pointing to NVR7.
- TRIM_SUPPLEMENTAL : Supplemental trim instance, pointing to NVR4.
- TRIM_CUSTOM_SIP1_SIGNATURE : SiP Signature for NVR6 custom trim calibration.
- TRIM_CUSTOM_CUST_SIGNATURE : Custom Signature for NVR6 custom trim calibration.
- SYS_TRIM_LOAD_DEFAULT : Load default trim values from NVR7.
- SYS_TRIM_LOAD_SUPPLEMENTAL : Load supplemental trim values from NVR4.
- SYS_TRIM_LOAD_CUSTOM : Load custom trim values from NVR6.

Functions

- Sys_Trim_LoadTrims : Load trim values from the specified memory location.
- Sys_Trim_LoadSingleTrim : Load a trim value for a specific voltage regulator or oscillator.
- Sys_Trim_VerifyTrims : Verify if the trims memory is populated correctly.
- Sys_Trim_CheckCRC : Check if the CRC for the indicated region is valid.
- Sys_Trim_GetTrim : Get the trim value requested, check if it is valid.
- Sys_Trim_LoadBandgap : Load target trim value, if present.
- Sys_Trim_LoadDCDC : Load target trim value for current mode (LDO or BUCK).
- Sys_Trim_LoadVDDC : Load target trim value, if present.
- Sys_Trim_LoadVDDM : Load target trim value, if present.
- Sys_Trim_LoadVDDPA : Load target trim value, if present.
- Sys_Trim_LoadVDDRF : Load target trim value, if present.
- Sys_Trim_LoadCustom : Load custom trim values from NVR6.
- Sys_Trim_LoadVDDFLASH : Load target trim value, if present.
- Sys_Trim_LoadRCOSC : Load target trim value, if present.
- Sys_Trim_LoadRCOSC32 : Load target trim value, if present.
- Sys_Trim_LoadThermistor : Load target trim value, if present.
- Sys_Trim_GetLSADTrim : Load LSAD gain and offset value from specified address. Verifies valid values first.
- Sys_Trim_LoadVDDIF : Load target trim value, if present.

17.20.2 Detailed Description

Trim Hardware Abstraction Layer.

17.20.3 Trimming Support Variable Documentation

17.20.3.1 trim_args1

```
uint32_t trim_args1[TRIM_NUM_FUNCTIONS_1_ARG]
```

Location: trim.h:271

Trim targets for items needing one trim.

17.20.3.2 trim_args2

```
uint32_t trim_args2[TRIM_NUM_FUNCTIONS_2_ARGS][2]
```

Location: trim.h:272

Trim targets for items needing two trims.

17.20.4 Trimming Support Enumeration Type Documentation

17.20.4.1 TrimTarget_t

Location: trim.h:169

Default trim targets present in NVR7.

Members

- `TARGET_BANDGAP_V = 75`

  750mV

- `TARGET_BANDGAP_I = 100`

  1000nA

- `TARGET_DCDC_1200 = 120`

  1.2V

- `TARGET_DCDC_1120 = 112`

  1.12V

- `TARGET_DCDC_1350 = 135`

1.35V

- `TARGET_DCDC_1100 = 110`

1.10V

- `TARGET_VDDC_1150 = 115`

1.15V

- `TARGET_VDDC_1000 = 100`

1.00V

- `TARGET_VDDC_1080 = 108`

1.08V

- `TARGET_VDDC_920 = 92`

0.92V

- `TARGET_VDDC_1050 = 105`

1.05V

- `TARGET_VDDC_STANDBY = 80`

0.80V

- `TARGET_VDDM_1150 = 115`

1.15V

- `TARGET_VDDM_1080 = 108`

  1.08V

- `TARGET_VDDM_1100 = 110`

  1.05V

- `TARGET_VDDM_STANDBY = 80`

  0.80V

- `TARGET_VDDRF_1100 = 110`

  1.10V

- `TARGET_VDDRF_1070 = 107`

  1.07V

- `TARGET_VDDRF_1200 = 120`

  1.20V

- `TARGET_VDDPA_1300 = 130`

  1.30V

- `TARGET_VDDPA_1260 = 126`

  1.26V

- `TARGET_VDDPA_1600 = 160`

  1.60V

- `TARGET_VDDPA_MIN_1100 = 110`

  1.10V

- `TARGET_VDDIF_1800 = 180`

  1.80V

- `TARGET_FLASH_1600 = 160`

  1.60V

- `TARGET_RC3 = 3000`

  3MHz

- `TARGET_RC12 = 12000`

  12MHz

- `TARGET_RC24 = 24000`

  24MHz

- `TARGET_RC48 = 48000`

  48MHz

- `TARGET_RC32K = 32768`

  32kHz

- `TARGET_THERMISTOR_10 = 10`

  10uA

- `TARGET_THERMISTOR_5 = 5`

  5.0uA

17.20.4.2 TrimName_t

Location: trim.h:208

Voltage rail and oscillator names.

Members

- `TRIM_BANDGAP`
- `TRIM_DCDC`

  Select loading bandgap trim values.

- `TRIM_VDDC`

  Select loading DCDC trim values.

- `TRIM_VDDM`

  Select loading VDDC trim values.

- `TRIM_VDDRF`

  Select loading VDDM trim values.

- `TRIM_VDDPA`

  Select loading VDDRF trim values.

- `TRIM_VDDIF`

  Select loading VDDPA trim values.

- `TRIM_FLASH`

  Select loading VDDIF trim values.

- `TRIM_RCOSC`

  Select loading VDDFLASH trim values.

- `TRIM_RCOSC32`

  Select loading RC oscillator trim values.

17.20.5 Trimming Support Macro Definition Documentation

17.20.5.1 NULL_POINTER

```
#define NULL_POINTER 0
```

Location: trim.h:54

NULL pointer.

17.20.5.2 MIN_32_BIT

```
#define MIN_32_BIT 0x00000000UL
```

Location: trim.h:58

Minimum 32-bit value.

17.20.5.3 MAX_32_BIT

```
#define MAX_32_BIT 0xFFFFFFFFUL
```

Location: trim.h:61

Maximum 32-bit value.

17.20.5.4 MIN_18_BIT

```
#define MIN_18_BIT 0x00000U
```

Location: trim.h:64

Minimum 18-bit value.

17.20.5.5 MAX_18_BIT

```
#define MAX_18_BIT 0x3FFFFU
```

Location: trim.h:67

Maximum 18-bit value.

17.20.5.6 MIN_16_BIT

```
#define MIN_16_BIT 0x0000U
```

Location: trim.h:70

Minimum 16-bit value.

17.20.5.7 MAX_16_BIT

```
#define MAX_16_BIT 0xFFFFU
```

Location: trim.h:73

Maximum 16-bit value.

17.20.5.8 MIN_8_BIT

```
#define MIN_8_BIT 0x00U
```

Location: trim.h:76

Minimum 8-bit value.

17.20.5.9 MAX_8_BIT

```
#define MAX_8_BIT 0xFFU
```

Location: trim.h:79

Maximum 8-bit value.

17.20.5.10 MAX_4_BIT

```
#define MAX_4_BIT 0xFU
```

Location: trim.h:82

Maximum 4-bit value.

17.20.5.11 ERROR_NO_ERROR

```
#define ERROR_NO_ERROR 0
```

Location: trim.h:86

errors

17.20.5.12 ERROR_NULL

```
#define ERROR_NULL (1 << 1)
```

Location: trim.h:89

Null pointer error.

17.20.5.13 ERROR_NO_TRIM_FOUND

```
#define ERROR_NO_TRIM_FOUND (1 << 3)
```

Location: trim.h:92

Target trim value not found.

17.20.5.14 ERROR_INVALID_TRIM

```
#define ERROR_INVALID_TRIM (1 << 4)
```

Location: trim.h:95

Trims in region specified are not valid.

17.20.5.15 ERROR_INVALID_CRC

```
#define ERROR_INVALID_CRC (1 << 5)
```

Location: trim.h:98

Trim region CRC has failed.

17.20.5.16 ERROR_BG_INVALID

```
#define ERROR_BG_INVALID (1 << 6)
```

Location: trim.h:101

Bandgap target value is invalid.

17.20.5.17 ERROR_BG_V_INVALID

```
#define ERROR_BG_V_INVALID (1 << 7)
```

Location: trim.h:104

Bandgap voltage trim is invalid.

17.20.5.18 ERROR_BG_I_INVALID

```
#define ERROR_BG_I_INVALID (1 << 8)
```

Location: trim.h:107

Bandgap current trim is invalid.

17.20.5.19 ERROR_DCDC_INVALID

```
#define ERROR_DCDC_INVALID (1 << 9)
```

Location: trim.h:110

DCDC trim is invalid.

17.20.5.20 ERROR_VDDC_INVALID

```
#define ERROR_VDDC_INVALID (1 << 10)
```

Location: trim.h:113

VDDC trim is invalid.

17.20.5.21 ERROR_VDDC_STBY_INVALID

```
#define ERROR_VDDC_STBY_INVALID (1 << 11)
```

Location: trim.h:116

VDCC standby trim is invalid.

17.20.5.22 ERROR_VDDM_INVALID

```
#define ERROR_VDDM_INVALID (1 << 12)
```

Location: trim.h:119

VDDM trim is invalid.

17.20.5.23 ERROR_VDDM_STBY_INVALID

```
#define ERROR_VDDM_STBY_INVALID (1 << 13)
```

Location: trim.h:122

VDCM standby trim is invalid.

17.20.5.24 ERROR_VDDRF_INVALID

```
#define ERROR_VDDRF_INVALID (1 << 14)
```

Location: trim.h:125

VDDRF trim is invalid.

17.20.5.25 ERROR_VDDPA_INVALID

```
#define ERROR_VDDPA_INVALID (1 << 15)
```

Location: trim.h:128

VDDPA trim is invalid.

17.20.5.26 ERROR_VDDPA_MIN_INVALID

```
#define ERROR_VDDPA_MIN_INVALID (1 << 16)
```

Location: trim.h:131

VDDPA minimum trim is invalid.

17.20.5.27 ERROR_VDDIF_INVALID

```
#define ERROR_VDDIF_INVALID (1 << 17)
```

Location: trim.h:134

VDDIF trim is invalid.

17.20.5.28 ERROR_VDDFLASH_INVALID

```
#define ERROR_VDDFLASH_INVALID (1 << 18)
```

Location: trim.h:137

VDDFLASH trim is invalid.

17.20.5.29 ERROR_RCOSC_INVALID

```
#define ERROR_RCOSC_INVALID (1 << 19)
```

Location: trim.h:140

RC start oscillator trim is invalid.

17.20.5.30 ERROR_RCOSC32_INVALID

```
#define ERROR_RCOSC32_INVALID (1 << 20)
```

Location: trim.h:143

RC standby oscillator trim is invalid.

17.20.5.31 ERROR_LSAD_INVALID

```
#define ERROR_LSAD_INVALID (1 << 21)
```

Location: trim.h:146

LSAD gain or offset is invalid.

17.20.5.32 ERROR_TEMPERATURE_INVALID

```
#define ERROR_TEMPERATURE_INVALID (1 << 22)
```

Location: trim.h:149

Temperature sensor gain or offset is invalid.

17.20.5.33 ERROR_THERMISTOR_INVALID

```
#define ERROR_THERMISTOR_INVALID (1 << 23)
```

Location: trim.h:152

Thermistor gain or offset is invalid.

17.20.5.34 ERROR_MEASURED_INVALID

```
#define ERROR_MEASURED_INVALID (1 << 25)
```

Location: trim.h:155

Measured reference temperature is invalid.

17.20.5.35 ERROR_TRIM_CUSTOM_SIGNATURE_INVALID

```
#define ERROR_TRIM_CUSTOM_SIGNATURE_INVALID (1 << 26)
```

Location: trim.h:158

Custom signature check is invalid.

17.20.5.36 ERROR_TRIM_CUSTOM_ICH_INVALID

```
#define ERROR_TRIM_CUSTOM_ICH_INVALID (1 << 27)
```

Location: trim.h:161

Custom ICH trim value is invalid.

17.20.5.37 ERROR_TRIM_CUSTOM_XTAL_INVALID

```
#define ERROR_TRIM_CUSTOM_XTAL_INVALID (1 << 28)
```

Location: trim.h:164

Custom Xtal trim value is invalid.

17.20.5.38 TR_REG_TRIM_MASK

```
#define TR_REG_TRIM_MASK 0x3FU
```

Location: trim.h:224

Temperature record 18-bit trim value mask.

17.20.5.39 TRIM_8_BIT_TRIM_MASK

```
#define TRIM_8_BIT_TRIM_MASK 0xFFU
```

Location: trim.h:227

8-bit trim value mask

17.20.5.40 TRIM_16_BIT_TRIM_MASK

```
#define TRIM_16_BIT_TRIM_MASK 0xFFFFU
```

Location: trim.h:230

16-bit trim value mask

17.20.5.41 LSAD_HF

```
#define LSAD_HF 0
```

Location: trim.h:234

LSAD high frequency compensation values.

17.20.5.42 LSAD_LF

```
#define LSAD_LF 1
```

Location: trim.h:237

LSAD low frequency compensation values.

17.20.5.43 LSAD_OFFSET

```
#define LSAD_OFFSET 0x00U
```

Location: trim.h:240

LSAD offset compensation address offset.

17.20.5.44 LSAD_OFFSET_MASK

```
#define LSAD_OFFSET_MASK 0xFFU
```

Location: trim.h:243

LSAD offset compensation mask.

17.20.5.45 LSAD_GAIN

```
#define LSAD_GAIN 0x04U
```

Location: trim.h:246

LSAD gain compensation address offset.

17.20.5.46 LSAD_GAIN_MASK

```
#define LSAD_GAIN_MASK 0x3FFU
```

Location: trim.h:249

LSAD gain compensation mask.

17.20.5.47 TRIM

```
#define TRIM (TRIM_Type *)TRIM_BASE_DEFAULT
```

Location: trim.h:259

Default trim instance, pointing to NVR7.

17.20.5.48 TRIM_SUPPLEMENTAL

```
#define TRIM_SUPPLEMENTAL (TRIM_Type *)FLASH0_NVR4_BASE
```

Location: trim.h:262

Supplemental trim instance, pointing to NVR4.

17.20.5.49 TRIM_CUSTOM_SIP1_SIGNATURE

```
#define TRIM_CUSTOM_SIP1_SIGNATURE 0x53495031
```

Location: trim.h:265

SiP Signature for NVR6 custom trim calibration.

17.20.5.50 TRIM_CUSTOM_CUST_SIGNATURE

```
#define TRIM_CUSTOM_CUST_SIGNATURE 0x43555354
```

Location: trim.h:268

Custom Signature for NVR6 custom trim calibration.

17.20.5.51 SYS_TRIM_LOAD_DEFAULT

```
#define SYS_TRIM_LOAD_DEFAULT Sys_Trim_LoadTrims(TRIM, trim_args1, trim_args2)
```

Location: trim.h:511

Load default trim values from NVR7.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for SYS_TRIM_LOAD_DEFAULT

```
    // Load all valid default trim values from NVR7 for
    // power rails and oscillators
    result = SYS_TRIM_LOAD_DEFAULT();
```

17.20.5.52 SYS_TRIM_LOAD_SUPPLEMENTAL

```
#define SYS_TRIM_LOAD_SUPPLEMENTAL Sys_Trim_LoadTrims(TRIM_SUPPLEMENTAL,x,y)
```

Location: trim.h:527

Load supplemental trim values from NVR4.

Returns:

A code indicating whether an error has occurred. Error codes: errors

NOTE: For detail on input parameters, see trim_args1 & 2 in trim.c

Example Code for SYS_TRIM_LOAD_SUPPLEMENTAL

```
    // Load the requested supplemental trim values
    result = SYS_TRIM_LOAD_DEFAULT();
```

Parameters

Direction	*Name*	Description
in	*x*	uint32_t[6] containing targets in this order: DCDC, VDDRF, VDDIF, VDDFLASH, RC 3MHz, RC 32kHz
in	*y*	uint32_t[4][2] containing targets in this order: Bandgap voltage | Bandgap current VDDC voltage | VDDC standby voltage VDDM voltage | VDDM standby voltage VDDPA voltage | VDDPA minimum voltage

17.20.5.53 SYS_TRIM_LOAD_CUSTOM

```
#define SYS_TRIM_LOAD_CUSTOM Sys_Trim_LoadCustom()
```

Location: trim.h:535

Load custom trim values from NVR6.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for SYS_TRIM_LOAD_CUSTOM

```
    // Load all valid custom trim values from NVR6
    result = SYS_TRIM_LOAD_CUSTOM()
```

17.20.6 Trimming Support Function Documentation

17.20.6.1 Sys_Trim_LoadTrims

```
uint32_t Sys_Trim_LoadTrims(TRIM_Type * trim_region, uint32_t targets_1, uint32_
t targets_2)
```

Location: trim.h:292

Load trim values from the specified memory location.

Returns:

A code indicating whether an error has occurred. Error codes: errors

NOTE: Does not work with LSAD gain/offset. Use lsadload Sys_Trim_GetLSADTrim() instead.

NOTE: For detail on input parameters, see trim_args1 & 2 in trim.c

Example Code for Sys_Trim_LoadTrims

```
    // Load all valid default trim values from MNVR7 for
    // power rails and oscillators
    result = Sys_Trim_LoadTrims(trim_region);
```

Parameters

Direction	Name	Description
`in`	*trim_region*	Pointer to section of memory containing trim values, typically base of NVR7.
`in`	*targets_1*	uint32_t[6] containing targets in this order: DCDC, VDDRF, VDDIF, VDDFLASH, RC 3MHz, RC 32kHz
`in`	*targets_2*	uint32_t[4][2] containing targets in this order: Bandgap voltage | Bandgap current VDDC voltage | VDDC standby voltage VDDM voltage | VDDM standby voltage VDDPA voltage | VDDPA minimum voltage

17.20.6.2 Sys_Trim_LoadSingleTrim

```
uint32_t Sys_Trim_LoadSingleTrim(uint32_t target_name, uint32_t target_value1,
uint32_t target_value2)
```

Location: trim.h:310

Load a trim value for a specific voltage regulator or oscillator.

This function attempts to load calibration values from customer trim settings in NVR4, then load manufacturing calibration values from NVR7 if customer calibration values are not found. target_name Voltage regulator or oscillator to load trim values for. target_value1 Main trim target value target_value2 Secondary trim target value used on some regulators.

Returns:

A code indicating whether an error has occurred. Error codes: errors

NOTE: Does not work with LSAD gain/offset. Use lsadload Sys_Trim_GetLSADTrim() instead.

Example Code for Sys_Trim_LoadSingleTrim

```
// Load all valid default trim values from MNVR7 for
// power rails and oscillators
result = Sys_Trim_LoadTrims(trim_region);
```

Returns:

A code indicating whether an error has occurred. Error codes: errors

NOTE: Does not work with LSAD gain/offset. Use lsadload Sys_Trim_GetLSADTrim() instead.

Example Code for Sys_Trim_LoadSingleTrim

```
// Load all valid default trim values from MNVR7 for
// power rails and oscillators
result = Sys_Trim_LoadTrims(trim_region);
```

Parameters

Direction	Name	Description
`in`	*target_name*	Voltage regulator or oscillator to load trim values for.
`in`	*target_value1*	Main trim target value
`in`	*target_value2*	Secondary trim target value used on some regulators.

17.20.6.3 Sys_Trim_VerifyTrims

```
uint32_t Sys_Trim_VerifyTrims(TRIM_Type * trim_region)
```

Location: trim.h:322

Verify if the trims memory is populated correctly.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_VerifyTrims

```
    // Verify that the input region contains validly proramed values.
    result = Sys_Trim_VerifyTrims(trim_region);
```

Parameters

Direction	Name	Description
`in`	*trim_region*	Pointer to section of memory containing trim values, typically base of NVR7.

17.20.6.4 Sys_Trim_CheckCRC

```
uint32_t Sys_Trim_CheckCRC(TRIM_Type * trim_region)
```

Location: trim.h:332

Check if the CRC for the indicated region is valid.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_CheckCRC

```
    // Performs a CRC on the data in the region storing the trim settings.
    result = Sys_Trim_CheckCRC(trim_region);
```

Parameters

Direction	Name	Description
in	*trim_region*	Pointer to section of memory containing trim values, typically base of NVR7.

17.20.6.5 Sys_Trim_GetTrim

```
 uint32_t Sys_Trim_GetTrim(uint32_t * addr, uint16_t trim_target, uint32_t
record_length, uint16_t * trim_val)
```

Location: trim.h:344

Get the trim value requested, check if it is valid.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_GetTrim

```
    // Retrieves a trim setting from the indicated region, if that target exists.
    result = Sys_Trim_GetTrim(trim_region, TARGET_RC12, 4, &trim_voltage);
```

Parameters

Direction	Name	Description
`in`	*addr*	Pointer to address of base of trim record.
`in`	*trim_target*	Target voltage/current/clock
`in`	*record_length*	Number of records for that trim value.
`out`	*trim_val*	Pointer to return retrieved trim value.

17.20.6.6 Sys_Trim_LoadBandgap

```
 uint32_t Sys_Trim_LoadBandgap(TRIM_Type * trim_values, uint32_t target_v,
uint32_t target_i)
```

Location: trim.h:359

Load target trim value, if present.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_LoadBandgap

```
    // Loads the bandgap trim settings from NVR7.
    result = Sys_Trim_LoadBandgap(trim_region, TARGET_BANDGAP_V, TARGET_BANDGAP_I);
```

Parameters

Direction	Name	Description
`in`	*trim_values*	Pointer to section of memory containing trim values, typically base of NVR7.
`in`	*target_v*	The target voltage trim setting.
`in`	*target_i*	The target current trim setting.

17.20.6.7 Sys_Trim_LoadDCDC

```
uint32_t Sys_Trim_LoadDCDC(TRIM_Type * trim_values, uint32_t target)
```

Location: trim.h:372

Load target trim value for current mode (LDO or BUCK).

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_LoadDCDC

```
    // Loads the DC-DC converter trim settings from NVR7 for 1.2 V.
    result = Sys_Trim_LoadDCDC(trim_region, TARGET_DCDC_1200);
```

Parameters

Direction	Name	Description
`in`	*trim_values*	Pointer to section of memory containing trim values, typically base of NVR7.
`in`	*target*	The target voltage trim setting.

17.20.6.8 Sys_Trim_LoadVDDC

```
uint32_t Sys_Trim_LoadVDDC(TRIM_Type * trim_values, uint32_t target, uint32_t
target_standby)
```

Location: trim.h:384

Load target trim value, if present.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_LoadVDDC

```
    // Loads the VDDC regulator trim settings from NVR7 for 1.15 V.
    result = Sys_Trim_LoadVDDC(trim_region, TARGET_VDDC_1150, TARGET_VDDC_STANDBY);
```

Parameters

Direction	Name	Description
in	*trim_values*	Pointer to section of memory containing trim values, typically base of NVR7.
in	*target*	The target voltage trim setting.
in	*target_standby*	The target standby voltage trim setting.

17.20.6.9 Sys_Trim_LoadVDDM

```
uint32_t Sys_Trim_LoadVDDM(TRIM_Type * trim_values, uint32_t target, uint32_t
target_standby)
```

Location: trim.h:397

Load target trim value, if present.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_LoadVDDM

```
    // Loads the VDDM regulator trim settings from NVR7 for 1.15 V.
    result = Sys_Trim_LoadVDDM(trim_region, TARGET_VDDM_1150, TARGET_VDDM_STANDBY);
```

Parameters

Direction	Name	Description
`in`	*trim_values*	Pointer to section of memory containing trim values, typically base of NVR7.
`in`	*target*	The target voltage trim setting.
`in`	*target_standby*	The target standby voltage trim setting.

17.20.6.10 Sys_Trim_LoadVDDPA

```
uint32_t Sys_Trim_LoadVDDPA(TRIM_Type * trim_values, uint32_t target)
```

Location: trim.h:409

Load target trim value, if present.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_LoadVDDPA

```
    // Loads the VDDPA regulator trim settings from NVR7 for 1.6 V.
    result = Sys_Trim_LoadVDDPA(trim_region, TARGET_VDDPA_1600, TARGET_VDDPA_MIN_1100);
```

Parameters

Direction	Name	Description
`in`	*trim_values*	Pointer to section of memory containing trim values, typically base of NVR7.
`in`	*target*	The target voltage trim setting.

17.20.6.11 Sys_Trim_LoadVDDRF

```
uint32_t Sys_Trim_LoadVDDRF(TRIM_Type * trim_values, uint32_t target)
```

Location: trim.h:420

Load target trim value, if present.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_LoadVDDRF

```
    // Loads the VDDRF regulator trim settings from NVR7 for 1.1 V.
    result = Sys_Trim_LoadVDDRF(trim_region, TARGET_VDDRF_1100);
```

Parameters

Direction	Name	Description
in	*trim_values*	Pointer to section of memory containing trim values, typically base of NVR7.
in	*target*	The target voltage trim setting.

17.20.6.12 Sys_Trim_LoadCustom

```
uint32_t Sys_Trim_LoadCustom()
```

Location: trim.h:428

Load custom trim values from NVR6.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_LoadCustom

```
    // Load all valid custom trim values from NVR6
    result = SYS_TRIM_LOAD_CUSTOM()
```

17.20.6.13 Sys_Trim_LoadVDDFLASH

```
uint32_t Sys_Trim_LoadVDDFLASH(TRIM_Type * trim_values, uint32_t target)
```

Location: trim.h:456

Load target trim value, if present.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_LoadVDDFLASH

```
    // Loads the VDDFLASH regulator trim settings from NVR7 for 1.6 V.
    result = Sys_Trim_LoadVDDFLASH(trim_region, TARGET_VDDFLASH_1600);
```

Parameters

Direction	Name	Description
`in`	*trim_values*	Pointer to section of memory containing trim values, typically base of NVR7.
`in`	*target*	The voltage trim setting desired.

17.20.6.14 Sys_Trim_LoadRCOSC

```
uint32_t Sys_Trim_LoadRCOSC(TRIM_Type * trim_values, uint32_t target)
```

Location: trim.h:467

Load target trim value, if present.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_LoadRCOSC

```
    // Loads the RC Start oscillator trim settings from NVR7 for 12 MHz.
    result = Sys_Trim_LoadRCOSC(trim_region, TARGET_RC12);
```

Parameters

Direction	Name	Description
`in`	*trim_values*	Pointer to section of memory containing trim values, typically base of NVR7.
`in`	*target*	The target clock trim setting.

17.20.6.15 Sys_Trim_LoadRCOSC32

```
uint32_t Sys_Trim_LoadRCOSC32(TRIM_Type * trim_values, uint32_t target)
```

Location: trim.h:478

Load target trim value, if present.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_LoadRCOSC32

```
    // Loads the RC 32768Hz oscillator trim settings from NVR7 for 32768 Hz.
    result = Sys_Trim_LoadRCOSC32(trim_region, TARGET_RC32K);
```

Parameters

Direction	Name	Description
`in`	*trim_values*	Pointer to section of memory containing trim values, typically base of NVR7.
`in`	*target*	The target clock trim setting.

17.20.6.16 Sys_Trim_LoadThermistor

```
uint32_t Sys_Trim_LoadThermistor(TRIM_Type * trim_values, uint16_t target)
```

Location: trim.h:489

Load target trim value, if present.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_LoadThermistor

```
    // Loads the trim settings for the thermistor current source.
    result = Sys_Trim_LoadThermistor(trim_region, TARGET_THERMISTOR);
```

Parameters

Direction	Name	Description
`in`	*trim_values*	Pointer to section of memory containing trim values, typically base of NVR7.
`in`	*target*	The target thermistor trim setting.

17.20.6.17 Sys_Trim_GetLSADTrim

```
 uint32_t Sys_Trim_GetLSADTrim(uint32_t * addr, uint32_t * gain, uint32_t *
offset)
```

Location: trim.h:503

Load LSAD gain and offset value from specified address. Verifies valid values first.

lsadload

Returns:

A code indicating whether an error has occurred. Error codes: errors

NOTE: Assumes format of LSAD gain and offset storage.

Example Code for Sys_Trim_GetLSADTrim

```
    // Loads gain and offset values for the LSAD module from NVR7.
    result = Sys_Trim_GetLSADTrim(trim_region, &gain, &offset);
```

Parameters

Direction	Name	Description
`in`	*addr*	Pointer to memory containing offset in the first word, and gain in the second word.
`out`	*gain*	Pointer to return gain value.
`out`	*offset*	Pointer to return offset value.

17.20.6.18 Sys_Trim_LoadVDDIF

```
uint32_t Sys_Trim_LoadVDDIF(TRIM_Type * trim_values, uint32_t target)
```

Location: trim_vddif.h:49

Load target trim value, if present.

Returns:

A code indicating whether an error has occurred. Error codes: errors

Example Code for Sys_Trim_LoadVDDIF

```
    // Loads the VDDIF regulator trim settings from NVR7 for 1.8 V.
    result = Sys_Trim_LoadVDDIF(trim_region, TARGET_VDDIF_1800);
```

Parameters

Direction	Name	Description
`in`	*trim_values*	Pointer to section of memory containing trim values, typically base of NVR7.
`in`	*target*	The target voltage trim setting.

17.21 UART

Universal Asynchronous Receiver/Transmitter (UART) hardware abstraction layer.

17.21.1 Summary

Macros

- UART_PADS_NUM : The number of input GPIO pad configurations for a UART interface (1 per instance)
- SYS_UART_GPIOCONFIG : Macro wrapper for Sys_UART_GPIOConfig().
- SYS_UART_CONFIG : Macro wrapper for Sys_UART_Config().

Functions

- Sys_UART_GPIOConfig : Configure two GPIOs for the specified UART interface.
- Sys_UART_Config : Configure and enable a UART interface.

17.21.2 UART Macro Definition Documentation

17.21.2.1 UART_PADS_NUM

```
#define UART_PADS_NUM 1
```

Location: uart.h:41

The number of input GPIO pad configurations for a UART interface (1 per instance)

17.21.2.2 SYS_UART_GPIOCONFIG

```
#define SYS_UART_GPIOCONFIG Sys_UART_GPIOConfig(UART, (cfg), (pad_tx), (pad_rx))
```

Location: uart.h:91

Macro wrapper for Sys_UART_GPIOConfig().

Configure two GPIOs for the specified UART interface. cfg GPIO pin configuration for the UART pads pad_tx GPIO to use as the UART transmit pad pad_rx GPIO to use as the UART receive pad

Example Code for SYS_UART_GPIOCONFIG

```
    // Configure GPIOs 5 and 6 for the default UART interface with
    // low-pass filter disabled, 8X drive-strength, and 1 kOhm pull-up resistors
    SYS_UART_GPIOCONFIG((GPIO_LPF_DISABLE | GPIO_1K_PULL_UP  |
                         GPIO_6X_DRIVE), GPIO5, GPIO6);
```

Example Code for SYS_UART_GPIOCONFIG

```
    // Configure GPIOs 5 and 6 for the default UART interface with
    // low-pass filter disabled, 8X drive-strength, and 1 kOhm pull-up resistors
    SYS_UART_GPIOCONFIG((GPIO_LPF_DISABLE | GPIO_1K_PULL_UP  |
                       GPIO_6X_DRIVE), GPIO5, GPIO6);
```

Parameters

Direction	*Name*	Description
in	*cfg*	GPIO pin configuration for the UART pads
in	*pad_tx*	GPIO to use as the UART transmit pad
in	*pad_rx*	GPIO to use as the UART receive pad

17.21.2.3 SYS_UART_CONFIG

```
#define SYS_UART_CONFIG Sys_UART_Config(UART, (uart_clk_hz), (baud), (config))
```

Location: uart.h:107

Macro wrapper for Sys_UART_Config().

Configure and enable a UART interface. uart_clk_hz UART clock speed in hertz baud Baud rate to which UART* is configured config DMA and interrupt mode enable; use UART_TX_DMA_[ENABLE | DISABLE] UART_RX_DMA_[ENABLE | DISABLE] UART_TX_INT_[ENABLE | DISABLE] UART_RX_INT_[ENABLE | DISABLE] UART_OVERRUN_INT_[ENABLE | DISABLE]

Example Code for SYS_UART_CONFIG

```
    // Enable and Configure the default UART:
    //  - 8 MHz clock speed
    //  - 9600 Hz baud rate
    //  - A TX DMA request is generated when new data is
    //    requested by the UART interface
    //  - An RX DMA request is generated when new data is
    //    received by the UART interface
    //  - Interrupts enabled
    SYS_UART_CONFIG(8000000, 9600, (UART_TX_DMA_ENABLE |
                    UART_RX_DMA_ENABLE | UART_TX_START_INT_ENABLE |
                    UART_RX_INT_ENABLE | UART_OVERRUN_INT_ENABLE));
```

Example Code for SYS_UART_CONFIG

```
    // Enable and Configure the default UART:
    //  - 8 MHz clock speed
    //  - 9600 Hz baud rate
    //  - A TX DMA request is generated when new data is
    //    requested by the UART interface
    //  - An RX DMA request is generated when new data is
    //    received by the UART interface
    //  - Interrupts enabled
    SYS_UART_CONFIG(8000000, 9600, (UART_TX_DMA_ENABLE |
                    UART_RX_DMA_ENABLE | UART_TX_START_INT_ENABLE |
                    UART_RX_INT_ENABLE | UART_OVERRUN_INT_ENABLE));
```

Parameters

Direction	*Name*	Description
in	*uart_clk_hz*	UART clock speed in hertz
in	*baud*	Baud rate to which UART* is configured
in	*config*	DMA and interrupt mode enable; use UART_TX_DMA_[ENABLE | DISABLE] UART_RX_DMA_[ENABLE | DISABLE] UART_TX_INT_[ENABLE | DISABLE] UART_RX_INT_[ENABLE | DISABLE] UART_OVERRUN_INT_[ENABLE | DISABLE]

17.21.3 UART Function Documentation

17.21.3.1 Sys_UART_GPIOConfig

```
void Sys_UART_GPIOConfig(const UART_Type * uart, uint32_t cfg, uint32_t pad_tx,
uint32_t pad_rx)
```

Location: uart.h:52

Configure two GPIOs for the specified UART interface.

Example Code for Sys_UART_GPIOConfig

```
// Configure GPIOs 5 and 6 for the UART interface with
// low-pass filter disabled, 8X drive-strength, and 1 kOhm pull-up resistors
Sys_UART_GPIOConfig(UART, (GPIO_LPF_DISABLE | GPIO_1K_PULL_UP |
                   GPIO_8X_DRIVE), GPIO5, GPIO6);
```

Parameters

Direction	Name	Description
`in`	*uart*	Pointer to the UART instance
`in`	*cfg*	GPIO pin configuration for the UART pads
`in`	*pad_tx*	GPIO to use as the UART transmit pad
`in`	*pad_rx*	GPIO to use as the UART receive pad

17.21.3.2 Sys_UART_Config

```
 void Sys_UART_Config(UART_Type * uart, uint32_t uart_clk_hz, uint32_t baud,
uint32_t config)
```

Location: uart.h:80

Configure and enable a UART interface.

Example Code for Sys_UART_Config

```
// Enable and Configure a UART:
//  - 8 MHz clock speed
//  - 9600 Hz baud rate
//  - A TX DMA request is generated when new data is
//    requested by the UART interface
//  - An RX DMA request is generated when new data is
//    received by the UART interface
//  - Interrupts enabled
Sys_UART_Config(UART, 8000000, 9600, (UART_TX_DMA_ENABLE |
                UART_RX_DMA_ENABLE | UART_TX_START_INT_ENABLE |
                UART_RX_INT_ENABLE | UART_OVERRUN_INT_ENABLE));
```

Parameters

Direction	Name	Description
`in`	*uart*	Pointer to the UART instance
`in`	*uart_clk_hz*	UART clock speed in hertz
`in`	*baud*	Baud rate to which UART* is configured
`in`	*config*	DMA and interrupt mode enable; use UART_TX_DMA_[ENABLE | DISABLE] UART_RX_DMA_[ENABLE | DISABLE] UART_TX_INT_[ENABLE | DISABLE] UART_RX_INT_[ENABLE | DISABLE] UART_OVERRUN_INT_[ENABLE | DISABLE]

17.22 WATCHDOG

WATCHDOG Hardware Abstraction Layer.

17.22.1 Summary

Macros

- SYS_WATCHDOG_REFRESH : Refresh the chip and software watchdog timers.
- SYS_WATCHDOG_SOFTWAREREFRESH : Refresh the software watchdog timer count.
- SYS_WATCHDOG_CHIPREFRESH : Refresh the chip watchdog timer count.

17.22.2 WATCHDOG Macro Definition Documentation

17.22.2.1 SYS_WATCHDOG_REFRESH

```
#define SYS_WATCHDOG_REFRESH WATCHDOG->CTRL = WATCHDOG_REFRESH; \
   ACS->SOC_WATCHDOG_CTRL = SOC_WATCHDOG_REFRESH
```

Location: watchdog.h:40

Refresh the chip and software watchdog timers.

17.22.2.2 SYS_WATCHDOG_SOFTWAREREFRESH

```
#define SYS_WATCHDOG_SOFTWAREREFRESH WATCHDOG->CTRL = WATCHDOG_REFRESH
```

Location: watchdog.h:46

Refresh the software watchdog timer count.

17.22.2.3 SYS_WATCHDOG_CHIPREFRESH

```
#define SYS_WATCHDOG_CHIPREFRESH ACS->SOC_WATCHDOG_CTRL = SOC_WATCHDOG_REFRESH
```

Location: watchdog.h:51

Refresh the chip watchdog timer count.

CHAPTER 18

Flash Library Reference

Flash Library Reference.

18.1 Summary

Variables

- FlashLib_Version : Firmware revision code variable.

Enumerations

- FlashStatus_t : Flash library return codes.
- FlashClockFrequency_t : Flash operational frequency values supported by the device.

Macros

- FLASH_FW_VER_MAJOR : Flash library major version number.
- FLASH_FW_VER_MINOR : Flash library minor version number.
- FLASH_FW_VER_REVISION : Flash library revision version number.
- FLASH_FW_VER : Flash library version number, concatenation of all version numbers.
- FLASH0 : Define FLASH0 as the first flash instance, if this is not defined in the headers.
- CODE_ROW_LEN_WORDS : Flash structure definitions.
- CODE_SECTOR_LEN_WORDS : Total number of words in a single sector in Code region.
- CODE_UNLOCK_REGION_LEN_BYTES : Total number of bytes in lock/unlock regions in Code region.
- CODE_UNLOCK_REGION_NUM : Total number of lock/unlock regions in Code region.
- DATA_ROW_LEN_WORDS : Total number of words in a single row in Data region.
- DATA_SECTOR_LEN_WORDS : Total number of words in a single sector in Data region.
- DATA_UNLOCK_REGION_LEN_BYTES : Total number of bytes in lock/unlock regions in Data region.
- DATA_UNLOCK_REGION_NUM : Total number of lock/unlock regions in Data region.
- NVR_ROW_LEN_WORDS : Total number of words in a single row in NVR region.
- NVR_SECTOR_LEN_WORDS : Total number of words in a single sector in NVR region.
- NVR_UNLOCK_REGION_LEN_BYTES : Total number of bytes in lock/unlock regions in NVR region.
- FLASH_INSTANCE_NUM : Total number of flash instances.
- FLASH_0_DESCR_NUM : Total number of descriptor types on flash 0 region example(Code,Data,NVR etc.)
- FLASH_1_DESCR_NUM : Total number of descriptor types on flash 1 region example(Code,Data,NVR etc.)

Functions

- Flash_Initialize : Initialize clock and access to flash.
- Flash_WriteWord : Write a word to a flash address.
- Flash_WriteBuffer : Write contents of a static memory buffer to flash.
- Flash_WriteDouble : Write a 38-bit word to flash.

- Flash_ReadWord : Read a 32-bit word from flash.
- Flash_ReadBuffer : Read contents of flash into a static memory buffer.
- Flash_ReadDouble : Read a 38-bit word from flash.
- Flash_EraseFlashBank : Erase a single flash bank.
- Flash_EraseChip : Erase all data and code flash.
- Flash_EraseSector : Erase a sector flash.
- Flash_BlankCheck : Check if flash region is blank.

18.2 Detailed Description

This reference chapter presents a detailed description of all the functions in the flash programming and erase support library. This reference includes calling parameters, returned values, and assumptions.

Warning: All functions provided by the flash library should be executed from RAM or ROM, as executing them from flash can result in hidden, flash-access-related failures.

18.3 Flash Library Reference Variable Documentation

18.3.1 FlashLib_Version

```
const short FlashLib_Version
```

Location: flash.h:58

Firmware revision code variable.

Access to this variable is available through the ROM tables.

18.4 Flash Library Reference Enumeration Type Documentation

18.4.1 FlashStatus_t

Location: flash.h:105

Flash library return codes.

Members

- `FLASH_ERR_NONE = 0x0`

  Flash no error.

- `FLASH_ERR_BAD_ADDRESS = 0x1`

  Flash error invalid address parameter.

- `FLASH_ERR_BAD_LENGTH = 0x2`

  Flash error invalid word length parameter.

- `FLASH_ERR_INACCESSIBLE = 0x3`

  Flash error flash is inaccessible.

- `FLASH_ERR_INVALID_PARAMS = 0x4`

  Flash error invalid function parameter.

- `FLASH_ERR_NULL_PARAM = 0x5`

  Flash error null pointer used.

- `FLASH_ERR_ADDRESS_WORD_ALIGN = 0x6`

  Flash error address is not word aligned.

- `FLASH_ERR_ZERO_LEN = 0x7`

  Flash error zero length parameter has passed.

- `FLASH_ERR_CRC_CHECK = 0x8`

  Flash error CRC verification has failed.

- `FLASH_ERR_UNKNOWN = 0x9`

  Flash error undefined.

18.4.2 FlashClockFrequency_t

Location: flash.h:122

Flash operational frequency values supported by the device.

Members

- `FLASH_CLOCK_3MHZ = 3000000UL`

  Flash Clock value of 3 MHz.

- `FLASH_CLOCK_4MHZ = 4000000UL`

  Flash Clock value of 4 MHz.

- `FLASH_CLOCK_5MHZ = 5000000UL`

  Flash Clock value of 5 MHz.

- `FLASH_CLOCK_8MHZ = 8000000UL`

  Flash Clock value of 8 MHz.

- `FLASH_CLOCK_10MHZ = 10000000UL`

  Flash Clock value of 10 MHz.

- `FLASH_CLOCK_12MHZ = 12000000UL`

  Flash Clock value of 12 MHz.

- `FLASH_CLOCK_16MHZ = 16000000UL`

  Flash Clock value of 16 MHz.

- `FLASH_CLOCK_20MHZ = 20000000UL`

  Flash Clock value of 20 MHz.

- `FLASH_CLOCK_24MHZ = 24000000UL`

  Flash Clock value of 24 MHz.

- `FLASH_CLOCK_48MHZ = 48000000UL`

  Flash Clock value of 48 MHz.

18.5 Flash Library Reference Macro Definition Documentation

18.5.1 FLASH_FW_VER_MAJOR

```
#define FLASH_FW_VER_MAJOR 0x03
```

Location: flash.h:43

Flash library major version number.

18.5.2 FLASH_FW_VER_MINOR

```
#define FLASH_FW_VER_MINOR 0x00
```

Location: flash.h:46

Flash library minor version number.

18.5.3 FLASH_FW_VER_REVISION

```
#define FLASH_FW_VER_REVISION 0x02
```

Location: flash.h:49

Flash library revision version number.

18.5.4 FLASH_FW_VER

```
#define FLASH_FW_VER ((FLASH_FW_VER_MAJOR << 12) | \
                                  (FLASH_FW_VER_MINOR << 8)  | \
                                  FLASH_FW_VER_REVISION)
```

Location: flash.h:52

Flash library version number, concatenation of all version numbers.

18.5.5 FLASH0

```
#define FLASH0 ((FLASH_Type *)FLASH_BASE)
```

Location: flash.h:62

Define FLASH0 as the first flash instance, if this is not defined in the headers.

18.5.6 CODE_ROW_LEN_WORDS

```
#define CODE_ROW_LEN_WORDS 0x80U
```

Location: flash.h:70

Flash structure definitions.

Total number of words in a single row in Code region

18.5.7 CODE_SECTOR_LEN_WORDS

```
#define CODE_SECTOR_LEN_WORDS 0x200U
```

Location: flash.h:73

Total number of words in a single sector in Code region.

18.5.8 CODE_UNLOCK_REGION_LEN_BYTES

```
#define CODE_UNLOCK_REGION_LEN_BYTES 0x16000U
```

Location: flash.h:76

Total number of bytes in lock/unlock regions in Code region.

18.5.9 CODE_UNLOCK_REGION_NUM

```
#define CODE_UNLOCK_REGION_NUM 0x4U
```

Location: flash.h:79

Total number of lock/unlock regions in Code region.

18.5.10 DATA_ROW_LEN_WORDS

```
#define DATA_ROW_LEN_WORDS 0x20U
```

Location: flash.h:82

Total number of words in a single row in Data region.

18.5.11 DATA_SECTOR_LEN_WORDS

```
#define DATA_SECTOR_LEN_WORDS 0x40U
```

Location: flash.h:85

Total number of words in a single sector in Data region.

18.5.12 DATA_UNLOCK_REGION_LEN_BYTES

```
#define DATA_UNLOCK_REGION_LEN_BYTES 0x5000U
```

Location: flash.h:88

Total number of bytes in lock/unlock regions in Data region.

18.5.13 DATA_UNLOCK_REGION_NUM

```
#define DATA_UNLOCK_REGION_NUM 0x8U
```

Location: flash.h:91

Total number of lock/unlock regions in Data region.

18.5.14 NVR_ROW_LEN_WORDS

```
#define NVR_ROW_LEN_WORDS DATA_ROW_LEN_WORDS
```

Location: flash.h:94

Total number of words in a single row in NVR region.

18.5.15 NVR_SECTOR_LEN_WORDS

```
#define NVR_SECTOR_LEN_WORDS DATA_SECTOR_LEN_WORDS
```

Location: flash.h:97

Total number of words in a single sector in NVR region.

18.5.16 NVR_UNLOCK_REGION_LEN_BYTES

```
#define NVR_UNLOCK_REGION_LEN_BYTES 0x100U
```

Location: flash.h:100

Total number of bytes in lock/unlock regions in NVR region.

18.5.17 FLASH_INSTANCE_NUM

```
#define FLASH_INSTANCE_NUM 0x2U
```

Location: flash_montana.h:30

Total number of flash instances.

18.5.18 FLASH_0_DESCR_NUM

```
#define FLASH_0_DESCR_NUM 0x3U
```

Location: flash_montana.h:33

Total number of descriptor types on flash 0 region example(Code,Data,NVR etc.)

18.5.19 FLASH_1_DESCR_NUM

```
#define FLASH_1_DESCR_NUM FLASH_0_DESCR_NUM
```

Location: flash_montana.h:36

Total number of descriptor types on flash 1 region example(Code,Data,NVR etc.)

18.6 Flash Library Reference Function Documentation

18.6.1 Flash_Initialize

```
FlashStatus_t Flash_Initialize(unsigned int num, FlashClockFrequency_t freq)
```

Location: flash.h:155

Initialize clock and access to flash.

This function powers-up and enables access to a flash region. It also applies the correct delay settings based on the specified flash clock frequency (freq). num Flash instance to be initialized freq Flash clock frequency in Hertz, only defined frequencies supported

See: FlashClockFrequency_t

Returns:

Flash API status code

See: FlashStatus_t

NOTE: System clock frequency should not be changed while the flash is being erased or programmed. An accurate system clock frequency of 1 MHz or higher is required for proper flash operation. If using the RC oscillator, care must be taken as the trimmed frequency for this oscillator has a high temperature dependency.

See: FlashClockFrequency_t

Returns:

Flash API status code

See: FlashStatus_t

NOTE: System clock frequency should not be changed while the flash is being erased or programmed. An accurate system clock frequency of 1 MHz or higher is required for proper flash operation. If using the RC oscillator, care must be taken as the trimmed frequency for this oscillator has a high temperature dependency.

Parameters

Direction	Name	Description
`in`	*num*	Flash instance to be initialized
`in`	*freq*	Flash clock frequency in Hertz, only defined frequencies supported

18.6.2 Flash_WriteWord

```
FlashStatus_t Flash_WriteWord(uint32_t addr, uint32_t word, bool enb_endurance)
```

Location: flash.h:175

Write a word to a flash address.

This function writes a single word to flash. addr Address of the word to be written. word Data to be written to flash. enb_endurance Set to 0 for default flash endurance;
Set to 1 to enable two-stage programming for higher endurance of the data programmed.
Returns:
Flash API status code

See: FlashStatus_t

NOTE: addr must be word aligned.
Contents of flash must be erased prior to performing a write.
Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

Returns:

Flash API status code

See: FlashStatus_t

NOTE: addr must be word aligned.
Contents of flash must be erased prior to performing a write.
Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

Parameters

Direction	Name	Description
`in`	*addr*	Address of the word to be written.
`in`	*word*	Data to be written to flash.
`in`	*enb_endurance*	Set to 0 for default flash endurance; Set to 1 to enable two-stage programming for higher endurance of the data programmed.

18.6.3 Flash_WriteBuffer

```
 FlashStatus_t Flash_WriteBuffer(uint32_t addr, uint32_t word_length, const
uint32_t * words, bool enb_endurance)
```

Location: flash.h:207

Write contents of a static memory buffer to flash.

This function writes the contents of a static memory buffer to flash. A read-back verification is performed after write to ensure the write has been successful. addr Address of first word location in flash. word_length Total number of words to be written to flash. words A 32-bit C pointer to the memory location of
the buffer to be written to flash. enb_endurance Set to 0 for default flash endurance;
Set to 1 to enable two-stage programming for higher endurance of data programmed.
Returns:
Flash API status code

See: FlashStatus_t

NOTE: addr must be word aligned.
Contents of flash must be erased prior to performing a write.
Interrupts are disabled during critical sections, to ensure proper flash operation.
Applications must ensure that the function completes and that the return value is FLASH_ERR_NONE to consider the two-stage programming to be complete.
Source address of data being read and destination address being written, can not be part the same flash instance.
CRC peripheral registers are modified during execution, and restored before returning. The CRC must not be used by the application while writing the buffer to flash.

Returns:

Flash API status code

See: FlashStatus_t

NOTE: addr must be word aligned.
Contents of flash must be erased prior to performing a write.
Interrupts are disabled during critical sections, to ensure proper flash operation.
Applications must ensure that the function completes and that the return value is FLASH_ERR_NONE to consider the two-stage programming to be complete.
Source address of data being read and destination address being written, can not be part the same flash instance.
CRC peripheral registers are modified during execution, and restored before returning. The CRC must not be used by the application while writing the buffer to flash.

Parameters

Direction	Name	Description
in	*addr*	Address of first word location in flash.
in	*word_length*	Total number of words to be written to flash.
in	*words*	A 32-bit C pointer to the memory location of the buffer to be written to flash.
in	*enb_endurance*	Set to 0 for default flash endurance; Set to 1 to enable two-stage programming for higher endurance of data programmed.

18.6.4 Flash_WriteDouble

```
FlashStatus_t Flash_WriteDouble(uint32_t addr, const uint32_t * word, bool enb_
endurance)
```

Location: flash.h:234

Write a 38-bit word to flash.

This function temporarily disables automatic flash ECC generation, allowing the user to write 38-bits to a single word address in flash.
A read-back verification is performed after write to ensure the write has been successful. addr Address of the word to be written in flash. word 32-bit C pointer to the word and ECC data to be written to flash:

- word[0] contains the data to be written to flash
- word[1] [5:0] contains the 6-bit data to be written as the ECC value.

enb_endurance Set to 0 for default flash endurance;
Set to 1 to enable two-stage programming for higher endurance of data programmed.
Returns:
Flash API status code

See: FlashStatus_t

NOTE: addr must be word aligned.
Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

A read-back verification is performed after write to ensure the write has been successful.
Returns:

Flash API status code

See: FlashStatus_t

NOTE: addr must be word aligned.
Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

Parameters

Direction	Name	Description
`in`	*addr*	Address of the word to be written in flash.
`in`	*word*	32-bit C pointer to the word and ECC data to be written to flash: • word[0] contains the data to be written to flash • word[1] [5:0] contains the 6-bit data to be written as the ECC value.
`in`	*enb_endurance*	Set to 0 for default flash endurance; Set to 1 to enable two-stage programming for higher endurance of data programmed.

18.6.5 Flash_ReadWord

```
FlashStatus_t Flash_ReadWord(uint32_t addr, uint32_t * word)
```

Location: flash.h:251

Read a 32-bit word from flash.

This function reads a 32-bit word from flash. If ECC is enabled (default), hardware will log/generate interrupt on ECC errors. addr Address in flash to be read. word 32-bit C pointer to the word read from flash.
Returns:
Flash API status code

See: FlashStatus_t

NOTE: addr must be word aligned.
Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

Returns:

Flash API status code

See: FlashStatus_t

NOTE: addr must be word aligned.
Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

Parameters

Direction	Name	Description
`in`	*addr*	Address in flash to be read.
`out`	*word*	32-bit C pointer to the word read from flash.

18.6.6 Flash_ReadBuffer

```
FlashStatus_t Flash_ReadBuffer(uint32_t flash_addr, uint32_t dram_addr, unsigned
int word_length)
```

Location: flash.h:272

Read contents of flash into a static memory buffer.

This function uses the flash copier to read contents of flash into a memory buffer. flash_addr Address of first word location in flash. dram_addr Address of first word location in static memory. word_length Total number of words to be read from flash.
Returns:
Flash API status code

See: FlashStatus_t

NOTE: flash_addr and dram_addr must be word aligned.
Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.
This function fails if the DMA or CryptoCell continuously blocks memory accesses by the flash copier by accessing memory on every cycle.

Returns:

Flash API status code

See: FlashStatus_t

NOTE: flash_addr and dram_addr must be word aligned.
Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.
This function fails if the DMA or CryptoCell continuously blocks memory accesses by the flash copier by accessing memory on every cycle.

Parameters

Direction	Name	Description
`in`	*flash_addr*	Address of first word location in flash.
`in`	*dram_addr*	Address of first word location in static memory.
`in`	*word_length*	Total number of words to be read from flash.

18.6.7 Flash_ReadDouble

```
FlashStatus_t Flash_ReadDouble(uint32_t addr, uint32_t * word)
```

Location: flash.h:297

Read a 38-bit word from flash.

This function temporarily disables automatic flash ECC generation, allowing the user to read all 38 bits from a single word address in flash.
NOTE: ECC checks are not performed on the 32-bit data word or 6-bit ECC value.
addr Address of the word to be read from flash. word 32-bit C pointer to the word and ECC data read from flash:

- word[0] contains the data to word read from flash.
- word[1] [5:0] contains the 6-bit data ECC value read.

Returns:
Flash API status code

See: FlashStatus_t

NOTE: addr must be word aligned.
Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

NOTE: ECC checks are not performed on the 32-bit data word or 6-bit ECC value.

Returns:

Flash API status code

See: FlashStatus_t

NOTE: addr must be word aligned.
Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

Parameters

Direction	Name	Description
`in`	*addr*	Address of the word to be read from flash.
`out`	*word*	32-bit C pointer to the word and ECC data read from flash: • word[0] contains the data to word read from flash. • word[1] [5:0] contains the 6-bit data ECC value read.

18.6.8 Flash_EraseFlashBank

```
FlashStatus_t Flash_EraseFlashBank(uint32_t num)
```

Location: flash.h:315

Erase a single flash bank.

This function erases all code and data regions of a flash instance. num Flash instance not to be erased.
Returns:
Flash API status code

See: FlashStatus_t

NOTE: A blank check is not performed to ensure that the flash has been successfully erased. Flash_BlankCheck can be used by an application to verify if the erase has been successful.
NVR regions are not erased. Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

Returns:

Flash API status code

See: FlashStatus_t

NOTE: A blank check is not performed to ensure that the flash has been successfully erased. Flash_BlankCheck can be used by an application to verify if the erase has been successful.
NVR regions are not erased. Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

Parameters

Direction	Name	Description
`in`	*num*	Flash instance not to be erased.

18.6.9 Flash_EraseChip

```
FlashStatus_t Flash_EraseChip()
```

Location: flash.h:332

Erase all data and code flash.

This function erases all code and data regions of all flash instances.
Returns:
Flash API status code

See: FlashStatus_t

NOTE: A blank check is not performed to ensure that the flash has been successfully erased. Flash_BlankCheck can be used by an application to verify if the erase has been successful.
NVR regions are not erased.
Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

Returns:

Flash API status code

See: FlashStatus_t

NOTE: A blank check is not performed to ensure that the flash has been successfully erased. Flash_BlankCheck can be used by an application to verify if the erase has been successful.
NVR regions are not erased.
Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

18.6.10 Flash_EraseSector

```
FlashStatus_t Flash_EraseSector(uint32_t addr, bool enb_endurance)
```

Location: flash.h:348

Erase a sector flash.

This function erases a flash sector (512 words for code, 64 words for data). addr An address within the flash sector to be erased. enb_endurance Set to 0 for default flash endurance;
Set to 1 to enable two-stage erase iteration for higher endurance of flash.
Returns:
Flash API status code. See FlashStatus_t

NOTE: Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

Returns:

Flash API status code. See FlashStatus_t

NOTE: Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.

Parameters

Direction	Name	Description
`in`	*addr*	An address within the flash sector to be erased.
`in`	*enb_endurance*	Set to 0 for default flash endurance; Set to 1 to enable two-stage erase iteration for higher endurance of flash.

18.6.11 Flash_BlankCheck

```
FlashStatus_t Flash_BlankCheck(uint32_t addr, unsigned int word_length)
```

Location: flash.h:364

Check if flash region is blank.

This function uses the flash copier in comparator mode to verify if the flash contents are empty (i.e containing the erase value 0xFFFFFFFF). addr Address of the first word in flash to be verified. word_length Total number of words to be verified.

Returns:

Flash API status code FlashStatus_t

NOTE: Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.
addr must be word aligned.

Returns:

Flash API status code FlashStatus_t

NOTE: Interrupts are not disabled; operation is undefined if the calling application modifies the values of flash registers before returning from this function call.
addr must be word aligned.

Parameters

Direction	Name	Description
`in`	*addr*	Address of the first word in flash to be verified.
`in`	*word_length*	Total number of words to be verified.

CHAPTER 19

Supplemental Calibration Library Reference

The supplemental calibration library is used to calibrate each voltage rail or RC oscillator to the desired level or frequency, respectively.

19.1 SUMMARY

Variables

- CalibrateLib_Version : Calibrate version variable.

Data Structures

- CalClock_Type : Contains the quantitative results of the calibration, returned from each clock calibration function.
- CalPower_Type : Contains the quantitative results of the calibration, returned from each rail calibration function.

Enumerations

- clock_check : Enumeration for selecting to check either the 48 MHz crystal or the 32 kHz crystal with Calibrate_Clock_CheckXTAL.

Macros

- CALIBRATE_FW_VER_MAJOR : Major version of Calibration Library.
- CALIBRATE_FW_VER_MINOR : Minor version of Calibration Library.
- CALIBRATE_FW_VER_REVISION : Revision of Calibration Library.
- CALIBRATE_FW_VER : Firmware Calibration Library Version Code.
- ERRNO_POWER_CAL_MARKER : Power supply calibration related errors marker.
- ERRNO_VBG_CAL_ERROR : Error during bandgap calibration process.
- ERRNO_VDDRF_CAL_ERROR : Error during VDDRF calibration process.
- ERRNO_VDDPA_CAL_ERROR : Error during VDDPA calibration process.
- ERRNO_DCDC_CAL_ERROR : Error during DC-DC calibration process.
- ERRNO_VDDC_CAL_ERROR : Error during VDDC calibration process.
- ERRNO_VDDM_CAL_ERROR : Error during VDDM calibration process.
- ERRNO_VDDIF_CAL_ERROR : Error during VDDIF calibration process.
- ERRNO_VDDFLASH_CAL_ERROR : Error during VDDFLASH calibration process.
- ERRNO_STORAGE_CAL_ERROR : Error during storage calibration process.
- ERRNO_CLK_CAL_MARKER : Clock calibration related errors marker.
- ERRNO_RCOSC_CAL_ERROR : Error during RC 32 kHz oscillator calibration process.
- ERRNO_START_OSC_CAL_ERROR : Error during RC Start oscillator calibration process.
- ERRNO_INVALID_MIN_MAX_ERROR : Invalid input to binary search algorithm.

- TRIMMING_STEP : Size of trimming steps for RC oscillator, 1.5%.
- ASYNC_CLK_PERIODS : Number of asynchronous clock periods to measure over.
- CAL_32K_RCOSC : Select to calibrate the 32 kHz RC oscillator.
- CAL_START_OSC : Select to calibrate the Start RC oscillator.
- CAL_RC32OSC_DEFAULT : RC32 OSC default target frequency.
- MHZ_TO_HZ : Conversion factor between SMHz to Hz or vice versa.
- TEN_MS_QUOTIENT : Conversion factor between MHz to Hz or vice versa.
- NON_MONOTONIC_POINTS : Number of points to check when encountering non-monotonic code.
- MONOTONIC_POINTS : Number of points to check when encountering monotonic code.
- NON_MONOTONIC_CODE32 : Number of points to check when encountering monotonic code.
- NON_MONOTONIC_CODE48 : Number of points to check when encountering monotonic code.
- MIN_RCCLK_24 : Minimum frequency of 24 MHz oscillator setting worst case.
- MAX_RCCLK_24 : Maximum frequency of 24 MHz oscillator setting worst case.
- RFCLK_FREQ : RFCLK frequency measured.
- XTAL48_ERROR_LIMIT_MIN : MIN = MIN RCCLK / 8 MHz * 16 ASCC periods.
- XTAL48_ERROR_LIMIT_MAX : MAX = MAX RCCLK / 8 MHz * 16 ASCC periods.
- STANDBYCLK_FREQ : RFCLK frequency measured.
- XTAL32_ERROR_LIMIT_MIN : MIN = MIN RCCLK / 32768 Hz * 16 ASCC periods.
- XTAL32_ERROR_LIMIT_MAX : MAX = MAX RCCLK / 32768 Hz * 16 ASCC periods.
- CONVERT_MHZ_TO_CYCLES : Calculates the number of cycles to be returned by the ASCC. The ASCC returns the number of SYSCLK cycles between n number of periods in the input clock.
- LSAD_STABILIZED_RANGE : LSAD variance in LSBs considered "stable" and not discharging.
- LSAD_MEASUREMENT_ERROR : Set the LSAD measurement error to 5 [mV].
- LSAD_IF_MEASUREMENT_ERROR : Set the LSAD measurement error to 13 [mV].
- LSAD_NUM_CHANNELS : Number of LSAD channels in device.
- VDDCM_TARGET_OFFSET : Offset the desired target to ensure while calibrating.
- BG_VTRIM_0P820V_BYTE : Bandgap maximum safe voltage - 1 trim step.
- VDDM_TRIM_0P95V_BYTE : Digital Memories voltage regulator minimum safe voltage to maintain function.
- VDDFLASH_TRIM_1P500V_BYTE : Flash voltage regulator minimum safe voltage to maintain function.
- VDDFLASH_TRIM_1P850V_BYTE : Flash voltage regulator maximum safe voltage.
- V_TO_MV : Factor for converting back and forth from mV to V.
- V_TO_MV_F : Float iteration of factor for converting back and forth from mV to V.
- CONVERT : Converts an ADC code to a voltage, calculated as follows voltage = adc_code * (2 V * 1000 [mV]/1 V / 2^14 steps.)
- SWAP : Swap the values in variables a and b.

Functions

- Calibrate_Clock_Initialize : Initialize the system to support the clock calibration, consisting of the 48 MHz XTAL oscillator and RC oscillator.
- Calibrate_Clock_32K_RCOSC : Used to calibrate the 32K RC oscillator to a specified frequency.
- Calibrate_Clock_Start_OSC : Used to calibrate the startup oscillator to a specified frequency.
- Calibrate_Clock_CheckXTAL : Used to determine if the specified crystal can oscillate correctly.
- Calibrate_Power_Initialize : Initialize the system to support power supply calibration.
- Calibrate_Power_VDDRF : Calibrate the radio front-end power supply (VDDRF).
- Calibrate_Power_VDDIF : Calibrate the interfaces power supply (VDDIF).
- Calibrate_Power_VDDFLASH : Calibrate the flash power supply (VDDFLASH).
- Calibrate_Power_VDDPA : Calibrate the radio power amplifier power supply (VDDPA).

- Calibrate_Power_DCDC : Calibrate the DC-DC converter (DCDC).
- Calibrate_Power_VDDC : Calibrate the digital core voltage power supply (VDDC).
- Calibrate_Power_VDDM : Calibrate the digital memory voltage (VDDM)

19.2 Supplemental Calibration Library Reference Variable Documentation

19.2.1 CalibrateLib_Version

```
const short CalibrateLib_Version
```

Location: calibrate.h:60

Calibrate version variable.

19.3 Supplemental Calibration Library Reference Data Structures Type Documentation

19.3.1 CalClock_Type

Location: calibrate_clock.h:124

Contains the quantitative results of the calibration, returned from each clock calibration function.

Data Fields

`Type`	*Name*	Description
`uint32_t`	*trim_setting*	The final trim setting in the relevant register.
`uint32_t`	*read_freq*	Last measured frequency at the current final trim setting. Value is in Hz.

19.3.2 CalPower_Type

Location: calibrate_power.h:104

Contains the quantitative results of the calibration, returned from each rail calibration function.

trim_setting the final trim setting in the relevant register. read_voltage last measured voltage at the current final trim setting. Value is in mV.

Data Fields

`Type`	*Name*	Description
`uint32_t`	*trim_setting*	Trim setting result.
`uint32_t`	*read_voltage*	Voltage measured after calibration.

19.4 Supplemental Calibration Library Reference Enumeration Type Documentation

19.4.1 clock_check

Location: calibrate_clock.h:100

Enumeration for selecting to check either the 48 MHz crystal or the 32 kHz crystal with Calibrate_Clock_CheckXTAL.

Members

- `XTAL_48MHZ`
- `XTAL_32KHZ`

19.5 Supplemental Calibration Library Reference Macro Definition Documentation

19.5.1 CALIBRATE_FW_VER_MAJOR

```
#define CALIBRATE_FW_VER_MAJOR 0x02
```

Location: calibrate.h:46

Major version of Calibration Library.

19.5.2 CALIBRATE_FW_VER_MINOR

```
#define CALIBRATE_FW_VER_MINOR 0x01
```

Location: calibrate.h:49

Minor version of Calibration Library.

19.5.3 CALIBRATE_FW_VER_REVISION

```
#define CALIBRATE_FW_VER_REVISION 0x00
```

Location: calibrate.h:52

Revision of Calibration Library.

19.5.4 CALIBRATE_FW_VER

```
#define CALIBRATE_FW_VER ((CALIBRATE_FW_VER_MAJOR << 12) | \
                          (CALIBRATE_FW_VER_MINOR << 8)  | \
                          CALIBRATE_FW_VER_REVISION)
```

Location: calibrate.h:55

Firmware Calibration Library Version Code.

19.5.5 ERRNO_POWER_CAL_MARKER

```
#define ERRNO_POWER_CAL_MARKER 0x10
```

Location: calibrate.h:64

Power supply calibration related errors marker.

19.5.6 ERRNO_VBG_CAL_ERROR

```
#define ERRNO_VBG_CAL_ERROR (0x0001 | ERRNO_POWER_CAL_MARKER)
```

Location: calibrate.h:67

Error during bandgap calibration process.

19.5.7 ERRNO_VDDRF_CAL_ERROR

```
#define ERRNO_VDDRF_CAL_ERROR (0x0002 | ERRNO_POWER_CAL_MARKER)
```

Location: calibrate.h:70

Error during VDDRF calibration process.

19.5.8 ERRNO_VDDPA_CAL_ERROR

```
#define ERRNO_VDDPA_CAL_ERROR (0x0003 | ERRNO_POWER_CAL_MARKER)
```

Location: calibrate.h:73

Error during VDDPA calibration process.

19.5.9 ERRNO_DCDC_CAL_ERROR

```
#define ERRNO_DCDC_CAL_ERROR (0x0004 | ERRNO_POWER_CAL_MARKER)
```

Location: calibrate.h:76

Error during DC-DC calibration process.

19.5.10 ERRNO_VDDC_CAL_ERROR

```
#define ERRNO_VDDC_CAL_ERROR (0x0005 | ERRNO_POWER_CAL_MARKER)
```

Location: calibrate.h:79

Error during VDDC calibration process.

19.5.11 ERRNO_VDDM_CAL_ERROR

```
#define ERRNO_VDDM_CAL_ERROR (0x0006 | ERRNO_POWER_CAL_MARKER)
```

Location: calibrate.h:82

Error during VDDM calibration process.

19.5.12 ERRNO_VDDIF_CAL_ERROR

```
#define ERRNO_VDDIF_CAL_ERROR (0x0007 | ERRNO_POWER_CAL_MARKER)
```

Location: calibrate.h:85

Error during VDDIF calibration process.

19.5.13 ERRNO_VDDFLASH_CAL_ERROR

```
#define ERRNO_VDDFLASH_CAL_ERROR (0x0008 | ERRNO_POWER_CAL_MARKER)
```

Location: calibrate.h:88

Error during VDDFLASH calibration process.

19.5.14 ERRNO_STORAGE_CAL_ERROR

```
#define ERRNO_STORAGE_CAL_ERROR (0x0009 | ERRNO_POWER_CAL_MARKER)
```

Location: calibrate.h:91

Error during storage calibration process.

19.5.15 ERRNO_CLK_CAL_MARKER

```
#define ERRNO_CLK_CAL_MARKER 0x20
```

Location: calibrate.h:94

Clock calibration related errors marker.

19.5.16 ERRNO_RCOSC_CAL_ERROR

```
#define ERRNO_RCOSC_CAL_ERROR (0x0001 | ERRNO_CLK_CAL_MARKER)
```

Location: calibrate.h:97

Error during RC 32 kHz oscillator calibration process.

19.5.17 ERRNO_START_OSC_CAL_ERROR

```
#define ERRNO_START_OSC_CAL_ERROR (0x0002 | ERRNO_CLK_CAL_MARKER)
```

Location: calibrate.h:100

Error during RC Start oscillator calibration process.

19.5.18 ERRNO_INVALID_MIN_MAX_ERROR

```
#define ERRNO_INVALID_MIN_MAX_ERROR (0x0003 | ERRNO_CLK_CAL_MARKER)
```

Location: calibrate.h:103

Invalid input to binary search algorithm.

19.5.19 TRIMMING_STEP

```
#define TRIMMING_STEP 0.015
```

Location: calibrate_clock.h:35

Size of trimming steps for RC oscillator, 1.5%.

19.5.20 ASYNC_CLK_PERIODS

```
#define ASYNC_CLK_PERIODS 16
```

Location: calibrate_clock.h:38

Number of asynchronous clock periods to measure over.

19.5.21 CAL_32K_RCOSC

```
#define CAL_32K_RCOSC 1
```

Location: calibrate_clock.h:41

Select to calibrate the 32 kHz RC oscillator.

19.5.22 CAL_START_OSC

```
#define CAL_START_OSC 2
```

Location: calibrate_clock.h:44

Select to calibrate the Start RC oscillator.

19.5.23 CAL_RC32OSC_DEFAULT

```
#define CAL_RC32OSC_DEFAULT 32768U
```

Location: calibrate_clock.h:47

RC32 OSC default target frequency.

19.5.24 MHZ_TO_HZ

```
#define MHZ_TO_HZ 1000000U
```

Location: calibrate_clock.h:50

Conversion factor between SMHz to Hz or vice versa.

19.5.25 TEN_MS_QUOTIENT

```
#define TEN_MS_QUOTIENT 100U
```

Location: calibrate_clock.h:53

Conversion factor between MHz to Hz or vice versa.

19.5.26 NON_MONOTONIC_POINTS

```
#define NON_MONOTONIC_POINTS 3
```

Location: calibrate_clock.h:56

Number of points to check when encountering non-monotonic code.

19.5.27 MONOTONIC_POINTS

```
#define MONOTONIC_POINTS 1
```

Location: calibrate_clock.h:59

Number of points to check when encountering monotonic code.

19.5.28 NON_MONOTONIC_CODE32

```
#define NON_MONOTONIC_CODE32 32
```

Location: calibrate_clock.h:62

Number of points to check when encountering monotonic code.

19.5.29 NON_MONOTONIC_CODE48

```
#define NON_MONOTONIC_CODE48 48
```

Location: calibrate_clock.h:65

Number of points to check when encountering monotonic code.

19.5.30 MIN_RCCLK_24

```
#define MIN_RCCLK_24 10500000UL
```

Location: calibrate_clock.h:75

Minimum frequency of 24 MHz oscillator setting worst case.

19.5.31 MAX_RCCLK_24

```
#define MAX_RCCLK_24 50000000UL
```

Location: calibrate_clock.h:78

Maximum frequency of 24 MHz oscillator setting worst case.

19.5.32 RFCLK_FREQ

```
#define RFCLK_FREQ 8000000UL
```

Location: calibrate_clock.h:81

RFCLK frequency measured.

19.5.33 XTAL48_ERROR_LIMIT_MIN

```
#define XTAL48_ERROR_LIMIT_MIN MIN_RCCLK_24 / RFCLK_FREQ * ASYNC_CLK_PERIODS
```

Location: calibrate_clock.h:84

MIN = MIN RCCLK / 8 MHz * 16 ASCC periods.

19.5.34 XTAL48_ERROR_LIMIT_MAX

```
#define XTAL48_ERROR_LIMIT_MAX MAX_RCCLK_24 / RFCLK_FREQ * ASYNC_CLK_PERIODS
```

Location: calibrate_clock.h:87

MAX = MAX RCCLK / 8 MHz * 16 ASCC periods.

19.5.35 STANDBYCLK_FREQ

```
#define STANDBYCLK_FREQ 32768U
```

Location: calibrate_clock.h:90

RFCLK frequency measured.

19.5.36 XTAL32_ERROR_LIMIT_MIN

```
#define XTAL32_ERROR_LIMIT_MIN MIN_RCCLK_24 / STANDBYCLK_FREQ * ASYNC_CLK_PERIODS
```

Location: calibrate_clock.h:93

MIN = MIN RCCLK / 32768 Hz * 16 ASCC periods.

19.5.37 XTAL32_ERROR_LIMIT_MAX

```
#define XTAL32_ERROR_LIMIT_MAX MAX_RCCLK_24 / STANDBYCLK_FREQ * ASYNC_CLK_PERIODS
```

Location: calibrate_clock.h:96

MAX = MAX RCCLK / 32768 Hz * 16 ASCC periods.

19.5.38 CONVERT_MHZ_TO_CYCLES

```
#define CONVERT_MHZ_TO_CYCLES (((y) * (z)) / (x))
```

Location: calibrate_clock.h:118

Calculates the number of cycles to be returned by the ASCC. The ASCC returns the number of SYSCLK cycles between n number of periods in the input clock.

Returns:

Number of system clock cycles between n periods of the input clock.

Example Code for CONVERT_MHZ_TO_CYCLES

```
    // Convert a frequency value in MHz to cycles.
    result = CONVERT_MHZ_TO_CYCLES(TARGET_RC32K, \
                                   SystemCoreClock, \
                                   ASYNC_CLK_PERIODS);
```

Parameters

Direction	*Name*	Description
in	*x*	The value in MHz that must be converted to a cycle count
in	*y*	The frequency of the SYSCLK (Hz)
in	*z*	The number of periods that the ASCC measures.

19.5.39 LSAD_STABILIZED_RANGE

```
#define LSAD_STABILIZED_RANGE 10
```

Location: calibrate_power.h:39

LSAD variance in LSBs considered "stable" and not discharging.

19.5.40 LSAD_MEASUREMENT_ERROR

```
#define LSAD_MEASUREMENT_ERROR 5
```

Location: calibrate_power.h:43

Set the LSAD measurement error to 5 [mV].

The trim step size is 10 mV so ideally every value 5 mV apart can be reached.

19.5.41 LSAD_IF_MEASUREMENT_ERROR

```
#define LSAD_IF_MEASUREMENT_ERROR 13
```

Location: calibrate_power.h:47

Set the LSAD measurement error to 13 [mV].

The trim step size for VDDIF/FLASH is 25 mV so ideally every value 12.5 mV apart can be reached, rounded up to 13 mV.

19.5.42 LSAD_NUM_CHANNELS

```
#define LSAD_NUM_CHANNELS 8
```

Location: calibrate_power.h:50

Number of LSAD channels in device.

19.5.43 VDDCM_TARGET_OFFSET

```
#define VDDCM_TARGET_OFFSET 5
```

Location: calibrate_power.h:54

Offset the desired target to ensure while calibrating.

We guarantee that we do not go under the desired target voltage

19.5.44 BG_VTRIM_0P820V_BYTE

```
#define BG_VTRIM_0P820V_BYTE ((uint8_t)(0x2AU << ACS_BG_CTRL_VTRIM_BYTE_Pos))
```

Location: calibrate_power.h:57

Bandgap maximum safe voltage - 1 trim step.

19.5.45 VDDM_TRIM_0P95V_BYTE

```
#define VDDM_TRIM_0P95V_BYTE ((uint8_t)(0x0FU << ACS_VDDM_CTRL_VTRIM_BYTE_Pos))
```

Location: calibrate_power.h:60

Digital Memories voltage regulator minimum safe voltage to maintain function.

19.5.46 VDDFLASH_TRIM_1P500V_BYTE

```
#define VDDFLASH_TRIM_1P500V_BYTE ((uint8_t)(0x1EU << ACS_VDDM_CTRL_VTRIM_BYTE_Pos))
```

Location: calibrate_power.h:63

Flash voltage regulator minimum safe voltage to maintain function.

19.5.47 VDDFLASH_TRIM_1P850V_BYTE

```
#define VDDFLASH_TRIM_1P850V_BYTE ((uint8_t)(0x2CU << ACS_VDDM_CTRL_VTRIM_BYTE_Pos))
```

Location: calibrate_power.h:66

Flash voltage regulator maximum safe voltage.

19.5.48 V_TO_MV

```
#define V_TO_MV 1000
```

Location: calibrate_power.h:69

Factor for converting back and forth from mV to V.

19.5.49 V_TO_MV_F

```
#define V_TO_MV_F 1000.0f
```

Location: calibrate_power.h:72

Float iteration of factor for converting back and forth from mV to V.

19.5.50 CONVERT

```
#define CONVERT ((uint32_t)((x * 1000) >> 13))
```

Location: calibrate_power.h:84

Converts an ADC code to a voltage, calculated as follows voltage = adc_code * (2 V * 1000 [mV]/1 V / 2^14 steps.)

Returns:

The voltage output in mV

Assumptions

Low frequency mode for the ADC is used, meaning that the resolution of the ADC is 14-bits. CONVERT provides voltage level as a milliVolt value based on the input ADC code.

Parameters

Direction	*Name*	Description
in	*x*	the ADC code input

19.5.51 SWAP

```
#define SWAP ((t) = (a), (a) = (b), (b) = (t))
```

Location: calibrate_power.h:95

Swap the values in variables a and b.

Returns:

a holds the value previously in b, b holds the value previously in a.

Parameters

Direction	*Name*	Description
in	*a*	holds the value that must go to b
in	*b*	holds the value that must go to a
in	*t*	a temporary buffer for the swap

19.6 Supplemental Calibration Library Reference Function Documentation

19.6.1 Calibrate_Clock_Initialize

```
void Calibrate_Clock_Initialize()
```

Location: calibrate_clock.h:138

Initialize the system to support the clock calibration, consisting of the 48 MHz XTAL oscillator and RC oscillator.

Example Code for Calibrate_Clock_Initialize

```
    // Initialize the internal oscillator and sets standby clock source to internal
    // 32 kHz oscillator.
    Calibrate_Clock_Initialize();
```

19.6.2 Calibrate_Clock_32K_RCOSC

```
unsigned int Calibrate_Clock_32K_RCOSC(uint32_t target, CalClock_Type * final_
results)
```

Location: calibrate_clock.h:149

Used to calibrate the 32K RC oscillator to a specified frequency.

Returns:

status code indicating whether the RCOSC calibration has succeeded.

Assumptions

Calibrate_Clock_Initialize() has been called.

Example Code for Calibrate_Clock_32K_RCOSC

```
    // Calibrate the interal RC 32768 Hz oscillator.
    result = Calibrate_Clock_32K_RCOSC(TARGET_RC32K, clock_results);
```

Parameters

Direction	Name	Description
`in`	*target*	Number of cycles required to achieve the desired clock frequency in Hz
`in`	*final_results*	Final trim results

19.6.3 Calibrate_Clock_Start_OSC

```
 unsigned int Calibrate_Clock_Start_OSC(uint32_t target, CalClock_Type * final_
results)
```

Location: calibrate_clock.h:162

Used to calibrate the startup oscillator to a specified frequency.

Returns:

Status code indicating whether the clock calibration has succeeded.

Assumptions

Calibrate_Clock_Initialize() has been called.

Assumptions

Standby clock (XTAL32) has been calibrated as close to 32768Hz as possible.

Assumptions

Sets SYSCLK to the RCCLK, not recommended to use while Blue Low Energy is active.

Example Code for Calibrate_Clock_Start_OSC

```
    // Calibrate the interal RC start oscillator to 12 MHz.
    result = Calibrate_Clock_Start_OSC(TARGET_RC12, clock_results);
```

Parameters

Direction	Name	Description
`in`	*target*	desired clock frequency in kHz
`in`	*final_results*	Final trim results

19.6.4 Calibrate_Clock_CheckXTAL

```
uint32_t Calibrate_Clock_CheckXTAL(uint32_t xtal, uint32_t gpio)
```

Location: calibrate_clock.h:174

Used to determine if the specified crystal can oscillate correctly.

Returns:

Status code indicating whether the selected crystal is oscillating correctly. 1 indicates success, 0 indicates failure.

Assumptions

Sets SYSCLK to the RCCLK, not recommended to use while Blue Low Energy is active.

Example Code for Calibrate_Clock_CheckXTAL

```
    // Check that the 48MHz crystal is oscillating correctly.
    result = Calibrate_Clock_CheckXTAL(XTAL_48M, GPIO1);
```

Parameters

Direction	Name	Description
in	*xtal*	The desired crystal to be checked, use XTAL_[48M | 32K]HZ
in	*gpio*	GPIO that the selected clock will be output on

19.6.5 Calibrate_Power_Initialize

```
void Calibrate_Power_Initialize()
```

Location: calibrate_power.h:122

Initialize the system to support power supply calibration.

1. Changes settings in all power supply control registers to their default values.
2. Sets the system clock source to RFCLK/3 (16 MHz).
3. Configures the ADC to enable measurement at 625 Hz
   Assumptions

   VBAT must be equal to 1.5V if calibrating bandgap.

Example Code for Calibrate_Power_Initialize

```
    // Initialze LSAD for measuring voltage rails, turn off automatic compensatio
n
    Calibrate_Power_Initialize();
```

19.6.6 Calibrate_Power_VDDRF

```
unsigned int Calibrate_Power_VDDRF(unsigned int adc_num, const volatile uint32_t
* adc_ptr, uint32_t target, CalPower_Type * final_trim)
```

Location: calibrate_power.h:137

Calibrate the radio front-end power supply (VDDRF).

Returns:

Status code indicating whether the calibration has succeeded

Assumptions

VBG has been calibrated.

Assumptions

Calibrate_Power_Initialize() has been called.

Assumptions

VCC is sufficiently high to trim VDDRF to the desired value. This is because VCC supplies VDDRF.

Example Code for Calibrate_Power_VDDRF

```
    // Calibrate VDDRF to 1100 mV.
    result = Calibrate_Power_VDDRF(LSAD_CALIB_CHANNEL,
                &(LSAD->DATA_TRIM_CH[LSAD_CALIB_CHANNEL]),
                TARGET_VDDRF_1100,
                pwr_results);
```

Parameters

Direction	Name	Description
`in`	*adc_num*	ADC channel number [0-7]
`in`	*adc_ptr*	Pointer to the ADC data register
`in`	*target*	Target voltage readback [10*mV]
`out`	*final_trim*	Result of calibration, trim setting & voltage (mV)

19.6.7 Calibrate_Power_VDDIF

```
unsigned int Calibrate_Power_VDDIF(unsigned int adc_num, const volatile uint32_t
* adc_ptr, uint32_t target, CalPower_Type * final_trim)
```

Location: calibrate_power.h:155

Calibrate the interfaces power supply (VDDIF).

Returns:

Status code indicating whether the calibration has succeeded

Assumptions

VBG has been calibrated.

Assumptions

Calibrate_Power_Initialize() has been called.

Assumptions

VDDCP is sufficiently high to trim VDDIF to the desired value. This is because VDDCP supplies VDDIF.

Example Code for Calibrate_Power_VDDIF

```
    // Calibrate VDDIF to 1800 mV.
    result = Calibrate_Power_VDDIF(LSAD_CALIB_CHANNEL,
                &(LSAD->DATA_TRIM_CH[LSAD_CALIB_CHANNEL]),
                TARGET_VDDIF_1800,
                pwr_results);
```

Parameters

Direction	Name	Description
`in`	*adc_num*	ADC channel number [0-7]
`in`	*adc_ptr*	Pointer to the ADC data register
`in`	*target*	Target voltage readback [10*mV]
`out`	*final_trim*	Result of calibration, trim setting & voltage (mV)

19.6.8 Calibrate_Power_VDDFLASH

```
 unsigned int Calibrate_Power_VDDFLASH(unsigned int adc_num, const volatile
uint32_t * adc_ptr, uint32_t target, CalPower_Type * final_trim)
```

Location: calibrate_power.h:173

Calibrate the flash power supply (VDDFLASH).

Returns:

Status code indicating whether the calibration has succeeded

Assumptions

VBG has been calibrated.

Assumptions

Calibrate_Power_Initialize() has been called.

Assumptions

VDDCP is sufficiently high to trim VDDFLASH to the desired value. This is because VDDCP supplies VDDFLASH.

Example Code for Calibrate_Power_VDDFLASH

```
    // Calibrate VDDFLASH to 1600 mV.
    result = Calibrate_Power_VDDFLASH(LSAD_CALIB_CHANNEL,
                &(LSAD->DATA_TRIM_CH[LSAD_CALIB_CHANNEL]),
                TARGET_VDDFLASH_1600,
                pwr_results);
```

Parameters

Direction	Name	Description
in	*adc_num*	ADC channel number [0-7]
in	*adc_ptr*	Pointer to the ADC data register
in	*target*	Target voltage readback [10*mV]
out	*final_trim*	Result of calibration, trim setting & voltage (mV)

19.6.9 Calibrate_Power_VDDPA

```
unsigned int Calibrate_Power_VDDPA(unsigned int adc_num, const volatile uint32_t
* adc_ptr, uint32_t target, CalPower_Type * final_trim)
```

Location: calibrate_power.h:189

Calibrate the radio power amplifier power supply (VDDPA).

Returns:

Status code indicating whether the calibration has succeeded

Assumptions

VBG has been calibrated.

Assumptions

Calibrate_Power_Initialize() has been called.

Example Code for Calibrate_Power_VDDPA

```
    // Calibrate VDDPA to 1600 mV.
    result = Calibrate_Power_VDDPA(LSAD_CALIB_CHANNEL,
                  &(LSAD->DATA_TRIM_CH[LSAD_CALIB_CHANNEL]),
                  TARGET_VDDPA_1600,
                  pwr_results);
```

Parameters

Direction	Name	Description
in	*adc_num*	ADC channel number [0-7]
in	*adc_ptr*	Pointer to the ADC data register
in	*target*	Target voltage readback [10*mV]
out	*final_trim*	Result of calibration, trim setting & voltage (mV)

19.6.10 Calibrate_Power_DCDC

```
unsigned int Calibrate_Power_DCDC(unsigned int adc_num, const volatile uint32_t
* adc_ptr, uint32_t target, CalPower_Type * final_trim)
```

Location: calibrate_power.h:207

Calibrate the DC-DC converter (DCDC).

Returns:

Status code indicating whether the calibration has succeeded

Assumptions

VBG has been calibrated.

Assumptions

User is responsible for selecting LDO or BUCK mode for the DCDC converter

Assumptions

Calibrate_Power_Initialize() has been called.

Example Code for Calibrate_Power_DCDC

```
    // Calibrate DCDC to 1200 mV.
    result = Calibrate_Power_DCDC(LSAD_CALIB_CHANNEL,
                &(LSAD->DATA_TRIM_CH[LSAD_CALIB_CHANNEL]),
                TARGET_DCDC_1200,
                pwr_results);
```

Parameters

Direction	Name	Description
`in`	*adc_num*	ADC channel number [0-7]
`in`	*adc_ptr*	Pointer to the ADC data register
`in`	*target*	Target voltage readback [10*mV]
`out`	*final_trim*	Result of calibration, trim setting & voltage (mV)

19.6.11 Calibrate_Power_VDDC

```
unsigned int Calibrate_Power_VDDC(unsigned int adc_num, const volatile uint32_t
* adc_ptr, uint32_t target, CalPower_Type * final_trim)
```

Location: calibrate_power.h:223

Calibrate the digital core voltage power supply (VDDC).

Returns:

Status code indicating whether the calibration has succeeded

Assumptions

VBG has been calibrated.

Assumptions

Calibrate_Power_Initialize() has been called.

Example Code for Calibrate_Power_VDDC

```
    // Calibrate VDDC to 1150 mV.
    result = Calibrate_Power_VDDC(LSAD_CALIB_CHANNEL,
                &(LSAD->DATA_TRIM_CH[LSAD_CALIB_CHANNEL]),
                TARGET_VDDC_1150,
                pwr_results);
```

Parameters

Direction	Name	Description
`in`	*adc_num*	ADC channel number [0-7]
`in`	*adc_ptr*	Pointer to the ADC data register
`in`	*target*	Target voltage readback [10*mV]
`out`	*final_trim*	Result of calibration, trim setting & voltage (mV)

19.6.12 Calibrate_Power_VDDM

```
unsigned int Calibrate_Power_VDDM(unsigned int adc_num, const volatile uint32_t
* adc_ptr, uint32_t target, CalPower_Type * final_trim)
```

Location: calibrate_power.h:239

Calibrate the digital memory voltage (VDDM)

Returns:

Status code indicating whether the calibration has succeeded

Assumptions

VBG has been calibrated.

Assumptions

Calibrate_Power_Initialize() has been called.

Example Code for Calibrate_Power_VDDM

```
    // Calibrate VDDM to 1150 mV.
    result = Calibrate_Power_VDDM(LSAD_CALIB_CHANNEL,
                 &(LSAD->DATA_TRIM_CH[LSAD_CALIB_CHANNEL]),
                 TARGET_VDDM_1150,
                 pwr_results);
```

Parameters

Direction	Name	Description
`in`	*adc_num*	ADC channel number [0-7]
`in`	*adc_ptr*	Pointer to the ADC data register
`in`	*target*	Target voltage readback [10*mV]
`out`	*final_trim*	Result of calibration, trim setting & voltage (mV)

CHAPTER 20

CMSIS Drivers Reference

CMSIS Drivers Reference.

20.1 Summary

Typedefs

- ARM_DRIVER_VERSION : Driver Version.
- ARM_POWER_STATE : General power states.

Data Structures

- _ARM_DRIVER_VERSION : Driver Version.

Enumerations

- _ARM_POWER_STATE : General power states.

Macros

- ARM_DRIVER_VERSION_MAJOR_MINOR : Driver API Version.
- ARM_DRIVER_OK : Operation succeeded.
- ARM_DRIVER_ERROR : Unspecified error.
- ARM_DRIVER_ERROR_BUSY : Driver is busy.
- ARM_DRIVER_ERROR_TIMEOUT : Timeout occurred.
- ARM_DRIVER_ERROR_UNSUPPORTED : Operation not supported.
- ARM_DRIVER_ERROR_PARAMETER : Parameter error.
- ARM_DRIVER_ERROR_SPECIFIC : Start of driver specific errors.

20.2 CMSIS Drivers Reference Typedef Documentation

20.2.1 ARM_DRIVER_VERSION

```
typedef struct _ARM_DRIVER_VERSION ARM_DRIVER_VERSION
```

Location: Driver_Common.h:62

Driver Version.

20.2.2 ARM_POWER_STATE

```
typedef enum _ARM_POWER_STATE ARM_POWER_STATE
```

Location: Driver_Common.h:80

General power states.

20.3 CMSIS Drivers Reference Data Structures Type Documentation

20.3.1 _ARM_DRIVER_VERSION

Location: Driver_Common.h:59

Driver Version.

Data Fields

`Type`	*Name*	Description
`uint16_t`	*api*	API version.
`uint16_t`	*drv*	Driver version.

20.4 CMSIS Drivers Reference Enumeration Type Documentation

20.4.1 _ARM_POWER_STATE

Location: Driver_Common.h:76

General power states.

Members

- `ARM_POWER_OFF`

  Power off: No operation possible.

- `ARM_POWER_LOW`

  Low Power mode: Retain state, detect and signal wake-up events.

- `ARM_POWER_FULL`

  Power on: Full operation at maximum performance.

20.5 CMSIS Drivers Reference Macro Definition Documentation

20.5.1 ARM_DRIVER_VERSION_MAJOR_MINOR

```
#define ARM_DRIVER_VERSION_MAJOR_MINOR (((major) << 8) | (minor))
```

Location: Driver_Common.h:54

Driver API Version.

20.5.2 ARM_DRIVER_OK

```
#define ARM_DRIVER_OK 0
```

Location: Driver_Common.h:65

Operation succeeded.

General return codes

20.5.3 ARM_DRIVER_ERROR

```
#define ARM_DRIVER_ERROR -1
```

Location: Driver_Common.h:66

Unspecified error.

20.5.4 ARM_DRIVER_ERROR_BUSY

```
#define ARM_DRIVER_ERROR_BUSY -2
```

Location: Driver_Common.h:67

Driver is busy.

20.5.5 ARM_DRIVER_ERROR_TIMEOUT

```
#define ARM_DRIVER_ERROR_TIMEOUT -3
```

Location: Driver_Common.h:68

Timeout occurred.

20.5.6 ARM_DRIVER_ERROR_UNSUPPORTED

```
#define ARM_DRIVER_ERROR_UNSUPPORTED -4
```

Location: Driver_Common.h:69

Operation not supported.

20.5.7 ARM_DRIVER_ERROR_PARAMETER

```
#define ARM_DRIVER_ERROR_PARAMETER -5
```

Location: Driver_Common.h:70

Parameter error.

20.5.8 ARM_DRIVER_ERROR_SPECIFIC

```
#define ARM_DRIVER_ERROR_SPECIFIC -6
```

Location: Driver_Common.h:71

Start of driver specific errors.

20.6 CMSIS DMA Driver

CMSIS DMA Driver Reference.

20.6.1 Summary

Typedefs

- DMA_SEL_t : Selects the DMA channel.
- DMA_TRG_t : Selects the DMA src/dst target interface.
- DMA_SRC_STEP_t : Selects the step size increment to the DMA channel source address.
- DMA_DST_STEP_t : Selects the step size increment to the DMA channel destination address.
- DMA_SRC_DST_TRANS_LENGHT_SEL_t : Selects whether the transfer length counter depends on either the source word counts or the destination word count.
- DMA_DATA_MODE_t : Selects how often data is transferred.
- DMA_BYTE_ORDER_t : Selects the order of the data bytes.
- DMA_WORD_SIZE_t : Selects the src/dst data word size.
- DMA_CH_PRI_t : Selects priority of DMA channels.
- ADC_EVENT_SRC_t : Selects DMA interrupt channel source.
- DMA_SignalEvent_t : Pointer to DMA_SignalEvent : Signal Timer event.
- DMA_CFG_t : DMA channel configuration.
- DMA_ADDR_CFG_t : DMA src/dst address configuration.
- DMA_PRI_CFG_t : DMA interrupt priority configuration.
- DMA_STATUS_t : DMA status.
- DRIVER_DMA_t : Access structure of the DMA Driver.

Data Structures

- _DMA_CFG_t : DMA channel configuration.
- _DMA_ADDR_CFG_t : DMA src/dst address configuration.
- _DMA_PRI_CFG_t : DMA interrupt priority configuration.
- _DMA_STATUS_t : DMA status.
- _DRIVER_DMA_t : Access structure of the DMA Driver.

Enumerations

- _DMA_SEL_t : Selects the DMA channel.
- _DMA_TRG_t : Selects the DMA src/dst target interface.
- _DMA_SRC_STEP_t : Selects the step size increment to the DMA channel source address.
- _DMA_DST_STEP_t : Selects the step size increment to the DMA channel destination address.
- _DMA_SRC_DST_TRANS_LENGHT_SEL_t : Selects whether the transfer length counter depends on either the source word counts or the destination word count.
- _DMA_DATA_MODE_t : Selects how often data is transferred.
- _DMA_BYTE_ORDER_t : Selects the order of the data bytes.
- _DMA_WORD_SIZE_t : Selects the src/dst data word size.
- _DMA_CH_PRI_t : Selects priority of DMA channels.
- _ADC_EVENT_SRC_t : Selects DMA interrupt channel source.

Macros

- ARM_DMA_API_VERSION : DMA API version.
- DMA_ERROR_UNCONFIGURED : DMA channel has not been configured yet.

Functions

- DMA_GetVersion : Get driver version.
- DMA_Initialize : Initialize DMA driver with default configuration.
- DMA_Configure : Configure particular DMA channel.
- DMA_ConfigureWord : Configure particular DMA channel.
- DMA_ConfigureAddr : Configure DMA channel source and destination addresses.
- DMA_SetInterruptPriority : Configure the DMA interrupt priority.
- DMA_CreateConfigWord : Create DMA channel configuration word.
- DMA_SetConfigWord : Quickly updates the DMA channel configuration.
- DMA_Start : Starts the DMA transfer.
- DMA_Stop : Stops the DMA transfer.
- DMA_GetCounterValue : Returns the current counter value of DMA channel.
- DMA_GetStatus : Returns the DMA channel status. Clears the status register on read.
- DMA_SignalEvent : Signal DMA events.

20.6.2 CMSIS DMA Driver Typedef Documentation

20.6.2.1 DMA_SEL_t

```
typedef enum _DMA_SEL_t DMA_SEL_t
```

Location: Driver_DMA.h:49

Selects the DMA channel.

20.6.2.2 DMA_TRG_t

```
typedef enum _DMA_TRG_t DMA_TRG_t
```

Location: Driver_DMA.h:63

Selects the DMA src/dst target interface.

20.6.2.3 DMA_SRC_STEP_t

```
typedef enum _DMA_SRC_STEP_t DMA_SRC_STEP_t
```

Location: Driver_DMA.h:85

Selects the step size increment to the DMA channel source address.

20.6.2.4 DMA_DST_STEP_t

```
typedef enum _DMA_DST_STEP_t DMA_DST_STEP_t
```

Location: Driver_DMA.h:107

Selects the step size increment to the DMA channel destination address.

20.6.2.5 DMA_SRC_DST_TRANS_LENGHT_SEL_t

```
typedef enum _DMA_SRC_DST_TRANS_LENGHT_SEL_t DMA_SRC_DST_TRANS_LENGHT_SEL_t
```

Location: Driver_DMA.h:116

Selects whether the transfer length counter depends on either the source word counts or the destination word count.

20.6.2.6 DMA_DATA_MODE_t

```
typedef enum _DMA_DATA_MODE_t DMA_DATA_MODE_t
```

Location: Driver_DMA.h:124

Selects how often data is transferred.

20.6.2.7 DMA_BYTE_ORDER_t

```
typedef enum _DMA_BYTE_ORDER_t DMA_BYTE_ORDER_t
```

Location: Driver_DMA.h:132

Selects the order of the data bytes.

20.6.2.8 DMA_WORD_SIZE_t

```
typedef enum _DMA_WORD_SIZE_t DMA_WORD_SIZE_t
```

Location: Driver_DMA.h:161

Selects the src/dst data word size.

20.6.2.9 DMA_CH_PRI_t

```
typedef enum _DMA_CH_PRI_t DMA_CH_PRI_t
```

Location: Driver_DMA.h:171

Selects priority of DMA channels.

20.6.2.10 ADC_EVENT_SRC_t

```
typedef enum _ADC_EVENT_SRC_t ADC_EVENT_SRC_t
```

Location: Driver_DMA.h:181

Selects DMA interrupt channel source.

20.6.2.11 DMA_SignalEvent_t

```
typedef void(* DMA_SignalEvent_t
```

Location: Driver_DMA.h:261

Pointer to DMA_SignalEvent : Signal Timer event.

20.6.2.12 DMA_CFG_t

```
typedef struct _DMA_CFG_t DMA_CFG_t
```

Location: Driver_DMA.h:275

DMA channel configuration.

20.6.2.13 DMA_ADDR_CFG_t

```
typedef struct _DMA_ADDR_CFG_t DMA_ADDR_CFG_t
```

Location: Driver_DMA.h:286

DMA src/dst address configuration.

20.6.2.14 DMA_PRI_CFG_t

```
typedef struct _DMA_PRI_CFG_t DMA_PRI_CFG_t
```

Location: Driver_DMA.h:297

DMA interrupt priority configuration.

20.6.2.15 DMA_STATUS_t

```
typedef struct _DMA_STATUS_t DMA_STATUS_t
```

Location: Driver_DMA.h:309

DMA status.

20.6.2.16 DRIVER_DMA_t

```
typedef struct _DRIVER_DMA_t DRIVER_DMA_t
```

Location: Driver_DMA.h:329

Access structure of the DMA Driver.

20.6.3 CMSIS DMA Driver Data Structures Type Documentation

20.6.3.1 _DMA_CFG_t

Location: Driver_DMA.h:266

DMA channel configuration.

Data Fields

Type	*Name*	Description
DMA_TRG_t	*src_sel*	DMA source target.
DMA_SRC_STEP_t	*src_step*	Source step mode.
DMA_WORD_SIZE_t	*word_size*	Source word size.
DMA_TRG_t	*dst_sel*	DMA destination target.
DMA_DST_STEP_t	*dst_step*	Destination step mode.
DMA_CH_PRI_t	*ch_priority*	Channel priority.
uint32_t	*__pad0__*	Reserved.

20.6.3.2 _DMA_ADDR_CFG_t

Location: Driver_DMA.h:280

DMA src/dst address configuration.

Data Fields

Type	Name	Description
`const void *`	*src_addr*	Source address.
`const void *`	*dst_addr*	Destination address.
`uint32_t`	*counter_len*	Value which when reached triggers the counter event.
`uint32_t`	*transfer_len*	DMA transfer length.

20.6.3.3 _DMA_PRI_CFG_t

Location: Driver_DMA.h:291

DMA interrupt priority configuration.

Data Fields

Type	Name	Description
`uint32_t`	*preempt_pri*	Preempt priority.
`uint32_t`	*__pad0__*	Reserved.
`uint32_t`	*subgrp_pri*	Subgroup priority.
`uint32_t`	*__pad1__*	Reserved.

20.6.3.4 _DMA_STATUS_t

Location: Driver_DMA.h:302

DMA status.

Data Fields

Type	Name	Description
`uint32_t`	*active*	Transfer was started.
`uint32_t`	*completed*	Transfer was completed.

uint32_t	*counter_reached*	Counter value was reached.
uint32_t	*buffer_fill_lvl*	Error occurred.
uint32_t	*__pad0__*	Reserved.

20.6.3.5 _DRIVER_DMA_t

Location: Driver_DMA.h:314

Access structure of the DMA Driver.

Data Fields

Type	*Name*	Description
ARM_DRIVER_VERSION(*	*GetVersion)(void)*	Pointer to DMA_GetVersion : Get driver version.
int32_t(*	*Initialize)(DMA_SignalEvent_t cb_event)*	Pointer to DMA_Initialize : Initialize DMA driver.
int32_t(*	*Configure)(DMA_SEL_t sel, const DMA_CFG_t *cfg, DMA_SignalEvent_t cb)*	Pointer to DMA_Configure : Configure DMA channel.
int32_t(*	*ConfigureWord)(DMA_SEL_t sel, uint32_t cfg, DMA_SignalEvent_t cb)*	Pointer to DMA_ConfigureWord : Configure DMA channel.
int32_t(*	*ConfigureAddr)(DMA_SEL_t sel, const DMA_ADDR_CFG_t *pri)*	Pointer to DMA_ConfigureAddr : Configure DMA interrupt priority.
int32_t(*	*SetInterruptPriority)(DMA_SEL_t sel, const DMA_PRI_CFG_t *pri)*	Pointer to DMA_SetInterruptPriority : Configure DMA interrupt priority.
uint32_t(*	*CreateConfigWord)(const DMA_CFG_t *cfg)*	Pointer to DMA_CreateConfigWord : Create DMA channel configuration word.
void(*	*SetConfigWord)(DMA_SEL_t sel, uint32_t cfg)*	Pointer to DMA_SetConfigWord : Quickly update DMA channel configuration word.

`int32_t(*`	*Start)(DMA_SEL_t sel)*	Pointer to DMA_Start : Start DMA transfer.
`int32_t(*`	*Stop)(DMA_SEL_t sel)*	Pointer to DMA_Stop : Stop DMA transfer.
`uint32_t(*`	*GetCounterValue)(DMA_SEL_t sel)*	Pointer to DMA_GetCounterValue : Get the current channel transfer counter.
`DMA_STATUS_t(*`	*GetStatus)(DMA_SEL_t sel)*	Pointer to DMA_GetStatus : Returns DMA channel status.

20.6.4 CMSIS DMA Driver Enumeration Type Documentation

20.6.4.1 _DMA_SEL_t

Location: Driver_DMA.h:44

Selects the DMA channel.

Members

- `DMA_CH_0 = 0`

  DMA channel 0.

- `DMA_CH_1 = 1`

  DMA channel 1.

- `DMA_CH_2 = 2`

  DMA channel 2.

- `DMA_CH_3 = 3`

  DMA channel 3.

20.6.4.2 _DMA_TRG_t

Location: Driver_DMA.h:54

Selects the DMA src/dst target interface.

Members

- `DMA_TRG_MEM = 0`
- `DMA_TRG_SPI0 = 1`

  Source / destination target = SPI0.
- `DMA_TRG_SPI1 = 2`

  Source / destination target = SPI1.
- `DMA_TRG_I2C0 = 3`

  Source / destination target = I2C0.
- `DMA_TRG_I2C1 = 4`

  Source / destination target = I2C1.
- `DMA_TRG_UART = 5`

  Source / destination target = UART.
- `DMA_TRG_PCM = 6`

  Source / destination target = PCM.

- `DMA_TRG_TOF = 7`

  Source / destination target = TOF.

20.6.4.3 _DMA_SRC_STEP_t

Location: Driver_DMA.h:68

Selects the step size increment to the DMA channel source address.

Members

- `DMA_CFG0_SRC_ADDR_STATIC = 0x00`

  Do not increment the source address used by DMA channel.

- `DMA_CFG0_SRC_ADDR_INCR_1 = 0x01`

  Set the step size of DMA channel source address to 1.

- `DMA_CFG0_SRC_ADDR_INCR_2 = 0x02`

Set the step size of DMA channel source address to 2.

- `DMA_CFG0_SRC_ADDR_INCR_3 = 0x03`

  Set the step size of DMA channel source address to 3.

- `DMA_CFG0_SRC_ADDR_INCR_4 = 0x04`

  Set the step size of DMA channel source address to 4.

- `DMA_CFG0_SRC_ADDR_INCR_5 = 0x05`

  Set the step size of DMA channel source address to 5.

- `DMA_CFG0_SRC_ADDR_INCR_6 = 0x06`

  Set the step size of DMA channel source address to 6.

- `DMA_CFG0_SRC_ADDR_INCR_7 = 0x07`

  Set the step size of DMA channel source address to 7.

- `DMA_CFG0_SRC_ADDR_DECR_8 = 0x08`

  Set the step size of DMA channel source address to negative 8.

- `DMA_CFG0_SRC_ADDR_DECR_7 = 0x09`

  Set the step size of DMA channel source address to negative 7.

- `DMA_CFG0_SRC_ADDR_DECR_6 = 0x0A`

  Set the step size of DMA channel source address to negative 6.

- `DMA_CFG0_SRC_ADDR_DECR_5 = 0x0B`

  Set the step size of DMA channel source address to negative 5.

- `DMA_CFG0_SRC_ADDR_DECR_4 = 0x0C`

  Set the step size of DMA channel source address to negative 4.

- `DMA_CFG0_SRC_ADDR_DECR_3 = 0x0D`

  Set the step size of DMA channel source address to negative 3.

- `DMA_CFG0_SRC_ADDR_DECR_2 = 0x0E`

  Set the step size of DMA channel source address to negative 2.

- `DMA_CFG0_SRC_ADDR_DECR_1 = 0x0F`

  Set the step size of DMA channel source address to negative 1.

20.6.4.4 _DMA_DST_STEP_t

Location: Driver_DMA.h:90

Selects the step size increment to the DMA channel destination address.

Members

- `DMA_CFG0_DEST_ADDR_STATIC = 0x00`

  Do not increment the destination address used by DMA channel.

- `DMA_CFG0_DEST_ADDR_INCR_1 = 0x01`

  Set the step size of DMA channel destination address to 1.

- `DMA_CFG0_DEST_ADDR_INCR_2 = 0x02`

  Set the step size of DMA channel destination address to 2.

- `DMA_CFG0_DEST_ADDR_INCR_3 = 0x03`

  Set the step size of DMA channel destination address to 3.

- `DMA_CFG0_DEST_ADDR_INCR_4 = 0x04`

  Set the step size of DMA channel destination address to 4.

- `DMA_CFG0_DEST_ADDR_INCR_5 = 0x05`

  Set the step size of DMA channel destination address to 5.

- `DMA_CFG0_DEST_ADDR_INCR_6 = 0x06`

  Set the step size of DMA channel destination address to 6.

- `DMA_CFG0_DEST_ADDR_INCR_7 = 0x07`

  Set the step size of DMA channel destination address to 7.

- `DMA_CFG0_DEST_ADDR_DECR_8 = 0x08`

  Set the step size of DMA channel destination address to negative 8.

- `DMA_CFG0_DEST_ADDR_DECR_7 = 0x09`

Set the step size of DMA channel destination address to negative 7.

- `DMA_CFG0_DEST_ADDR_DECR_6 = 0x0A`

  Set the step size of DMA channel destination address to negative 6.

- `DMA_CFG0_DEST_ADDR_DECR_5 = 0x0B`

  Set the step size of DMA channel destination address to negative 5.

- `DMA_CFG0_DEST_ADDR_DECR_4 = 0x0C`

  Set the step size of DMA channel destination address to negative 4.

- `DMA_CFG0_DEST_ADDR_DECR_3 = 0x0D`

  Set the step size of DMA channel destination address to negative 3.

- `DMA_CFG0_DEST_ADDR_DECR_2 = 0x0E`

  Set the step size of DMA channel destination address to negative 2.

- `DMA_CFG0_DEST_ADDR_DECR_1 = 0x0F`

  Set the step size of DMA channel destination address to negative 1.

20.6.4.5 _DMA_SRC_DST_TRANS_LENGHT_SEL_t

Location: Driver_DMA.h:112

Selects whether the transfer length counter depends on either the source word counts or the destination word count.

Members

- `DMA_CFG0_DEST_TRANS_LENGTH_SEL = 0x00`

  Transfer length counter depends on the destination word count.

- `DMA_CFG0_SRC_TRANS_LENGTH_SEL = 0x01`

  Transfer length counter depends on the size word count.

20.6.4.6 _DMA_DATA_MODE_t

Location: Driver_DMA.h:121

Selects how often data is transferred.

Members

- `DMA_REPEAT = 0x0U`

  Data to be transfered repeatedly.

- `DMA_SINGLE = 0x1U`

  Single data transfer.

20.6.4.7 _DMA_BYTE_ORDER_t

Location: Driver_DMA.h:129

Selects the order of the data bytes.

Members

- `DMA_ENDIANNESS_LITTLE = 0x0U`

  Little endian to be used.

- `DMA_ENDIANNESS_BIG = 0x1U`

  Big endian to be used.

20.6.4.8 _DMA_WORD_SIZE_t

Location: Driver_DMA.h:137

Selects the src/dst data word size.

Members

- `DMA_CFG0_DEST_WORD_SIZE_32BITS_TO_32BITS = 0x00`

  Source data uses 32-bit word and destination data uses 32-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_32BITS_TO_4BITS = 0x01`

Source data uses 32-bit word and destination data uses 4-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_32BITS_TO_8BITS = 0x02`

  Source data uses 32-bit word and destination data uses 8-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_32BITS_TO_16BITS = 0x04`

  Source data uses 32-bit word and destination data uses 16-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_4BITS_TO_32BITS = 0x08`

  Source data uses 4-bit word and destination data uses 32-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_4BITS_TO_4BITS = 0x09`

  Source data uses 4-bit word and destination data uses 4-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_4BITS_TO_8BITS = 0x0A`

  Source data uses 4-bit word and destination data uses 8-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_4BITS_TO_16BITS = 0x0C`

  Source data uses 4-bit word and destination data uses 16-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_4BITS_TO_24BITS = 0x0E`

  Source data uses 4-bit word and destination data uses 24-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_8BITS_TO_32BITS = 0x10`

  Source data uses 8-bit word and destination data uses 32-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_8BITS_TO_4BITS = 0x11`

  Source data uses 8-bit word and destination data uses 4-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_8BITS_TO_8BITS = 0x12`

  Source data uses 8-bit word and destination data uses 8-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_8BITS_TO_16BITS = 0x14`

Source data uses 8-bit word and destination data uses 16-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_8BITS_TO_24BITS = 0x16`

  Source data uses 8-bit word and destination data uses 24-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_16BITS_TO_32BITS = 0x20`

  Source data uses 16-bit word and destination data uses 32-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_16BITS_TO_4BITS = 0x21`

  Source data uses 16-bit word and destination data uses 4-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_16BITS_TO_8BITS = 0x22`

  Source data uses 16-bit word and destination data uses 8-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_16BITS_TO_16BITS = 0x24`

  Source data uses 16-bit word and destination data uses 16-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_16BITS_TO_24BITS = 0x26`

Source data uses 16-bit word and destination data uses 24-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_24BITS_TO_4BITS = 0x31`

  Source data uses 24-bit word and destination data uses 4-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_24BITS_TO_8BITS = 0x32`

  Source data uses 24-bit word and destination data uses 8-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_24BITS_TO_16BITS = 0x34`

  Source data uses 24-bit word and destination data uses 16-bit word.

- `DMA_CFG0_DEST_WORD_SIZE_24BITS_TO_24BITS = 0x36`

  Source data uses 24-bit word and destination data uses 24-bit word.

20.6.4.9 _DMA_CH_PRI_t

Location: Driver_DMA.h:166

Selects priority of DMA channels.

Members

- `DMA_CH_PRI_0 = 0x0U`

  Channel priority = 0.

- `DMA_CH_PRI_1 = 0x1U`

  Channel priority = 1.

- `DMA_CH_PRI_2 = 0x2U`

  Channel priority = 2.

- `DMA_CH_PRI_3 = 0x3U`

  Channel priority = 3.

20.6.4.10 _ADC_EVENT_SRC_t

Location: Driver_DMA.h:176

Selects DMA interrupt channel source.

Members

- `DMA_DMA0_EVENT = 1 << DMA_CH_0`

  DMA channel 0 event.

- `DMA_DMA1_EVENT = 1 << DMA_CH_1`

  DMA channel 1 event.

- `DMA_DMA2_EVENT = 1 << DMA_CH_2`

DMA channel 2 event.

- `DMA_DMA3_EVENT = 1 << DMA_CH_3`

DMA channel 3 event.

20.6.5 CMSIS DMA Driver Macro Definition Documentation

20.6.5.1 ARM_DMA_API_VERSION

```
#define ARM_DMA_API_VERSION ARM_DRIVER_VERSION_MAJOR_MINOR(1,0)
```

Location: Driver_DMA.h:37

DMA API version.

20.6.5.2 DMA_ERROR_UNCONFIGURED

```
#define DMA_ERROR_UNCONFIGURED (ARM_DRIVER_ERROR_SPECIFIC - 1)
```

Location: Driver_DMA.h:184

DMA channel has not been configured yet.

20.6.6 CMSIS DMA Driver Function Documentation

20.6.6.1 DMA_GetVersion

```
ARM_DRIVER_VERSION DMA_GetVersion()
```

Location: Driver_DMA.c:21

Get driver version.

Returns:

ARM_DRIVER_VERSION

20.6.6.2 DMA_Initialize

```
int32_t DMA_Initialize()
```

Location: Driver_DMA.c:22

Initialize DMA driver with default configuration.

Returns:

execution_status

20.6.6.3 DMA_Configure

```
int32_t DMA_Configure(DMA_SEL_t sel, const DMA_CFG_t * cfg, DMA_SignalEvent_t
cb)
```

Location: Driver_DMA.c:23

Configure particular DMA channel.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*sel*	DMA channel to be configured (DMA_SEL_t)
`in`	*cfg*	Pointer to DMA_CFG_t
`in`	*cb*	Pointer to DMA_SignalEvent_t

20.6.6.4 DMA_ConfigureWord

```
int32_t DMA_ConfigureWord(DMA_SEL_t sel, uint32_t cfg, DMA_SignalEvent_t cb)
```

Location: Driver_DMA.c:24

Configure particular DMA channel.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*sel*	DMA channel to be configured (DMA_SEL_t)
`in`	*cfg*	Configuration word
`in`	*cb*	Pointer to DMA_SignalEvent_t

20.6.6.5 DMA_ConfigureAddr

```
int32_t DMA_ConfigureAddr(DMA_SEL_t sel, const DMA_ADDR_CFG_t * cfg)
```

Location: Driver_DMA.c:25

Configure DMA channel source and destination addresses.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*sel*	DMA to be configured (DMA_SEL_t)
`in`	*cfg*	Pointer to DMA_ADDR_CFG_t

20.6.6.6 DMA_SetInterruptPriority

```
int32_t DMA_SetInterruptPriority(DMA_SEL_t sel, const DMA_PRI_CFG_t * cfg)
```

Location: Driver_DMA.c:26

Configure the DMA interrupt priority.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*sel*	DMA channel to be configured (DMA_SEL_t)
`in`	*cfg*	Pointer to DMA_PRI_CFG_t

20.6.6.7 DMA_CreateConfigWord

```
uint32_t DMA_CreateConfigWord(const DMA_CFG_t * cfg)
```

Location: Driver_DMA.c:27

Create DMA channel configuration word.

Returns:

configuration word

Parameters

Direction	Name	Description
`in`	*cfg*	Pointer to DMA_CFG_t

20.6.6.8 DMA_SetConfigWord

```
int32_t DMA_SetConfigWord(DMA_SEL_t sel, uint32_t cfg)
```

Location: Driver_DMA.c:28

Quickly updates the DMA channel configuration.

Returns:

none

Parameters

Direction	Name	Description
`in`	*sel*	DMA channel to be configured (DMA_SEL_t)
`in`	*cfg*	configuration word

20.6.6.9 DMA_Start

```
int32_t DMA_Start(DMA_SEL_t sel)
```

Location: Driver_DMA.c:29

Starts the DMA transfer.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*sel*	DMA channel number to be used (DMA_SEL_t)

20.6.6.10 DMA_Stop

```
int32_t DMA_Stop(DMA_SEL_t sel)
```

Location: Driver_DMA.c:30

Stops the DMA transfer.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*sel*	DMA channel number to be used(DMA_SEL_t)

20.6.6.11 DMA_GetCounterValue

```
uint32_t DMA_GetCounterValue(DMA_SEL_t sel)
```

Location: Driver_DMA.c:31

Returns the current counter value of DMA channel.

Returns:

DMA channel counter value

Parameters

Direction	Name	Description
in	*sel*	DMA channel value to be read (DMA_SEL_t)

20.6.6.12 DMA_GetStatus

```
DMA_STATUS_t DMA_GetStatus(DMA_SEL_t sel)
```

Location: Driver_DMA.c:32

Returns the DMA channel status. Clears the status register on read.

Returns:

DMA channel status (DMA_STATUS_t)

Parameters

Direction	Name	Description
in	*sel*	DMA channel value to be read (DMA_SEL_t)

20.6.6.13 DMA_SignalEvent

```
void DMA_SignalEvent(uint32_t event)
```

Location: Driver_DMA.c:34

Signal DMA events.

Returns:

None

Parameters

Direction	Name	Description
`in`	*event*	Notification mask (_ADC_EVENT_SRC_t)

20.7 CMSIS GPIO Driver

CMSIS GPIO Driver Reference.

20.7.1 Summary

Typedefs

- GPIO_SEL_t : GPIO Control Codes: GPIO Selection.
- GPIO_INT_SEL_t : GPIO Control Codes: GPIO INT Selection.
- GPIO_DRIVE_t : GPIO Control Codes: Drive strength.
- GPIO_LPF_t : GPIO Control Codes: Low pass filter.
- GPIO_PULL_t : GPIO Control Codes: Pull control.
- GPIO_MODE_t : GPIO Control Codes: IO Mode.
- GPIO_FUNC_REGISTERS_t : GPIO Control Codes: GPIO alternative function registers.
- GPIO_EN_DIS_t : GPIO Control Codes: Enable / Disable values.
- GPIO_DIR_t : GPIO Control Codes: GPIO direction.
- GPIO_EVENT_t : GPIO Control Codes: Interrupts events.
- GPIO_DBC_CLK_t : GPIO Control Codes: Debounce clock source.
- GPIO_DRIVE_STRENGTHS_t : GPIO Control Codes: Pads strength.
- GPIO_SignalEvent_t : Pointer to GPIO_SignalEvent : Signal GPIO Event.
- GPIO_DBF_CFG_t : Debounce filter configuration.
- GPIO_PRI_CFG_t : GPIO interrupt priority configuration.
- GPIO_CFG_t : GPIO Driver configuration.
- GPIO_PAD_CFG_t : GPIO PAD configuration.
- GPIO_INT_CFG_t : GPIO INT configuration.
- GPIO_EXTCLK_CFG_t : External clock pad configuration.
- GPIO_JTAG_SW_CFG_t : JTAG configuration.
- DRIVER_GPIO_t : Access structure of the GPIO Driver.

Data Structures

- _GPIO_DBF_CFG_t : Debounce filter configuration.
- _GPIO_PRI_CFG_t : GPIO interrupt priority configuration.
- _GPIO_CFG_t : GPIO Driver configuration.
- _GPIO_PAD_CFG_t : GPIO PAD configuration.
- _GPIO_INT_CFG_t : GPIO INT configuration.
- _GPIO_EXTCLK_CFG_t : External clock pad configuration.
- _GPIO_JTAG_SW_CFG_t : JTAG configuration.
- _DRIVER_GPIO_t : Access structure of the GPIO Driver.

Enumerations

- _GPIO_SEL_t : GPIO Control Codes: GPIO Selection.
- _GPIO_INT_SEL_t : GPIO Control Codes: GPIO INT Selection.
- _GPIO_DRIVE_t : GPIO Control Codes: Drive strength.
- _GPIO_LPF_t : GPIO Control Codes: Low pass filter.
- _GPIO_PULL_t : GPIO Control Codes: Pull control.
- _GPIO_MODE_t : GPIO Control Codes: IO Mode.
- _GPIO_FUNC_REGISTERS_t : GPIO Control Codes: GPIO alternative function registers.
- _GPIO_EN_DIS_t : GPIO Control Codes: Enable / Disable values.
- _GPIO_DIR_t : GPIO Control Codes: GPIO direction.
- _GPIO_EVENT_t : GPIO Control Codes: Interrupts events.
- _GPIO_DBC_CLK_t : GPIO Control Codes: Debounce clock source.
- _GPIO_DRIVE_STRENGTHS_t : GPIO Control Codes: Pads strength.

Macros

- ARM_GPIO_API_VERSION : GPIO API version.
- GPIO_EVENT_0_IRQ : GPIO Event.
- GPIO_EVENT_1_IRQ : GPIO1 interrupt event value.
- GPIO_EVENT_2_IRQ : GPIO2 interrupt event value.
- GPIO_EVENT_3_IRQ : GPIO3 interrupt event value.

Functions

- GPIO_GetVersion : Get driver version.
- GPIO_Initialize : Initialize the GPIO driver.
- GPIO_Configure : Configure common GPIO settings.
- GPIO_ConfigurePad : Configure the GPIO pad.
- GPIO_ConfigureInterrupt : Configure the GPIO interrupt.
- GPIO_SetInterruptPriority : Configure GPIO interrupt priority.
- GPIO_ConfigureJTAG : Configure the GPIO JTAG mode.
- GPIO_SetDir : Set particular GPIO pad direction.
- GPIO_SetHigh : Set particular GPIO pad.
- GPIO_ToggleValue : Toggle particular GPIO pad.

- GPIO_SetLow : Reset particular GPIO pad.
- GPIO_ReadValue : Returns the selected GPIO pad value.
- GPIO_ResetAltFuncRegister : Reset the particular alternative function register.
- GPIO_SignalEvent : Signal GPIO events.

20.7.2 CMSIS GPIO Driver Typedef Documentation

20.7.2.1 GPIO_SEL_t

```
typedef enum _GPIO_SEL_t GPIO_SEL_t
```

Location: Driver_GPIO.h:59

GPIO Control Codes: GPIO Selection.

20.7.2.2 GPIO_INT_SEL_t

```
typedef enum _GPIO_INT_SEL_t GPIO_INT_SEL_t
```

Location: Driver_GPIO.h:67

GPIO Control Codes: GPIO INT Selection.

20.7.2.3 GPIO_DRIVE_t

```
typedef enum _GPIO_DRIVE_t GPIO_DRIVE_t
```

Location: Driver_GPIO.h:75

GPIO Control Codes: Drive strength.

20.7.2.4 GPIO_LPF_t

```
typedef enum _GPIO_LPF_t GPIO_LPF_t
```

Location: Driver_GPIO.h:81

GPIO Control Codes: Low pass filter.

20.7.2.5 GPIO_PULL_t

```
typedef enum _GPIO_PULL_t GPIO_PULL_t
```

Location: Driver_GPIO.h:89

GPIO Control Codes: Pull control.

20.7.2.6 GPIO_MODE_t

```
typedef enum _GPIO_MODE_t GPIO_MODE_t
```

Location: Driver_GPIO.h:233

GPIO Control Codes: IO Mode.

20.7.2.7 GPIO_FUNC_REGISTERS_t

```
typedef enum _GPIO_FUNC_REGISTERS_t GPIO_FUNC_REGISTERS_t
```

Location: Driver_GPIO.h:251

GPIO Control Codes: GPIO alternative function registers.

20.7.2.8 GPIO_EN_DIS_t

```
typedef enum _GPIO_EN_DIS_t GPIO_EN_DIS_t
```

Location: Driver_GPIO.h:257

GPIO Control Codes: Enable / Disable values.

20.7.2.9 GPIO_DIR_t

```
typedef enum _GPIO_DIR_t GPIO_DIR_t
```

Location: Driver_GPIO.h:263

GPIO Control Codes: GPIO direction.

20.7.2.10 GPIO_EVENT_t

```
typedef enum _GPIO_EVENT_t GPIO_EVENT_t
```

Location: Driver_GPIO.h:273

GPIO Control Codes: Interrupts events.

20.7.2.11 GPIO_DBC_CLK_t

```
typedef enum _GPIO_DBC_CLK_t GPIO_DBC_CLK_t
```

Location: Driver_GPIO.h:279

GPIO Control Codes: Debounce clock source.

20.7.2.12 GPIO_DRIVE_STRENGTHS_t

```
typedef enum _GPIO_DRIVE_STRENGTHS_t GPIO_DRIVE_STRENGTHS_t
```

Location: Driver_GPIO.h:285

GPIO Control Codes: Pads strength.

20.7.2.13 GPIO_SignalEvent_t

```
typedef void(* GPIO_SignalEvent_t
```

Location: Driver_GPIO.h:369

Pointer to GPIO_SignalEvent : Signal GPIO Event.

20.7.2.14 GPIO_DBF_CFG_t

```
typedef struct _GPIO_DBF_CFG_t GPIO_DBF_CFG_t
```

Location: Driver_GPIO.h:379

Debounce filter configuration.

20.7.2.15 GPIO_PRI_CFG_t

```
typedef struct _GPIO_PRI_CFG_t GPIO_PRI_CFG_t
```

Location: Driver_GPIO.h:390

GPIO interrupt priority configuration.

20.7.2.16 GPIO_CFG_t

```
typedef struct _GPIO_CFG_t GPIO_CFG_t
```

Location: Driver_GPIO.h:400

GPIO Driver configuration.

20.7.2.17 GPIO_PAD_CFG_t

```
typedef struct _GPIO_PAD_CFG_t GPIO_PAD_CFG_t
```

Location: Driver_GPIO.h:414

GPIO PAD configuration.

20.7.2.18 GPIO_INT_CFG_t

```
typedef struct _GPIO_INT_CFG_t GPIO_INT_CFG_t
```

Location: Driver_GPIO.h:426

GPIO INT configuration.

20.7.2.19 GPIO_EXTCLK_CFG_t

```
typedef struct _GPIO_EXTCLK_CFG_t GPIO_EXTCLK_CFG_t
```

Location: Driver_GPIO.h:436

External clock pad configuration.

20.7.2.20 GPIO_JTAG_SW_CFG_t

```
typedef struct _GPIO_JTAG_SW_CFG_t GPIO_JTAG_SW_CFG_t
```

Location: Driver_GPIO.h:451

JTAG configuration.

20.7.2.21 DRIVER_GPIO_t

```
typedef struct _DRIVER_GPIO_t DRIVER_GPIO_t
```

Location: Driver_GPIO.h:470

Access structure of the GPIO Driver.

20.7.3 CMSIS GPIO Driver Data Structures Type Documentation

20.7.3.1 _GPIO_DBF_CFG_t

Location: Driver_GPIO.h:374

Debounce filter configuration.

Data Fields

Type	*Name*	Description
`uint8_t`	*count*	Debounce filter count value
`GPIO_DBC_CLK_t`	*clk_source*	Debounce filter clock source.
`uint8_t`	*__pad0__*	Reserved

20.7.3.2 _GPIO_PRI_CFG_t

Location: Driver_GPIO.h:384

GPIO interrupt priority configuration.

Data Fields

`Type`	*Name*	Description
`uint32_t`	*preempt_pri*	Preempt priority
`uint32_t`	*__pad0__*	Reserved
`uint32_t`	*subgrp_pri*	Subgroup priority.
`uint32_t`	*__pad1__*	Reserved

20.7.3.3 _GPIO_CFG_t

Location: Driver_GPIO.h:395

GPIO Driver configuration.

Data Fields

`Type`	*Name*	Description
`GPIO_DRIVE_STRENGTHS_t`	*drive_strengths*	Drive strengths configuration.
`uint8_t`	*__pad0__*	Reserved
`GPIO_DBF_CFG_t`	*debounce_cfg*	Debounce filter configuration.

20.7.3.4 _GPIO_PAD_CFG_t

Location: Driver_GPIO.h:405

GPIO PAD configuration.

Data Fields

Type	Name	Description
GPIO_PULL_t	*pull_mode*	Pull control
uint8_t	*__pad0__*	Reserved
GPIO_DRIVE_t	*drive_mode*	Drive mode
uint8_t	*__pad1__*	Reserved
GPIO_LPF_t	*lpf_en*	Low pass filter enable.
uint16_t	*__pad2__*	Reserved
GPIO_MODE_t	*io_mode*	IO mode

20.7.3.5 _GPIO_INT_CFG_t

Location: Driver_GPIO.h:419

GPIO INT configuration.

Data Fields

Type	Name	Description
GPIO_SEL_t	*src_sel*	Interrupt source selection.
GPIO_EVENT_t	*event*	Event selection
GPIO_EN_DIS_t	*debounce_en*	Debouce filter enable
GPIO_EN_DIS_t	*interrup_en*	Interrupt enable flag
uint8_t	*__pad0__*	Reserved

20.7.3.6 _GPIO_EXTCLK_CFG_t

Location: Driver_GPIO.h:431

External clock pad configuration.

Data Fields

Type	*Name*	Description
GPIO_PULL_t	*pull_mode*	Pull control
GPIO_LPF_t	*lpf_en*	Low pass filter enable.
uint8_t	*__pad0__*	Reserved

20.7.3.7 _GPIO_JTAG_SW_CFG_t

Location: Driver_GPIO.h:441

JTAG configuration.

Data Fields

Type	*Name*	Description
GPIO_LPF_t	*swclk_jtck_lpf_en*	SWCLK/JTCK low pass filter enable
GPIO_LPF_t	*swdio_jtms_lpf_en*	SWDIO/JTMS low pass filter enable
GPIO_EN_DIS_t	*jtag_data_en*	JTAG data available on GPIO[2:3]
GPIO_EN_DIS_t	*jtag_trst_en*	JTAG trst available on GPIO4
GPIO_PULL_t	*swclk_jtck_pull*	SWCLK/JTCK pull mode
GPIO_PULL_t	*swdio_jtms_pull*	SWDIO/JTMS pull mode
GPIO_DRIVE_t	*swdio_jtms_drive*	SWDIO/JTMS drive mode
uint8_t	*__pad0__*	Reserved

20.7.3.8 _DRIVER_GPIO_t

Location: Driver_GPIO.h:456

Access structure of the GPIO Driver.

Data Fields

Type	*Name*	Description
ARM_DRIVER_VERSION(*	*GetVersion)(void)*	Pointer to GPIO_GetVersion : Get driver version.
int32_t(*	*Initialize)(GPIO_SignalEvent_t cb)*	Pointer to GPIO_Initialize : Initialize the GPIO driver.
int32_t(*	*Configure)(const GPIO_CFG_t *cfg)*	Pointer to GPIO_Configure : Configure common GPIO settings.
int32_t(*	*ConfigurePad)(GPIO_SEL_t sel, const GPIO_PAD_CFG_t *cfg)*	Pointer to GPIO_ConfigurePad : Configure the GPIO pad.
int32_t(*	*ConfigureInterrupt)(GPIO_INT_SEL_t sel, const GPIO_INT_CFG_t *cfg)*	Pointer to GPIO_ConfigureInterrupt : Configure the GPIO interrupt.
int32_t(*	*SetInterruptPriority)(GPIO_INT_SEL_t sel, const GPIO_PRI_CFG_t *pri)*	Pointer to GPIO_SetInterruptPriority : Configure GPIO interrupt priority.
int32_t(*	*ConfigureJTAGSW)(const GPIO_JTAG_SW_CFG_t *cfg)*	Pointer to GPIO_ConfigureJTAG : Configure the GPIO JTAG mode.
void(*	*SetDir)(GPIO_DIR_t dir)*	Pointer to GPIO_SetDir : Set particular GPIO pad direction.
void(*	*SetHigh)(GPIO_SEL_t sel)*	Pointer to GPIO_SetHigh : Set particular GPIO pad.
void(*	*ToggleValue)(GPIO_SEL_t sel)*	Pointer to GPIO_ToggleValue : Toggle particular GPIO pad.
void(*	*SetLow)(GPIO_SEL_t sel)*	Pointer to GPIO_SetLow : Reset particular GPIO pad.
uint32_t(*	*ReadValue)(GPIO_SEL_t sel)*	Pointer to GPIO_ReadValue : Return the selected GPIO value.
int32_t(*	*ResetAltFuncRegister)(GPIO_FUNC_REGISTERS_t sel)*	Pointer to GPIO_ResetAltFuncRegister : Reset GPIO alternative function register.

20.7.4 CMSIS GPIO Driver Enumeration Type Documentation

20.7.4.1 _GPIO_SEL_t

Location: Driver_GPIO.h:42

GPIO Control Codes: GPIO Selection.

Members

- `GPIO_0 = 0x0`

  GPIO pad 0

- `GPIO_1 = 0x1`

  GPIO pad 1

- `GPIO_2 = 0x2`

  GPIO pad 2

- `GPIO_3 = 0x3`

  GPIO pad 3

- `GPIO_4 = 0x4`

  GPIO pad 4

- `GPIO_5 = 0x5`

  GPIO pad 5

- `GPIO_6 = 0x6`

  GPIO pad 6

- `GPIO_7 = 0x7`

  GPIO pad 7
- `GPIO_8 = 0x8`

  GPIO pad 8
- `GPIO_9 = 0x9`

  GPIO pad 9
- `GPIO_10 = 0xA`

  GPIO pad 10.
- `GPIO_11 = 0xB`

  GPIO pad 11.
- `GPIO_12 = 0xC`

  GPIO pad 12.
- `GPIO_13 = 0xD`

  GPIO pad 13.
- `GPIO_14 = 0xE`

  GPIO pad 14.
- `GPIO_15 = 0xF`

  GPIO pad 15.

20.7.4.2 _GPIO_INT_SEL_t

Location: Driver_GPIO.h:62

GPIO Control Codes: GPIO INT Selection.

Members

- `GPIO_INT_0 = 0x0`

  GPIO interrupt 0.

- `GPIO_INT_1 = 0x1`

  GPIO interrupt 1.

- `GPIO_INT_2 = 0x2`

  GPIO interrupt 2.

- `GPIO_INT_3 = 0x3`

  GPIO interrupt 3.

20.7.4.3 _GPIO_DRIVE_t

Location: Driver_GPIO.h:70

GPIO Control Codes: Drive strength.

Members

- `GPIO_2X = 0x0`

  2x drive strength

- `GPIO_3X = 0x1`

  3x drive strength

- `GPIO_5X = 0x2`

  5x drive strength

- `GPIO_6X = 0x3`

  6x drive strength

20.7.4.4 _GPIO_LPF_t

Location: Driver_GPIO.h:78

GPIO Control Codes: Low pass filter.

Members

- `GPIO_LPF_DISABLED = 0x0`

  Low pass filter disabled.

- `GPIO_LPF_ENABLED = 0x1`

  Low pass filter enabled

20.7.4.5 _GPIO_PULL_t

Location: Driver_GPIO.h:84

GPIO Control Codes: Pull control.

Members

- `GPIO_PC_NO_PULL = 0x0`

  No pull selected
- `GPIO_PC_WEAK_PULL_UP = 0x1`

  Weak pull-up selected
- `GPIO_PC_WEAK_PULL_DOWN = 0x2`

  Weak pull-down selected.
- `GPIO_PC_STRONG_PULL_UP = 0x3`

  Strong pull-up selected.

20.7.4.6 _GPIO_MODE_t

Location: Driver_GPIO.h:92

GPIO Control Codes: IO Mode.

Members

- `MODE_GPIO_DISABLE = 0x000`
- `MODE_GPIO_INPUT = 0x001`
- `MODE_GPIO_GPIO_IN = 0x002`
- `MODE_GPIO_GPIO_OUT = 0x003`
- `MODE_GPIO_SLOWCLK = 0x004`
- `MODE_GPIO_SYSCLK = 0x005`
- `MODE_GPIO_USRCLK = 0x006`
- `MODE_GPIO_RCCLK = 0x007`
- `MODE_GPIO_SWCLK_DIV = 0x008`
- `MODE_GPIO_EXTCLK_DIV = 0x009`
- `MODE_GPIO_RFCLK = 0x00A`
- `MODE_GPIO_STANDBYCLK = 0x00B`
- `MODE_GPIO_SENSORCLK = 0x00C`
- `MODE_GPIO_SPI0_IO0 = 0x00D`
- `MODE_GPIO_SPI0_IO1 = 0x00E`
- `MODE_GPIO_SPI0_IO2 = 0x00F`
- `MODE_GPIO_SPI0_IO3 = 0x010`
- `MODE_GPIO_SPI0_CS = 0x011`
- `MODE_GPIO_SPI0_CLK = 0x012`
- `MODE_GPIO_SPI1_IO0 = 0x013`
- `MODE_GPIO_SPI1_IO1 = 0x014`
- `MODE_GPIO_SPI1_IO2 = 0x015`
- `MODE_GPIO_SPI1_IO3 = 0x016`
- `MODE_GPIO_SPI1_CS = 0x017`
- `MODE_GPIO_SPI1_CLK = 0x018`
- `MODE_GPIO_UART0_TX = 0x019`
- `MODE_GPIO_I2C0_SCL = 0x01A`
- `MODE_GPIO_I2C0_SDA = 0x01B`
- `MODE_GPIO_I2C1_SCL = 0x01C`

- MODE_GPIO_I2C1_SDA = 0x01D
- MODE_GPIO_PCM0_SERO = 0x01E
- MODE_GPIO_PCM0_FRAME = 0x01F
- MODE_GPIO_PWM0 = 0x020
- MODE_GPIO_PWM1 = 0x021
- MODE_GPIO_PWM2 = 0x022
- MODE_GPIO_PWM3 = 0x023
- MODE_GPIO_PWM4 = 0x024
- MODE_GPIO_PWM0_INV = 0x025
- MODE_GPIO_PWM1_INV = 0x026
- MODE_GPIO_PWM2_INV = 0x027
- MODE_GPIO_PWM3_INV = 0x028
- MODE_GPIO_PWM4_INV = 0x029
- MODE_GPIO_LIN0_TX = 0x02A
- MODE_GPIO_BB_TX_DATA = 0x02B
- MODE_GPIO_BB_TX_DATA_VALID = 0x02C
- MODE_GPIO_BB_SPI_CSN = 0x02D
- MODE_GPIO_BB_SPI_CLK = 0x02E
- MODE_GPIO_BB_SPI_MOSI = 0x02F
- MODE_GPIO_BB_DBG_0 = 0x030
- MODE_GPIO_BB_DBG_1 = 0x031
- MODE_GPIO_BB_DBG_2 = 0x032
- MODE_GPIO_BB_DBG_3 = 0x033
- MODE_GPIO_BB_DBG_4 = 0x034
- MODE_GPIO_BB_DBG_5 = 0x035
- MODE_GPIO_BB_DBG_6 = 0x036
- MODE_GPIO_BB_DBG_7 = 0x037
- MODE_GPIO_BB_BLE_SYNC = 0x038
- MODE_GPIO_BB_BLE_IN_PROCESS = 0x039
- MODE_GPIO_BB_BLE_TX = 0x03A

- `MODE_GPIO_BB_BLE_RX = 0x03B`
- `MODE_GPIO_BB_BLE_PTI_0 = 0x03C`
- `MODE_GPIO_BB_BLE_PTI_1 = 0x03D`
- `MODE_GPIO_BB_BLE_PTI_2 = 0x03E`
- `MODE_GPIO_BB_BLE_PTI_3 = 0x03F`
- `MODE_GPIO_BB_ANT_SW_EN = 0x040`
- `MODE_GPIO_BB_ANT_SW_0 = 0x041`
- `MODE_GPIO_BB_ANT_SW_1 = 0x042`
- `MODE_GPIO_BB_ANT_SW_2 = 0x043`
- `MODE_GPIO_BB_ANT_SW_3 = 0x044`
- `MODE_GPIO_BB_ANT_SW_4 = 0x045`
- `MODE_GPIO_BB_ANT_SW_5 = 0x046`
- `MODE_GPIO_BB_ANT_SW_6 = 0x047`
- `MODE_GPIO_BB_CTE_MODE = 0x048`
- `MODE_GPIO_BB_CTE_SAMPLE_P = 0x049`
- `MODE_GPIO_RF_SPI_MISO = 0x04A`
- `MODE_GPIO_RF_GPIO0 = 0x04B`
- `MODE_GPIO_RF_GPIO1 = 0x04C`
- `MODE_GPIO_RF_GPIO2 = 0x04D`
- `MODE_GPIO_RF_GPIO3 = 0x04E`
- `MODE_GPIO_RF_GPIO4 = 0x04F`
- `MODE_GPIO_RF_GPIO5 = 0x050`
- `MODE_GPIO_RF_GPIO6 = 0x051`
- `MODE_GPIO_RF_GPIO7 = 0x052`
- `MODE_GPIO_RF_GPIO8 = 0x053`
- `MODE_GPIO_RF_GPIO9 = 0x054`
- `MODE_GPIO_RF_IQ_DATA_P = 0x055`
- `MODE_GPIO_RF_I_DATA_0 = 0x056`
- `MODE_GPIO_RF_I_DATA_1 = 0x057`
- `MODE_GPIO_RF_I_DATA_2 = 0x058`

- MODE_GPIO_RF_I_DATA_3 = 0x059
- MODE_GPIO_RF_I_DATA_4 = 0x05A
- MODE_GPIO_RF_I_DATA_5 = 0x05B
- MODE_GPIO_RF_I_DATA_6 = 0x05C
- MODE_GPIO_RF_I_DATA_7 = 0x05D
- MODE_GPIO_RF_Q_DATA_0 = 0x05E
- MODE_GPIO_RF_Q_DATA_1 = 0x05F
- MODE_GPIO_RF_Q_DATA_2 = 0x060
- MODE_GPIO_RF_Q_DATA_3 = 0x061
- MODE_GPIO_RF_Q_DATA_4 = 0x062
- MODE_GPIO_RF_Q_DATA_5 = 0x063
- MODE_GPIO_RF_Q_DATA_6 = 0x064
- MODE_GPIO_RF_Q_DATA_7 = 0x065
- MODE_GPIO_RF_ANT_SW_0 = 0x066
- MODE_GPIO_RF_ANT_SW_1 = 0x067
- MODE_GPIO_RF_ANT_SW_2 = 0x068
- MODE_GPIO_RF_ANT_SW_3 = 0x069
- MODE_GPIO_TOF_START = 0x06A
- MODE_GPIO_TOF_STOP = 0x06B
- MODE_GPIO_PCM_SERI_IN = 0x100
- MODE_GPIO_PCM_FRAME_IN = 0x101
- MODE_GPIO_PCM_FRAME_OUT = 0x102
- MODE_GPIO_PCM_CLK_IN = 0x103
- MODE_GPIO_SPI0_CS_IN = 0x200
- MODE_GPIO_SPI0_CLK_IN = 0x201
- MODE_GPIO_SPI1_CS_IN = 0x202
- MODE_GPIO_SPI1_CLK_IN = 0x203
- MODE_GPIO_UART_RX_IN = 0x300
- MODE_GPIO_I2C0_SCL_IN = 0x400
- MODE_GPIO_I2C0_SDA_IN = 0x401

- `MODE_GPIO_I2C1_SCL_IN = 0x402`
- `MODE_GPIO_I2C1_SDA_IN = 0x403`
- `MODE_GPIO_NMI_IN = 0x500`
- `MODE_GPIO_BB_RX_CLK_IN = 0x600`
- `MODE_GPIO_BB_RX_DATA_IN = 0x601`
- `MODE_GPIO_BB_SYNC_P_IN = 0x602`
- `MODE_GPIO_BB_SPI_MISO_IN = 0x603`
- `MODE_GPIO_RF_SPI_MOSI_IN = 0x700`
- `MODE_GPIO_RF_SPI_CSN_IN = 0x701`
- `MODE_GPIO_RF_SPI_CLK_IN = 0x702`
- `MODE_GPIO_RF_GPIO0_IN = 0x800`
- `MODE_GPIO_RF_GPIO1_IN = 0x801`
- `MODE_GPIO_RF_GPIO2_IN = 0x802`
- `MODE_GPIO_RF_GPIO3_IN = 0x803`
- `MODE_GPIO_RF_GPIO4_IN = 0x804`
- `MODE_GPIO_RF_GPIO5_IN = 0x805`
- `MODE_GPIO_RF_GPIO6_IN = 0x806`
- `MODE_GPIO_RF_GPIO7_IN = 0x807`
- `MODE_GPIO_RF_GPIO8_IN = 0x808`
- `MODE_GPIO_RF_GPIO9_IN = 0x809`
- `MODE_GPIO_ADC_IN = 0x80A`

20.7.4.7 _GPIO_FUNC_REGISTERS_t

Location: Driver_GPIO.h:236

GPIO Control Codes: GPIO alternative function registers.

Members

- `GPIO_FUNC_REG_SPI0 = 0x00`

  SPI 0 register
- `GPIO_FUNC_REG_SPI1 = 0x01`

  SPI 1 register
- `GPIO_FUNC_REG_UART0 = 0x03`

  UART register
- `GPIO_FUNC_REG_I2C0 = 0x04`

  I2C 0 register
- `GPIO_FUNC_REG_I2C1 = 0x05`

  I2C 1 register
- `GPIO_FUNC_REG_PCM0 = 0x06`

  PCM register
- `GPIO_FUNC_REG_NMI = 0x07`

  NMI register
- `GPIO_FUNC_REG_BB_RX = 0x08`

  BB RX register
- `GPIO_FUNC_REG_BB_SPI = 0x09`

  BB SPI register

- `GPIO_FUNC_REG_RF_SPI = 0x0A`

  RF SPI register

- `GPIO_FUNC_REG_RF_GPIO03 = 0x0B`

  RF GPIO03 register

- `GPIO_FUNC_REG_RF_GPIO47 = 0x0C`

  RF GPIO47 register

- `GPIO_FUNC_REG_RF_GPIO89 = 0x0D`

  RF GPIO89 register

- `GPIO_FUNC_REG_JTAG_SW_PAD = 0x0E`

  JTAG SW pad register.

20.7.4.8 _GPIO_EN_DIS_t

Location: Driver_GPIO.h:254

GPIO Control Codes: Enable / Disable values.

Members

- `GPIO_DISABLE = 0`

  GPIO disable value.

- `GPIO_ENABLE = 1`

GPIO enable value

20.7.4.9 _GPIO_DIR_t

Location: Driver_GPIO.h:260

GPIO Control Codes: GPIO direction.

Members

- `GPIO_IN = 0`

  GPIO direction input

- `GPIO_OUT = 1`

  GPIO direction output.

20.7.4.10 _GPIO_EVENT_t

Location: Driver_GPIO.h:266

GPIO Control Codes: Interrupts events.

Members

- `GPIO_IN_EVENT_NONE = 0`

  Interrupt event none

- `GPIO_IN_EVENT_HIGH_LEVEL = 1`

  Interrupt event high level

- `GPIO_IN_EVENT_LOW_LEVEL = 2`

  Interrupt event low level

- `GPIO_IN_EVENT_RISING_EDGE = 3`

  Interrupt event rising edge

- `GPIO_IN_EVENT_FALLING_EDGE = 4`

  Interrupt event falling edge.

- `GPIO_IN_EVENT_TRANSITION = 5`

  Interrupt event transition

20.7.4.11 _GPIO_DBC_CLK_t

Location: Driver_GPIO.h:276

GPIO Control Codes: Debounce clock source.

Members

- `GPIO_DBC_CLK_SLOWCLK_DIV32 = 0`

  Debounce clock source = slow clock / 32

- `GPIO_DBC_CLK_SLOWCLK_DIV1024 = 1`

Debounce clock source = slow clock / 1024.

20.7.4.12 _GPIO_DRIVE_STRENGTHS_t

Location: Driver_GPIO.h:282

GPIO Control Codes: Pads strength.

Members

- `GPIO_LOW_DRIVE = 0`

  Regular drive strengths

- `GPIO_HIGH_DRIVE = 1`

  Drive strengths increased by ~50%.

20.7.5 CMSIS GPIO Driver Macro Definition Documentation

20.7.5.1 ARM_GPIO_API_VERSION

```
#define ARM_GPIO_API_VERSION ARM_DRIVER_VERSION_MAJOR_MINOR(1,0)
```

Location: Driver_GPIO.h:37

GPIO API version.

20.7.5.2 GPIO_EVENT_0_IRQ

```
#define GPIO_EVENT_0_IRQ (1UL << 0)
```

Location: Driver_GPIO.h:288

GPIO Event.

GPIO0 interrupt event value

20.7.5.3 GPIO_EVENT_1_IRQ

```
#define GPIO_EVENT_1_IRQ (1UL << 1)
```

Location: Driver_GPIO.h:289

GPIO1 interrupt event value.

20.7.5.4 GPIO_EVENT_2_IRQ

```
#define GPIO_EVENT_2_IRQ (1UL << 2)
```

Location: Driver_GPIO.h:290

GPIO2 interrupt event value.

20.7.5.5 GPIO_EVENT_3_IRQ

```
#define GPIO_EVENT_3_IRQ (1UL << 3)
```

Location: Driver_GPIO.h:291

GPIO3 interrupt event value.

20.7.6 CMSIS GPIO Driver Function Documentation

20.7.6.1 GPIO_GetVersion

```
ARM_DRIVER_VERSION GPIO_GetVersion()
```

Location: Driver_GPIO.c:21

Get driver version.

Returns:

ARM_DRIVER_VERSION

20.7.6.2 GPIO_Initialize

```
int32_t GPIO_Initialize(GPIO_SignalEvent_t cb)
```

Location: Driver_GPIO.c:22

Initialize the GPIO driver.

Returns:

execution_status

Parameters

Direction	Name	Description
in	*cb*	Pointer to GPIO_SignalEvent

20.7.6.3 GPIO_Configure

```
int32_t GPIO_Configure(const GPIO_CFG_t * cfg)
```

Location: Driver_GPIO.c:23

Configure common GPIO settings.

Returns:

execution_status

Parameters

Direction	Name	Description
in	*cfg*	Pointer to GPIO_CFG_t

20.7.6.4 GPIO_ConfigurePad

```
int32_t GPIO_ConfigurePad(GPIO_SEL_t sel, const GPIO_PAD_CFG_t * cfg)
```

Location: Driver_GPIO.c:24

Configure the GPIO pad.

Returns:

execution_status

Parameters

Direction	Name	Description
in	*sel*	Pad selection GPIO_SEL_t
in	*cfg*	Pointer to GPIO_PAD_CFG_t

20.7.6.5 GPIO_ConfigureInterrupt

```
int32_t GPIO_ConfigureInterrupt(GPIO_INT_SEL_t sel, const GPIO_INT_CFG_t * cfg)
```

Location: Driver_GPIO.c:25

Configure the GPIO interrupt.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*sel*	Interrupt selection GPIO_INT_SEL_t
`in`	*cfg*	Pointer to GPIO_INT_CFG_t

20.7.6.6 GPIO_SetInterruptPriority

```
 int32_t GPIO_SetInterruptPriority(GPIO_INT_SEL_t sel, const GPIO_PRI_CFG_t *
cfg)
```

Location: Driver_GPIO.c:26

Configure GPIO interrupt priority.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*sel*	Interrupt selection GPIO_INT_SEL_t
`in`	*cfg*	Pointer to GPIO_PRI_CFG_t

20.7.6.7 GPIO_ConfigureJTAG

```
 int32_t GPIO_ConfigureJTAG(const GPIO_JTAG_SW_CFG_t * cfg)
```

Location: Driver_GPIO.c:27

Configure the GPIO JTAG mode.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*cfg*	Pointer to GPIO_JTAG_SW_CFG_t

20.7.6.8 GPIO_SetDir

```
void GPIO_SetDir(GPIO_DIR_t dir)
```

Location: Driver_GPIO.c:28

Set particular GPIO pad direction.

Returns:

None

Parameters

Direction	Name	Description
`in`	*dir*	Pad direction GPIO_DIR_t

20.7.6.9 GPIO_SetHigh

```
void GPIO_SetHigh(GPIO_SEL_t sel)
```

Location: Driver_GPIO.c:29

Set particular GPIO pad.

Returns:

None

Parameters

Direction	Name	Description
in	*sel*	Pad selection GPIO_SEL_t

20.7.6.10 GPIO_ToggleValue

```
void GPIO_ToggleValue(GPIO_SEL_t sel)
```

Location: Driver_GPIO.c:30

Toggle particular GPIO pad.

Returns:

None

Parameters

Direction	Name	Description
in	*sel*	Pad selection GPIO_SEL_t

20.7.6.11 GPIO_SetLow

```
void GPIO_SetLow(GPIO_SEL_t sel)
```

Location: Driver_GPIO.c:31

Reset particular GPIO pad.

Returns:

None

Parameters

Direction	Name	Description
`in`	*sel*	Pad selection GPIO_SEL_t

20.7.6.12 GPIO_ReadValue

```
uint32_t GPIO_ReadValue(GPIO_SEL_t sel)
```

Location: Driver_GPIO.c:32

Returns the selected GPIO pad value.

Returns:

GPIO pad value

Parameters

Direction	Name	Description
`in`	*sel*	Pad selection GPIO_SEL_t

20.7.6.13 GPIO_ResetAltFuncRegister

```
int32_t GPIO_ResetAltFuncRegister(GPIO_FUNC_REGISTERS_t reg)
```

Location: Driver_GPIO.c:33

Reset the particular alternative function register.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*reg*	Register selection GPIO_FUNC_REGISTERS_t

20.7.6.14 GPIO_SignalEvent

```
void GPIO_SignalEvent(uint32_t event)
```

Location: Driver_GPIO.c:35

Signal GPIO events.

Returns:

None

Parameters

Direction	Name	Description
`in`	*event*	notification mask

20.8 CMSIS I2C Driver

CMSIS I^{2}C Driver Reference.

20.8.1 Summary

Typedefs

- ARM_I2C_STATUS : I2C Status.
- ARM_I2C_SignalEvent_t : Pointer to ARM_I2C_SignalEvent : Signal I2C Event.
- ARM_I2C_CAPABILITIES : I2C Driver Capabilities.
- ARM_DRIVER_I2C : Access structure of the I2C Driver.

Data Structures

- _ARM_I2C_STATUS : I2C Status.
- _ARM_I2C_CAPABILITIES : I2C Driver Capabilities.
- _ARM_DRIVER_I2C : Access structure of the I2C Driver.

Macros

- ARM_I2C_API_VERSION : I2C API version.
- ARM_I2C_OWN_ADDRESS : Set Own Slave Address; arg = address.
- ARM_I2C_BUS_SPEED : Set Bus Speed; arg = speed.
- ARM_I2C_BUS_CLEAR : Execute Bus clear: send nine clock pulses.
- ARM_I2C_ABORT_TRANSFER : Abort Master/Slave Transmit/Receive.
- ARM_I2C_BUS_SPEED_STANDARD : Standard Speed (100kHz)
- ARM_I2C_BUS_SPEED_FAST : Fast Speed (400kHz)
- ARM_I2C_BUS_SPEED_FAST_PLUS : Fast+ Speed (1MHz)
- ARM_I2C_BUS_SPEED_HIGH : High Speed (3.4MHz)
- ARM_I2C_ADDRESS_10BIT : 10-bit address flag
- ARM_I2C_ADDRESS_GC : General Call flag.
- ARM_I2C_EVENT_TRANSFER_DONE : Master/Slave Transmit/Receive finished.
- ARM_I2C_EVENT_TRANSFER_INCOMPLETE : Master/Slave Transmit/Receive incomplete transfer.
- ARM_I2C_EVENT_SLAVE_TRANSMIT : Slave Transmit operation requested.
- ARM_I2C_EVENT_SLAVE_RECEIVE : Slave Receive operation requested.
- ARM_I2C_EVENT_ADDRESS_NACK : Address not acknowledged from Slave.
- ARM_I2C_EVENT_GENERAL_CALL : General Call indication.
- ARM_I2C_EVENT_ARBITRATION_LOST : Master lost arbitration.
- ARM_I2C_EVENT_BUS_ERROR : Bus error detected (START/STOP at illegal position)
- ARM_I2C_EVENT_BUS_CLEAR : Bus clear finished.

Functions

- ARM_I2C_GetVersion : Get driver version.
- ARM_I2C_GetCapabilities : Get driver capabilities.
- ARM_I2C_Initialize : Initialize I2C Interface.
- ARM_I2C_Uninitialize : De-initialize I2C Interface.
- ARM_I2C_PowerControl : Control I2C Interface Power.
- ARM_I2C_MasterTransmit : Start transmitting data as I2C Master.
- ARM_I2C_MasterReceive : Start receiving data as I2C Master.
- ARM_I2C_SlaveTransmit : Start transmitting data as I2C Slave.

- ARM_I2C_SlaveReceive : Start receiving data as I2C Slave.
- ARM_I2C_GetDataCount : Get transferred data count.
- ARM_I2C_Control : Control I2C Interface.
- ARM_I2C_GetStatus : Get I2C status.
- ARM_I2C_SignalEvent : Signal I2C Events.

20.8.2 CMSIS I2C Driver Typedef Documentation

20.8.2.1 ARM_I2C_STATUS

```
typedef struct _ARM_I2C_STATUS ARM_I2C_STATUS
```

Location: Driver_I2C.h:112

I2C Status.

20.8.2.2 ARM_I2C_SignalEvent_t

```
typedef void(* ARM_I2C_SignalEvent_t
```

Location: Driver_I2C.h:198

Pointer to ARM_I2C_SignalEvent : Signal I2C Event.

20.8.2.3 ARM_I2C_CAPABILITIES

```
typedef struct _ARM_I2C_CAPABILITIES ARM_I2C_CAPABILITIES
```

Location: Driver_I2C.h:207

I2C Driver Capabilities.

20.8.2.4 ARM_DRIVER_I2C

```
typedef struct _ARM_DRIVER_I2C ARM_DRIVER_I2C
```

Location: Driver_I2C.h:226

Access structure of the I2C Driver.

20.8.3 CMSIS I2C Driver Data Structures Type Documentation

20.8.3.1 _ARM_I2C_STATUS

Location: Driver_I2C.h:104

I2C Status.

Data Fields

Type	Name	Description
uint32_t	*busy*	Busy flag.
uint32_t	*mode*	Mode: 0=Slave, 1=Master.
uint32_t	*direction*	Direction: 0=Transmitter, 1=Receiver.
uint32_t	*general_call*	General Call indication (cleared on start of next Slave operation)
uint32_t	*arbitration_lost*	Master lost arbitration (cleared on start of next Master operation)
uint32_t	*bus_error*	Bus error detected (cleared on start of next Master/Slave operation)
uint32_t	*reserved*	(Reserved for future use)

20.8.3.2 _ARM_I2C_CAPABILITIES

Location: Driver_I2C.h:204

I2C Driver Capabilities.

Data Fields

Type	Name	Description
uint32_t	*address_10_bit*	Supports 10-bit addressing.
uint32_t	*reserved*	Reserved (must be zero)

20.8.3.3 _ARM_DRIVER_I2C

Location: Driver_I2C.h:213

Access structure of the I2C Driver.

Data Fields

Type	*Name*	Description
ARM_DRIVER_VERSION(*	*GetVersion)(void)*	Pointer to ARM_I2C_GetVersion : Get driver version.
ARM_I2C_CAPABILITIES(*	*GetCapabilities)(void)*	Pointer to ARM_I2C_GetCapabilities : Get driver capabilities.
`int32_t(*`	*Initialize)(ARM_I2C_SignalEvent_t cb_event)*	Pointer to ARM_I2C_Initialize : Initialize I2C Interface.
`int32_t(*`	*Uninitialize)(void)*	Pointer to ARM_I2C_Uninitialize : De-initialize I2C Interface.
`int32_t(*`	*PowerControl)(ARM_POWER_STATE state)*	Pointer to ARM_I2C_PowerControl : Control I2C Interface Power.
`int32_t(*`	*MasterTransmit)(uint32_t addr, const uint8_t *data, uint32_t num, bool xfer_pending)*	Pointer to ARM_I2C_MasterTransmit : Start transmitting data as I2C Master.
`int32_t(*`	*MasterReceive)(uint32_t addr, uint8_t *data, uint32_t num, bool xfer_pending)*	Pointer to ARM_I2C_MasterReceive : Start receiving data as I2C Master.
`int32_t(*`	*SlaveTransmit)(const uint8_t *data, uint32_t num)*	Pointer to ARM_I2C_SlaveTransmit : Start transmitting data as I2C Slave.
`int32_t(*`	*SlaveReceive)(uint8_t *data, uint32_t num)*	Pointer to ARM_I2C_SlaveReceive : Start receiving data as I2C Slave.
`int32_t(*`	*GetDataCount)(void)*	Pointer to ARM_I2C_GetDataCount : Get transferred data count.
`int32_t(*`	*Control)(uint32_t control, uint32_t arg)*	Pointer to ARM_I2C_Control : Control I2C Interface.
ARM_I2C_STATUS(*	*GetStatus)(void)*	Pointer to ARM_I2C_GetStatus : Get I2C status.

20.8.4 CMSIS I2C Driver Macro Definition Documentation

20.8.4.1 ARM_I2C_API_VERSION

```
#define ARM_I2C_API_VERSION ARM_DRIVER_VERSION_MAJOR_MINOR(2,3)
```

Location: Driver_I2C.h:78

I2C API version.

20.8.4.2 ARM_I2C_OWN_ADDRESS

```
#define ARM_I2C_OWN_ADDRESS (0x01)
```

Location: Driver_I2C.h:83

Set Own Slave Address; arg = address.

20.8.4.3 ARM_I2C_BUS_SPEED

```
#define ARM_I2C_BUS_SPEED (0x02)
```

Location: Driver_I2C.h:84

Set Bus Speed; arg = speed.

20.8.4.4 ARM_I2C_BUS_CLEAR

```
#define ARM_I2C_BUS_CLEAR (0x03)
```

Location: Driver_I2C.h:85

Execute Bus clear: send nine clock pulses.

20.8.4.5 ARM_I2C_ABORT_TRANSFER

```
#define ARM_I2C_ABORT_TRANSFER (0x04)
```

Location: Driver_I2C.h:86

Abort Master/Slave Transmit/Receive.

20.8.4.6 ARM_I2C_BUS_SPEED_STANDARD

```
#define ARM_I2C_BUS_SPEED_STANDARD (0x01)
```

Location: Driver_I2C.h:89

Standard Speed (100kHz)

20.8.4.7 ARM_I2C_BUS_SPEED_FAST

```
#define ARM_I2C_BUS_SPEED_FAST (0x02)
```

Location: Driver_I2C.h:90

Fast Speed (400kHz)

20.8.4.8 ARM_I2C_BUS_SPEED_FAST_PLUS

```
#define ARM_I2C_BUS_SPEED_FAST_PLUS (0x03)
```

Location: Driver_I2C.h:91

Fast+ Speed (1MHz)

20.8.4.9 ARM_I2C_BUS_SPEED_HIGH

```
#define ARM_I2C_BUS_SPEED_HIGH (0x04)
```

Location: Driver_I2C.h:92

High Speed (3.4MHz)

20.8.4.10 ARM_I2C_ADDRESS_10BIT

```
#define ARM_I2C_ADDRESS_10BIT (0x0400)
```

Location: Driver_I2C.h:97

10-bit address flag

20.8.4.11 ARM_I2C_ADDRESS_GC

```
#define ARM_I2C_ADDRESS_GC (0x8000)
```

Location: Driver_I2C.h:98

General Call flag.

20.8.4.12 ARM_I2C_EVENT_TRANSFER_DONE

```
#define ARM_I2C_EVENT_TRANSFER_DONE (1UL << 0)
```

Location: Driver_I2C.h:116

Master/Slave Transmit/Receive finished.

I2C Event

20.8.4.13 ARM_I2C_EVENT_TRANSFER_INCOMPLETE

```
#define ARM_I2C_EVENT_TRANSFER_INCOMPLETE (1UL << 1)
```

Location: Driver_I2C.h:117

Master/Slave Transmit/Receive incomplete transfer.

20.8.4.14 ARM_I2C_EVENT_SLAVE_TRANSMIT

```
#define ARM_I2C_EVENT_SLAVE_TRANSMIT (1UL << 2)
```

Location: Driver_I2C.h:118

Slave Transmit operation requested.

20.8.4.15 ARM_I2C_EVENT_SLAVE_RECEIVE

```
#define ARM_I2C_EVENT_SLAVE_RECEIVE (1UL << 3)
```

Location: Driver_I2C.h:119

Slave Receive operation requested.

20.8.4.16 ARM_I2C_EVENT_ADDRESS_NACK

```
#define ARM_I2C_EVENT_ADDRESS_NACK (1UL << 4)
```

Location: Driver_I2C.h:120

Address not acknowledged from Slave.

20.8.4.17 ARM_I2C_EVENT_GENERAL_CALL

```
#define ARM_I2C_EVENT_GENERAL_CALL (1UL << 5)
```

Location: Driver_I2C.h:121

General Call indication.

20.8.4.18 ARM_I2C_EVENT_ARBITRATION_LOST

```
#define ARM_I2C_EVENT_ARBITRATION_LOST (1UL << 6)
```

Location: Driver_I2C.h:122

Master lost arbitration.

20.8.4.19 ARM_I2C_EVENT_BUS_ERROR

```
#define ARM_I2C_EVENT_BUS_ERROR (1UL << 7)
```

Location: Driver_I2C.h:123

Bus error detected (START/STOP at illegal position)

20.8.4.20 ARM_I2C_EVENT_BUS_CLEAR

```
#define ARM_I2C_EVENT_BUS_CLEAR (1UL << 8)
```

Location: Driver_I2C.h:124

Bus clear finished.

20.8.5 CMSIS I2C Driver Function Documentation

20.8.5.1 ARM_I2C_GetVersion

```
ARM_DRIVER_VERSION ARM_I2C_GetVersion()
```

Location: Driver_I2C.c:38

Get driver version.

Returns:

ARM_DRIVER_VERSION

20.8.5.2 ARM_I2C_GetCapabilities

```
ARM_I2C_CAPABILITIES ARM_I2C_GetCapabilities()
```

Location: Driver_I2C.c:43

Get driver capabilities.

Returns:

ARM_I2C_CAPABILITIES

20.8.5.3 ARM_I2C_Initialize

```
int32_t ARM_I2C_Initialize(ARM_I2C_SignalEvent_t cb_event)
```

Location: Driver_I2C.c:48

Initialize I2C Interface.

Returns:

execution_status

Parameters

Direction	Name	Description
in	*cb_event*	Pointer to ARM_I2C_SignalEvent

20.8.5.4 ARM_I2C_Uninitialize

```
int32_t ARM_I2C_Uninitialize()
```

Location: Driver_I2C.c:52

De-initialize I2C Interface.

Returns:

execution_status

20.8.5.5 ARM_I2C_PowerControl

```
int32_t ARM_I2C_PowerControl(ARM_POWER_STATE state)
```

Location: Driver_I2C.c:56

Control I2C Interface Power.

Returns:

execution_status

Parameters

Direction	Name	Description
in	*state*	Power state

20.8.5.6 ARM_I2C_MasterTransmit

```
 int32_t ARM_I2C_MasterTransmit(uint32_t addr, const uint8_t * data, uint32_t
num, bool xfer_pending)
```

Location: Driver_I2C.c:72

Start transmitting data as I2C Master.

Returns:

execution_status

Parameters

Direction	Name	Description
in	*addr*	Slave address (7-bit or 10-bit)
in	*data*	Pointer to buffer with data to transmit to I2C Slave
in	*num*	Number of data bytes to transmit
in	*xfer_pending*	Transfer operation is pending - Stop condition will not be generated

20.8.5.7 ARM_I2C_MasterReceive

```
 int32_t ARM_I2C_MasterReceive(uint32_t addr, uint8_t * data, uint32_t num, bool
xfer_pending)
```

Location: Driver_I2C.c:76

Start receiving data as I2C Master.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*addr*	Slave address (7-bit or 10-bit)
`out`	*data*	Pointer to buffer for data to receive from I2C Slave
`in`	*num*	Number of data bytes to receive
`in`	*xfer_pending*	Transfer operation is pending - Stop condition will not be generated

20.8.5.8 ARM_I2C_SlaveTransmit

```
 int32_t ARM_I2C_SlaveTransmit(const uint8_t * data, uint32_t num)
```

Location: Driver_I2C.c:80

Start transmitting data as I2C Slave.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*data*	Pointer to buffer with data to transmit to I2C Master
`in`	*num*	Number of data bytes to transmit

20.8.5.9 ARM_I2C_SlaveReceive

```
int32_t ARM_I2C_SlaveReceive(uint8_t * data, uint32_t num)
```

Location: Driver_I2C.c:84

Start receiving data as I2C Slave.

Returns:

execution_status

Parameters

Direction	Name	Description
`out`	*data*	Pointer to buffer for data to receive from I2C Master
`in`	*num*	Number of data bytes to receive

20.8.5.10 ARM_I2C_GetDataCount

```
int32_t ARM_I2C_GetDataCount()
```

Location: Driver_I2C.c:88

Get transferred data count.

Returns:

number of data bytes transferred; -1 when Slave is not addressed by Master

20.8.5.11 ARM_I2C_Control

```
int32_t ARM_I2C_Control(uint32_t control, uint32_t arg)
```

Location: Driver_I2C.c:92

Control I2C Interface.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*control*	Operation
`in`	*arg*	Argument of operation (optional)

20.8.5.12 ARM_I2C_GetStatus

```
ARM_I2C_STATUS ARM_I2C_GetStatus()
```

Location: Driver_I2C.c:124

Get I2C status.

Returns:

I2C status ARM_I2C_STATUS

20.8.5.13 ARM_I2C_SignalEvent

```
void ARM_I2C_SignalEvent(uint32_t event)
```

Location: Driver_I2C.c:128

Signal I2C Events.

Parameters

Direction	Name	Description
`in`	*event*	I2C_events notification mask

20.9 CMSIS SPI Driver

CMSIS SPI Driver Reference.

20.9.1 Summary

Typedefs

- ARM_SPI_STATUS : SPI Status.
- ARM_SPI_SignalEvent_t : Pointer to ARM_SPI_SignalEvent : Signal SPI Event.
- ARM_SPI_CAPABILITIES : SPI Driver Capabilities.
- ARM_DRIVER_SPI : Access structure of the SPI Driver.

Data Structures

- _ARM_SPI_STATUS : SPI Status.
- _ARM_SPI_CAPABILITIES : SPI Driver Capabilities.
- _ARM_DRIVER_SPI : Access structure of the SPI Driver.

Macros

- ARM_SPI_API_VERSION : SPI API version.
- ARM_SPI_CONTROL_Pos : Position of the 0th bit of the SPI Control field in the ARM_SPI structure.
- ARM_SPI_CONTROL_Msk : Positioning of SPI Control field in the ARM_SPI structure.
- ARM_SPI_MODE_INACTIVE : SPI Inactive.
- ARM_SPI_MODE_MASTER : SPI Master (Output on MOSI, Input on MISO); arg = Bus Speed in bps.
- ARM_SPI_MODE_SLAVE : SPI Slave (Output on MISO, Input on MOSI).
- ARM_SPI_MODE_MASTER_SIMPLEX : SPI Master (Output/Input on MOSI); arg = Bus Speed in bps.
- ARM_SPI_MODE_SLAVE_SIMPLEX : SPI Slave (Output/Input on MISO).
- ARM_SPI_FRAME_FORMAT_Pos : Position of the 0th bit of the Frame format field in the ARM_SPI structure.

- ARM_SPI_FRAME_FORMAT_Msk : Positioning of Frame format field in the ARM_SPI structure.
- ARM_SPI_CPOL0_CPHA0 : Clock Polarity 0, Clock Phase 0 (default).
- ARM_SPI_CPOL0_CPHA1 : Clock Polarity 0, Clock Phase 1.
- ARM_SPI_CPOL1_CPHA0 : Clock Polarity 1, Clock Phase 0.
- ARM_SPI_CPOL1_CPHA1 : Clock Polarity 1, Clock Phase 1.
- ARM_SPI_TI_SSI : Texas Instruments Frame Format.
- ARM_SPI_MICROWIRE : National Microwire Frame Format.
- ARM_SPI_DATA_BITS_Pos : Position of the 0th bit of the Data bits field in the ARM_SPI structure.
- ARM_SPI_DATA_BITS_Msk : Positioning of the Data bits field in the ARM_SPI structure.
- ARM_SPI_DATA_BITS : Number of Data bits.
- ARM_SPI_BIT_ORDER_Pos : Position of the 0th bit of the Bit order field in the ARM_SPI structure.
- ARM_SPI_BIT_ORDER_Msk : Positioning of the Bit order field in the ARM_SPI structure.
- ARM_SPI_MSB_LSB : SPI Bit order from MSB to LSB (default).
- ARM_SPI_LSB_MSB : SPI Bit order from LSB to MSB.
- ARM_SPI_SS_MASTER_MODE_Pos : Position of the 0th bit of the Slave Select Master Mode field in the ARM_SPI structure.
- ARM_SPI_SS_MASTER_MODE_Msk : Positioning of the Slave Select Master Mode field in the ARM_SPI structure.
- ARM_SPI_SS_MASTER_UNUSED : SPI Slave Select when Master: Not used (default).
- ARM_SPI_SS_MASTER_SW : SPI Slave Select when Master: Software controlled.
- ARM_SPI_SS_MASTER_HW_OUTPUT : SPI Slave Select when Master: Hardware controlled Output.
- ARM_SPI_SS_MASTER_HW_INPUT : SPI Slave Select when Master: Hardware monitored Input.
- ARM_SPI_SS_SLAVE_MODE_Pos : Position of the 0th bit of the Slave Select Slave Mode field in the ARM_SPI structure.
- ARM_SPI_SS_SLAVE_MODE_Msk : Positioning of the Slave Select Slave mode field in the ARM_SPI structure.
- ARM_SPI_SS_SLAVE_HW : SPI Slave Select when Slave: Hardware monitored (default).
- ARM_SPI_SS_SLAVE_SW : SPI Slave Select when Slave: Software controlled.
- ARM_SPI_SET_BUS_SPEED : Set Bus Speed in bps; arg = value.
- ARM_SPI_GET_BUS_SPEED : Get Bus Speed in bps.
- ARM_SPI_SET_DEFAULT_TX_VALUE : Set default Transmit value; arg = value.
- ARM_SPI_CONTROL_SS : Control Slave Select; arg: 0=inactive, 1=active.
- ARM_SPI_ABORT_TRANSFER : Abort current data transfer.
- ARM_SPI_SS_INACTIVE : SPI Slave Select Signal Inactive.
- ARM_SPI_SS_ACTIVE : SPI Slave Select Signal Active.
- ARM_SPI_ERROR_MODE : Specified Mode not supported.
- ARM_SPI_ERROR_FRAME_FORMAT : Specified Frame Format not supported.
- ARM_SPI_ERROR_DATA_BITS : Specified number of Data bits not supported.
- ARM_SPI_ERROR_BIT_ORDER : Specified Bit order not supported.
- ARM_SPI_ERROR_SS_MODE : Specified Slave Select Mode not supported.
- ARM_SPI_EVENT_TRANSFER_COMPLETE : Data Transfer completed.
- ARM_SPI_EVENT_DATA_LOST : Data lost: Receive overflow / Transmit underflow.
- ARM_SPI_EVENT_MODE_FAULT : Master Mode Fault (SS deactivated when Master).

Functions

- ARM_SPI_GetVersion : Get driver version.
- ARM_SPI_GetCapabilities : Get driver capabilities.

- ARM_SPI_Initialize : Initialize SPI Interface.
- ARM_SPI_Uninitialize : De-initialize SPI Interface.
- ARM_SPI_PowerControl : Control SPI Interface Power.
- ARM_SPI_Send : Start sending data to SPI transmitter.
- ARM_SPI_Receive : Start receiving data from SPI receiver.
- ARM_SPI_Transfer : Start sending/receiving data to/from SPI transmitter/receiver.
- ARM_SPI_GetDataCount : Get transferred data count.
- ARM_SPI_Control : Control SPI Interface.
- ARM_SPI_GetStatus : Get SPI status.
- ARM_SPI_SignalEvent : Signal SPI Events.

20.9.2 CMSIS SPI Driver Typedef Documentation

20.9.2.1 ARM_SPI_STATUS

```
typedef struct _ARM_SPI_STATUS ARM_SPI_STATUS
```

Location: Driver_SPI.h:148

SPI Status.

20.9.2.2 ARM_SPI_SignalEvent_t

```
typedef void(* ARM_SPI_SignalEvent_t
```

Location: Driver_SPI.h:222

Pointer to ARM_SPI_SignalEvent : Signal SPI Event.

20.9.2.3 ARM_SPI_CAPABILITIES

```
typedef struct _ARM_SPI_CAPABILITIES ARM_SPI_CAPABILITIES
```

Location: Driver_SPI.h:234

SPI Driver Capabilities.

20.9.2.4 ARM_DRIVER_SPI

```
typedef struct _ARM_DRIVER_SPI ARM_DRIVER_SPI
```

Location: Driver_SPI.h:253

Access structure of the SPI Driver.

20.9.3 CMSIS SPI Driver Data Structures Type Documentation

20.9.3.1 _ARM_SPI_STATUS

Location: Driver_SPI.h:143

SPI Status.

Data Fields

Type	*Name*	Description
`uint32_t`	*busy*	Transmitter/Receiver busy flag.
`uint32_t`	*data_lost*	Data lost: Receive overflow / Transmit underflow (cleared on start of transfer operation).
`uint32_t`	*mode_fault*	Mode fault detected; optional (cleared on start of transfer operation).
`uint32_t`	*reserved*	(Reserved for future use)

20.9.3.2 _ARM_SPI_CAPABILITIES

Location: Driver_SPI.h:228

SPI Driver Capabilities.

Data Fields

Type	*Name*	Description
`uint32_t`	*simplex*	supports Simplex Mode (Master and Slave).
`uint32_t`	*ti_ssi*	supports TI Synchronous Serial Interface.
`uint32_t`	*microwire*	supports Microwire Interface.
`uint32_t`	*event_mode_fault*	Signal Mode Fault event: ARM_SPI_EVENT_MODE_FAULT.
`uint32_t`	*reserved*	Reserved (must be zero).

20.9.3.3 _ARM_DRIVER_SPI

Location: Driver_SPI.h:239

Access structure of the SPI Driver.

Data Fields

Type	*Name*	Description
`ARM_DRIVER_VERSION(*`	*GetVersion)(void)*	Pointer to ARM_SPI_GetVersion : Get driver version.
`ARM_SPI_CAPABILITIES(*`	*GetCapabilities)(void)*	Pointer to ARM_SPI_GetCapabilities : Get driver capabilities.
`int32_t(*`	*Initialize)(ARM_SPI_SignalEvent_t cb_event)*	Pointer to ARM_SPI_Initialize : Initialize SPI Interface.
`int32_t(*`	*Uninitialize)(void)*	Pointer to ARM_SPI_Uninitialize : De-initialize SPI Interface.
`int32_t(*`	*PowerControl)(ARM_POWER_STATE state)*	Pointer to ARM_SPI_PowerControl : Control SPI Interface Power.
`int32_t(*`	*Send)(const void *data, uint32_t num)*	Pointer to ARM_SPI_Send : Start sending data to SPI Interface.
`int32_t(*`	*Receive)(void *data, uint32_t num)*	Pointer to ARM_SPI_Receive : Start receiving data from SPI Interface.
`int32_t(*`	*Transfer)(const void *data_out, void *data_*	Pointer to ARM_SPI_Transfer : Start sending/receiving data to/from SPI.

	in, uint32_t num)	
`uint32_t(*`	*GetDataCount)(void)*	Pointer to ARM_SPI_GetDataCount : Get transferred data count.
`int32_t(*`	*Control)(uint32_t control, uint32_t arg)*	Pointer to ARM_SPI_Control : Control SPI Interface.
`ARM_SPI_STATUS(*`	*GetStatus)(void)*	Pointer to ARM_SPI_GetStatus : Get SPI status.

20.9.4 CMSIS SPI Driver Macro Definition Documentation

20.9.4.1 ARM_SPI_API_VERSION

```
#define ARM_SPI_API_VERSION ARM_DRIVER_VERSION_MAJOR_MINOR(2,2)
```

Location: Driver_SPI.h:69

SPI API version.

20.9.4.2 ARM_SPI_CONTROL_Pos

```
#define ARM_SPI_CONTROL_Pos 0
```

Location: Driver_SPI.h:74

Position of the 0th bit of the SPI Control field in the ARM_SPI structure.

20.9.4.3 ARM_SPI_CONTROL_Msk

```
#define ARM_SPI_CONTROL_Msk (0xFFUL << ARM_SPI_CONTROL_Pos)
```

Location: Driver_SPI.h:75

Positioning of SPI Control field in the ARM_SPI structure.

20.9.4.4 ARM_SPI_MODE_INACTIVE

```
#define ARM_SPI_MODE_INACTIVE (0x00UL << ARM_SPI_CONTROL_Pos)
```

Location: Driver_SPI.h:78

SPI Inactive.

20.9.4.5 ARM_SPI_MODE_MASTER

```
#define ARM_SPI_MODE_MASTER (0x01UL << ARM_SPI_CONTROL_Pos)
```

Location: Driver_SPI.h:79

SPI Master (Output on MOSI, Input on MISO); arg = Bus Speed in bps.

20.9.4.6 ARM_SPI_MODE_SLAVE

```
#define ARM_SPI_MODE_SLAVE (0x02UL << ARM_SPI_CONTROL_Pos)
```

Location: Driver_SPI.h:80

SPI Slave (Output on MISO, Input on MOSI).

20.9.4.7 ARM_SPI_MODE_MASTER_SIMPLEX

```
#define ARM_SPI_MODE_MASTER_SIMPLEX (0x03UL << ARM_SPI_CONTROL_Pos)
```

Location: Driver_SPI.h:81

SPI Master (Output/Input on MOSI); arg = Bus Speed in bps.

20.9.4.8 ARM_SPI_MODE_SLAVE_SIMPLEX

```
#define ARM_SPI_MODE_SLAVE_SIMPLEX (0x04UL << ARM_SPI_CONTROL_Pos)
```

Location: Driver_SPI.h:82

SPI Slave (Output/Input on MISO).

20.9.4.9 ARM_SPI_FRAME_FORMAT_Pos

```
#define ARM_SPI_FRAME_FORMAT_Pos 8
```

Location: Driver_SPI.h:85

Position of the 0th bit of the Frame format field in the ARM_SPI structure.

20.9.4.10 ARM_SPI_FRAME_FORMAT_Msk

```
#define ARM_SPI_FRAME_FORMAT_Msk (7UL << ARM_SPI_FRAME_FORMAT_Pos)
```

Location: Driver_SPI.h:86

Positioning of Frame format field in the ARM_SPI structure.

20.9.4.11 ARM_SPI_CPOL0_CPHA0

```
#define ARM_SPI_CPOL0_CPHA0 (0UL << ARM_SPI_FRAME_FORMAT_Pos)
```

Location: Driver_SPI.h:87

Clock Polarity 0, Clock Phase 0 (default).

20.9.4.12 ARM_SPI_CPOL0_CPHA1

```
#define ARM_SPI_CPOL0_CPHA1 (1UL << ARM_SPI_FRAME_FORMAT_Pos)
```

Location: Driver_SPI.h:88

Clock Polarity 0, Clock Phase 1.

20.9.4.13 ARM_SPI_CPOL1_CPHA0

```
#define ARM_SPI_CPOL1_CPHA0 (2UL << ARM_SPI_FRAME_FORMAT_Pos)
```

Location: Driver_SPI.h:89

Clock Polarity 1, Clock Phase 0.

20.9.4.14 ARM_SPI_CPOL1_CPHA1

```
#define ARM_SPI_CPOL1_CPHA1 (3UL << ARM_SPI_FRAME_FORMAT_Pos)
```

Location: Driver_SPI.h:90

Clock Polarity 1, Clock Phase 1.

20.9.4.15 ARM_SPI_TI_SSI

```
#define ARM_SPI_TI_SSI (4UL << ARM_SPI_FRAME_FORMAT_Pos)
```

Location: Driver_SPI.h:91

Texas Instruments Frame Format.

20.9.4.16 ARM_SPI_MICROWIRE

```
#define ARM_SPI_MICROWIRE (5UL << ARM_SPI_FRAME_FORMAT_Pos)
```

Location: Driver_SPI.h:92

National Microwire Frame Format.

20.9.4.17 ARM_SPI_DATA_BITS_Pos

```
#define ARM_SPI_DATA_BITS_Pos 12
```

Location: Driver_SPI.h:95

Position of the 0th bit of the Data bits field in the ARM_SPI structure.

20.9.4.18 ARM_SPI_DATA_BITS_Msk

```
#define ARM_SPI_DATA_BITS_Msk (0x3FUL << ARM_SPI_DATA_BITS_Pos)
```

Location: Driver_SPI.h:96

Positioning of the Data bits field in the ARM_SPI structure.

20.9.4.19 ARM_SPI_DATA_BITS

```
#define ARM_SPI_DATA_BITS (((n) & 0x3F) << ARM_SPI_DATA_BITS_Pos)
```

Location: Driver_SPI.h:97

Number of Data bits.

20.9.4.20 ARM_SPI_BIT_ORDER_Pos

```
#define ARM_SPI_BIT_ORDER_Pos 18
```

Location: Driver_SPI.h:100

Position of the 0th bit of the Bit order field in the ARM_SPI structure.

20.9.4.21 ARM_SPI_BIT_ORDER_Msk

```
#define ARM_SPI_BIT_ORDER_Msk (1UL << ARM_SPI_BIT_ORDER_Pos)
```

Location: Driver_SPI.h:101

Positioning of the Bit order field in the ARM_SPI structure.

20.9.4.22 ARM_SPI_MSB_LSB

```
#define ARM_SPI_MSB_LSB (0UL << ARM_SPI_BIT_ORDER_Pos)
```

Location: Driver_SPI.h:102

SPI Bit order from MSB to LSB (default).

20.9.4.23 ARM_SPI_LSB_MSB

```
#define ARM_SPI_LSB_MSB (1UL << ARM_SPI_BIT_ORDER_Pos)
```

Location: Driver_SPI.h:103

SPI Bit order from LSB to MSB.

20.9.4.24 ARM_SPI_SS_MASTER_MODE_Pos

```
#define ARM_SPI_SS_MASTER_MODE_Pos 19
```

Location: Driver_SPI.h:106

Position of the 0th bit of the Slave Select Master Mode field in the ARM_SPI structure.

20.9.4.25 ARM_SPI_SS_MASTER_MODE_Msk

```
#define ARM_SPI_SS_MASTER_MODE_Msk (3UL << ARM_SPI_SS_MASTER_MODE_Pos)
```

Location: Driver_SPI.h:107

Positioning of the Slave Select Master Mode field in the ARM_SPI structure.

20.9.4.26 ARM_SPI_SS_MASTER_UNUSED

```
#define ARM_SPI_SS_MASTER_UNUSED (0UL << ARM_SPI_SS_MASTER_MODE_Pos)
```

Location: Driver_SPI.h:108

SPI Slave Select when Master: Not used (default).

20.9.4.27 ARM_SPI_SS_MASTER_SW

```
#define ARM_SPI_SS_MASTER_SW (1UL << ARM_SPI_SS_MASTER_MODE_Pos)
```

Location: Driver_SPI.h:109

SPI Slave Select when Master: Software controlled.

20.9.4.28 ARM_SPI_SS_MASTER_HW_OUTPUT

```
#define ARM_SPI_SS_MASTER_HW_OUTPUT (2UL << ARM_SPI_SS_MASTER_MODE_Pos)
```

Location: Driver_SPI.h:110

SPI Slave Select when Master: Hardware controlled Output.

20.9.4.29 ARM_SPI_SS_MASTER_HW_INPUT

```
#define ARM_SPI_SS_MASTER_HW_INPUT (3UL << ARM_SPI_SS_MASTER_MODE_Pos)
```

Location: Driver_SPI.h:111

SPI Slave Select when Master: Hardware monitored Input.

20.9.4.30 ARM_SPI_SS_SLAVE_MODE_Pos

```
#define ARM_SPI_SS_SLAVE_MODE_Pos 21
```

Location: Driver_SPI.h:112

Position of the 0th bit of the Slave Select Slave Mode field in the ARM_SPI structure.

20.9.4.31 ARM_SPI_SS_SLAVE_MODE_Msk

```
#define ARM_SPI_SS_SLAVE_MODE_Msk (1UL << ARM_SPI_SS_SLAVE_MODE_Pos)
```

Location: Driver_SPI.h:113

Positioning of the Slave Select Slave mode field in the ARM_SPI structure.

20.9.4.32 ARM_SPI_SS_SLAVE_HW

```
#define ARM_SPI_SS_SLAVE_HW (0UL << ARM_SPI_SS_SLAVE_MODE_Pos)
```

Location: Driver_SPI.h:114

SPI Slave Select when Slave: Hardware monitored (default).

20.9.4.33 ARM_SPI_SS_SLAVE_SW

```
#define ARM_SPI_SS_SLAVE_SW (1UL << ARM_SPI_SS_SLAVE_MODE_Pos)
```

Location: Driver_SPI.h:115

SPI Slave Select when Slave: Software controlled.

20.9.4.34 ARM_SPI_SET_BUS_SPEED

```
#define ARM_SPI_SET_BUS_SPEED (0x10UL << ARM_SPI_CONTROL_Pos)
```

Location: Driver_SPI.h:119

Set Bus Speed in bps; arg = value.

20.9.4.35 ARM_SPI_GET_BUS_SPEED

```
#define ARM_SPI_GET_BUS_SPEED (0x11UL << ARM_SPI_CONTROL_Pos)
```

Location: Driver_SPI.h:120

Get Bus Speed in bps.

20.9.4.36 ARM_SPI_SET_DEFAULT_TX_VALUE

```
#define ARM_SPI_SET_DEFAULT_TX_VALUE (0x12UL << ARM_SPI_CONTROL_Pos)
```

Location: Driver_SPI.h:121

Set default Transmit value; arg = value.

20.9.4.37 ARM_SPI_CONTROL_SS

```
#define ARM_SPI_CONTROL_SS (0x13UL << ARM_SPI_CONTROL_Pos)
```

Location: Driver_SPI.h:122

Control Slave Select; arg: 0=inactive, 1=active.

20.9.4.38 ARM_SPI_ABORT_TRANSFER

```
#define ARM_SPI_ABORT_TRANSFER (0x14UL << ARM_SPI_CONTROL_Pos)
```

Location: Driver_SPI.h:123

Abort current data transfer.

20.9.4.39 ARM_SPI_SS_INACTIVE

```
#define ARM_SPI_SS_INACTIVE 0
```

Location: Driver_SPI.h:127

SPI Slave Select Signal Inactive.

20.9.4.40 ARM_SPI_SS_ACTIVE

```
#define ARM_SPI_SS_ACTIVE 1
```

Location: Driver_SPI.h:128

SPI Slave Select Signal Active.

20.9.4.41 ARM_SPI_ERROR_MODE

```
#define ARM_SPI_ERROR_MODE (ARM_DRIVER_ERROR_SPECIFIC - 1)
```

Location: Driver_SPI.h:132

Specified Mode not supported.

20.9.4.42 ARM_SPI_ERROR_FRAME_FORMAT

```
#define ARM_SPI_ERROR_FRAME_FORMAT (ARM_DRIVER_ERROR_SPECIFIC - 2)
```

Location: Driver_SPI.h:133

Specified Frame Format not supported.

20.9.4.43 ARM_SPI_ERROR_DATA_BITS

```
#define ARM_SPI_ERROR_DATA_BITS (ARM_DRIVER_ERROR_SPECIFIC - 3)
```

Location: Driver_SPI.h:134

Specified number of Data bits not supported.

20.9.4.44 ARM_SPI_ERROR_BIT_ORDER

```
#define ARM_SPI_ERROR_BIT_ORDER (ARM_DRIVER_ERROR_SPECIFIC - 4)
```

Location: Driver_SPI.h:135

Specified Bit order not supported.

20.9.4.45 ARM_SPI_ERROR_SS_MODE

```
#define ARM_SPI_ERROR_SS_MODE (ARM_DRIVER_ERROR_SPECIFIC - 5)
```

Location: Driver_SPI.h:136

Specified Slave Select Mode not supported.

20.9.4.46 ARM_SPI_EVENT_TRANSFER_COMPLETE

```
#define ARM_SPI_EVENT_TRANSFER_COMPLETE (1UL << 0)
```

Location: Driver_SPI.h:152

Data Transfer completed.

SPI Event

20.9.4.47 ARM_SPI_EVENT_DATA_LOST

```
#define ARM_SPI_EVENT_DATA_LOST (1UL << 1)
```

Location: Driver_SPI.h:153

Data lost: Receive overflow / Transmit underflow.

20.9.4.48 ARM_SPI_EVENT_MODE_FAULT

```
#define ARM_SPI_EVENT_MODE_FAULT (1UL << 2)
```

Location: Driver_SPI.h:154

Master Mode Fault (SS deactivated when Master).

20.9.5 CMSIS SPI Driver Function Documentation

20.9.5.1 ARM_SPI_GetVersion

```
ARM_DRIVER_VERSION ARM_SPI_GetVersion()
```

Location: Driver_SPI.c:42

Get driver version.

Returns:

ARM_DRIVER_VERSION

20.9.5.2 ARM_SPI_GetCapabilities

```
ARM_SPI_CAPABILITIES ARM_SPI_GetCapabilities()
```

Location: Driver_SPI.c:47

Get driver capabilities.

Returns:

ARM_SPI_CAPABILITIES

20.9.5.3 ARM_SPI_Initialize

```
int32_t ARM_SPI_Initialize(ARM_SPI_SignalEvent_t cb_event)
```

Location: Driver_SPI.c:52

Initialize SPI Interface.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*cb_event*	Pointer to ARM_SPI_SignalEvent

20.9.5.4 ARM_SPI_Uninitialize

```
int32_t ARM_SPI_Uninitialize()
```

Location: Driver_SPI.c:56

De-initialize SPI Interface.

Returns:

execution_status

20.9.5.5 ARM_SPI_PowerControl

```
int32_t ARM_SPI_PowerControl(ARM_POWER_STATE state)
```

Location: Driver_SPI.c:60

Control SPI Interface Power.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*state*	Power state

20.9.5.6 ARM_SPI_Send

```
int32_t ARM_SPI_Send(const void * data, uint32_t num)
```

Location: Driver_SPI.c:76

Start sending data to SPI transmitter.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*data*	Pointer to buffer with data to send to SPI transmitter
`in`	*num*	Number of data items to send

20.9.5.7 ARM_SPI_Receive

```
int32_t ARM_SPI_Receive(void * data, uint32_t num)
```

Location: Driver_SPI.c:80

Start receiving data from SPI receiver.

Returns:

execution_status

Parameters

Direction	Name	Description
`out`	*data*	Pointer to buffer for data to receive from SPI receiver
`in`	*num*	Number of data items to receive

20.9.5.8 ARM_SPI_Transfer

```
int32_t ARM_SPI_Transfer(const void * data_out, void * data_in, uint32_t num)
```

Location: Driver_SPI.c:84

Start sending/receiving data to/from SPI transmitter/receiver.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*data_out*	Pointer to buffer with data to send to SPI transmitter
`out`	*data_in*	Pointer to buffer for data to receive from SPI receiver
`in`	*num*	Number of data items to transfer

20.9.5.9 ARM_SPI_GetDataCount

```
uint32_t ARM_SPI_GetDataCount()
```

Location: Driver_SPI.c:88

Get transferred data count.

Returns:

number of data items transferred

20.9.5.10 ARM_SPI_Control

```
int32_t ARM_SPI_Control(uint32_t control, uint32_t arg)
```

Location: Driver_SPI.c:92

Control SPI Interface.

Returns:

common execution_status and driver specific spi_execution_status

Parameters

Direction	Name	Description
`in`	*control*	Operation
`in`	*arg*	Argument of operation (optional)

20.9.5.11 ARM_SPI_GetStatus

```
ARM_SPI_STATUS ARM_SPI_GetStatus()
```

Location: Driver_SPI.c:125

Get SPI status.

Returns:

SPI status ARM_SPI_STATUS

20.9.5.12 ARM_SPI_SignalEvent

```
void ARM_SPI_SignalEvent(uint32_t event)
```

Location: Driver_SPI.c:129

Signal SPI Events.

Returns:

none

Parameters

Direction	Name	Description
`in`	*event*	SPI_events notification mask

20.10 CMSIS TIMER DRIVER

CMSIS Timer Driver Reference.

20.10.1 Summary

Typedefs

- TIMER_SEL_t : Timer selection.
- TIMER_MODE_t : Timer mode selection.
- TIMER_CLKSRC_t : Timer clock source selection.

- TIMER_PRESCALE_t : Timer prescale values selection.
- TIMER_MULTI_COUNT_t : Timer multi-count values selection.
- TIMER_GPIO_STATUS_t : Timer GPIO status.
- TIMER_GPIO_INT_MODE_t : Timer GPIO capture mode.
- TIMER_GPIO_t : Timer GPIO interrupt selection.
- TIMER_SYSTICK_CLKSRC_t : Timer SysTick Clock sources.
- ADC_EVENT_t : Timer interrupt events selection.
- TIMER_SignalEvent_t : Pointer to TIMER_SignalEvent : Signal Timer event.
- TIMER_t : Timer Driver configuration.
- SYSTICK_t : SysTick Driver configuration.
- TIMER_CFG_t : Common TIMER driver configuration.
- TIMER_PRI_CFG_t : Timer interrupt priority configuration.
- DRIVER_TIMER_t : Access structure of the TIMER Driver.

Data Structures

- _TIMER_t : Timer Driver configuration.
- _SYSTICK_t : SysTick Driver configuration.
- _TIMER_PRI_CFG_t : Timer interrupt priority configuration.
- _DRIVER_TIMER_t : Access structure of the TIMER Driver.

Enumerations

- _TIMER_SEL_t : Timer selection.
- _TIMER_MODE_t : Timer mode selection.
- _TIMER_CLKSRC_t : Timer clock source selection.
- _TIMER_PRESCALE_t : Timer prescale values selection.
- _TIMER_MULTI_COUNT_t : Timer multi-count values selection.
- _TIMER_GPIO_STATUS_t : Timer GPIO status.
- _TIMER_GPIO_INT_MODE_t : Timer GPIO capture mode.
- _TIMER_GPIO_t : Timer GPIO interrupt selection.
- _TIMER_SYSTICK_CLKSRC_t : Timer SysTick Clock sources.
- _ADC_EVENT_t : Timer interrupt events selection.

Macros

- ARM_TIMER_API_VERSION : Timer API version.
- TIMER_ERROR_UNCONFIGURED : Driver has not been configured yet.

Functions

- TIMER_GetVersion : Get driver version.
- TIMER_Initialize : Initialize Timer driver with default configuration.
- TIMER_Configure : Configure particular Timer.
- TIMER_SetInterruptPriority : Configure the Timer interrupt priority.

- TIMER_Start : Starts the Timer.
- TIMER_Stop : Stops the Timer.
- TIMER_SetValue : Sets the timeout / reload value of the selected Timer.
- TIMER_GetValue : Returns the current value of Timer.
- TIMER_GetValueCapture : Returns the current value of Timer.
- TIMER_GetSysTickState : Returns 1 if SysTick has already reached 0.
- TIMER_SignalEvent : Signal Timer events.
- TIMER_SetGPIOInterrupt : Set GPIO interrupt capture status.

20.10.2 CMSIS Timer Driver Typedef Documentation

20.10.2.1 TIMER_SEL_t

```
typedef enum _TIMER_SEL_t TIMER_SEL_t
```

Location: Driver_TIMER.h:49

Timer selection.

20.10.2.2 TIMER_MODE_t

```
typedef enum _TIMER_MODE_t TIMER_MODE_t
```

Location: Driver_TIMER.h:57

Timer mode selection.

20.10.2.3 TIMER_CLKSRC_t

```
typedef enum _TIMER_CLKSRC_t TIMER_CLKSRC_t
```

Location: Driver_TIMER.h:65

Timer clock source selection.

20.10.2.4 TIMER_PRESCALE_t

```
typedef enum _TIMER_PRESCALE_t TIMER_PRESCALE_t
```

Location: Driver_TIMER.h:79

Timer prescale values selection.

20.10.2.5 TIMER_MULTI_COUNT_t

```
typedef enum _TIMER_MULTI_COUNT_t TIMER_MULTI_COUNT_t
```

Location: Driver_TIMER.h:93

Timer multi-count values selection.

20.10.2.6 TIMER_GPIO_STATUS_t

```
typedef enum _TIMER_GPIO_STATUS_t TIMER_GPIO_STATUS_t
```

Location: Driver_TIMER.h:101

Timer GPIO status.

20.10.2.7 TIMER_GPIO_INT_MODE_t

```
typedef enum _TIMER_GPIO_INT_MODE_t TIMER_GPIO_INT_MODE_t
```

Location: Driver_TIMER.h:109

Timer GPIO capture mode.

20.10.2.8 TIMER_GPIO_t

```
typedef enum _TIMER_GPIO_t TIMER_GPIO_t
```

Location: Driver_TIMER.h:119

Timer GPIO interrupt selection.

20.10.2.9 TIMER_SYSTICK_CLKSRC_t

```
typedef enum _TIMER_SYSTICK_CLKSRC_t TIMER_SYSTICK_CLKSRC_t
```

Location: Driver_TIMER.h:127

Timer SysTick Clock sources.

20.10.2.10 ADC_EVENT_t

```
typedef enum _ADC_EVENT_t ADC_EVENT_t
```

Location: Driver_TIMER.h:138

Timer interrupt events selection.

20.10.2.11 TIMER_SignalEvent_t

```
typedef void(* TIMER_SignalEvent_t
```

Location: Driver_TIMER.h:207

Pointer to TIMER_SignalEvent : Signal Timer event.

20.10.2.12 TIMER_t

```
typedef struct _TIMER_t TIMER_t
```

Location: Driver_TIMER.h:223

Timer Driver configuration.

20.10.2.13 SYSTICK_t

```
typedef struct _SYSTICK_t SYSTICK_t
```

Location: Driver_TIMER.h:233

SysTick Driver configuration.

20.10.2.14 TIMER_CFG_t

```
typedef union _TIMER_CFG_t TIMER_CFG_t
```

Location: Driver_TIMER.h:242

Common TIMER driver configuration.

20.10.2.15 TIMER_PRI_CFG_t

```
typedef struct _TIMER_PRI_CFG_t TIMER_PRI_CFG_t
```

Location: Driver_TIMER.h:253

Timer interrupt priority configuration.

20.10.2.16 DRIVER_TIMER_t

```
typedef struct _DRIVER_TIMER_t DRIVER_TIMER_t
```

Location: Driver_TIMER.h:270

Access structure of the TIMER Driver.

20.10.3 CMSIS Timer Driver Data Structures Type Documentation

20.10.3.1 _TIMER_t

Location: Driver_TIMER.h:212

Timer Driver configuration.

Data Fields

`Type`	*Name*	Description
`TIMER_MODE_t`	*mode*	Timer mode to be used.
`TIMER_CLKSRC_t`	*clk_src*	Clock source to be used.

TIMER_GPIO_INT_MODE_t	*gpio_mode*	GPIO capture mode to be used.
uint32_t	*__pad0__*	Reserved.
TIMER_PRESCALE_t	*prescale_val*	Timer prescale value.
TIMER_MULTI_COUNT_t	*multi_cnt*	Multi count value.
uint32_t	*__pad1__*	Reserved.
TIMER_GPIO_t	*gpio_int*	GPIO value.
uint32_t	*timeout_val*	Timer timeout value.

20.10.3.2 _SYSTICK_t

Location: Driver_TIMER.h:228

SysTick Driver configuration.

Data Fields

Type	*Name*	Description
TIMER_SYSTICK_CLKSRC_t	*clk_src*	Clock source to be used.
uint32_t	*__pad0__*	Reserved.
uint32_t	*reload_val*	SysTick value.

20.10.3.3 _TIMER_PRI_CFG_t

Location: Driver_TIMER.h:247

Timer interrupt priority configuration.

Data Fields

Type	Name	Description
uint32_t	*preempt_pri*	Preempt priority.
uint32_t	*__pad0__*	Reserved.
uint32_t	*subgrp_pri*	Subgroup priority.
uint32_t	*__pad1__*	Reserved.

20.10.3.4 _DRIVER_TIMER_t

Location: Driver_TIMER.h:258

Access structure of the TIMER Driver.

Data Fields

Type	Name	Description
ARM_DRIVER_VERSION(*	*GetVersion)(void)*	Pointer to TIMER_GetVersion : Get driver version.
int32_t(*	*Initialize)(TIMER_SignalEvent_t cb)*	Pointer to TIMER_Initialize : Initialize Timer driver.
int32_t(*	*Configure)(TIMER_SEL_t sel, const TIMER_CFG_t *cfg)*	Pointer to TIMER_Configure : Configure driver.
int32_t(*	*SetInterruptPriority)(TIMER_SEL_t sel, const TIMER_PRI_CFG_t *pri)*	Pointer to TIMER_SetInterruptPriority : Configure Timer interrupt priority.
int32_t(*	*Start)(TIMER_SEL_t sel)*	Pointer to TIMER_Start : Start particular Timer.
int32_t(*	*Stop)(TIMER_SEL_t sel)*	Pointer to TIMER_Stop : Stop particular Timer.
int32_t(*	*SetValue)(TIMER_SEL_t sel, uint32_t value)*	Pointer to TIMER_SetValue : Set the particular Timer value.
int32_t(*	*SetGPIOInterrupt)(TIMER_SEL_t sel)*	Pointer to TIMER_SetGPIOInterrupt : Set GPIO interrupt capture status.
uint32_t(*	*GetValue)(TIMER_*	Pointer to TIMER_GetValue : Get the particular Timer

	SEL_t sel)	value.
`uint32_t(*`	*GetValueCapture)(TIMER_SEL_t sel)*	Pointer to TIMER_GetValueCapture : Get the Timer GPIO Interrupt Captured Value.
`uint32_t(*`	*GetSysTickState)(void)*	Pointer to TIMER_GetSysTickState : Returns 1 if SysTick has already reached 0.

20.10.4 CMSIS Timer Driver Enumeration Type Documentation

20.10.4.1 _TIMER_SEL_t

Location: Driver_TIMER.h:43

Timer selection.

Members

- `TIMER_0 = 0`

  Timer module 0.

- `TIMER_1 = 1`

  Timer module 1.

- `TIMER_2 = 2`

  Timer module 2.

- `TIMER_3 = 3`

  Timer module 3.

- `TIMER_SYSTICK = 4`

  Timer module SysTick.

20.10.4.2 _TIMER_MODE_t

Location: Driver_TIMER.h:54

Timer mode selection.

Members

- `TIMER_MODE_SHOT = 0x0U`

  Timer mode = TIMER_SHOT_MODE_BITBAND.

- `TIMER_MODE_FREE_RUN = 0x1U`

  Timer mode = TIMER_FREE_RUN_BITBAND.

20.10.4.3 _TIMER_CLKSRC_t

Location: Driver_TIMER.h:62

Timer clock source selection.

Members

- `TIMER_SLOWCLOCK_DIV32 = 0x0U`

  Timer src = SLOWCLOCK DIV32.

- `TIMER_SLOWCLOCK_DIV2 = 0x1U`

  Timer src = SLOWCLOCK DIV2.

20.10.4.4 _TIMER_PRESCALE_t

Location: Driver_TIMER.h:70

Timer prescale values selection.

Members

- `TIMER_PRESCALE_VAL_1 = 0x0U`

  Timer prescale = 1.

- `TIMER_PRESCALE_VAL_2 = 0x1U`

  Timer prescale = 2.

- `TIMER_PRESCALE_VAL_4 = 0x2U`

  Timer prescale = 4.

- `TIMER_PRESCALE_VAL_8 = 0x3U`

  Timer prescale = 8.

- `TIMER_PRESCALE_VAL_16 = 0x4U`

  Timer prescale = 16.

- `TIMER_PRESCALE_VAL_32 = 0x5U`

  Timer prescale = 32.

- `TIMER_PRESCALE_VAL_64 = 0x6U`

Timer prescale = 64.

- `TIMER_PRESCALE_VAL_128 = 0x7U`

  Timer prescale = 128.

20.10.4.5 _TIMER_MULTI_COUNT_t

Location: Driver_TIMER.h:84

Timer multi-count values selection.

Members

- `TIMER_MULTI_COUNT_VAL_1 = 0x0U`

  Timer multiCount = 1.
- `TIMER_MULTI_COUNT_VAL_2 = 0x1U`

  Timer multiCount = 2.
- `TIMER_MULTI_COUNT_VAL_3 = 0x2U`

  Timer multiCount = 3.
- `TIMER_MULTI_COUNT_VAL_4 = 0x3U`

  Timer multiCount = 4.

- `TIMER_MULTI_COUNT_VAL_5 = 0x4U`

  Timer multiCount = 5.

- `TIMER_MULTI_COUNT_VAL_6 = 0x5U`

  Timer multiCount = 6.

- `TIMER_MULTI_COUNT_VAL_7 = 0x6U`

  Timer multiCount = 7.

- `TIMER_MULTI_COUNT_VAL_8 = 0x7U`

  Timer multiCount = 8.

20.10.4.6 _TIMER_GPIO_STATUS_t

Location: Driver_TIMER.h:98

Timer GPIO status.

Members

- `TIMER_GPIO_INT_DISABLE_STATUS = 0x0U`

  Timer GPIO status = disable.

- `TIMER_GPIO_INT_ENABLE_STATUS = 0x1U`

  Timer GPIO status = enable.

20.10.4.7 _TIMER_GPIO_INT_MODE_t

Location: Driver_TIMER.h:106

Timer GPIO capture mode.

Members

- `TIMER_GPIO_SINGLE_MODE = 0x0U`

  Timer capture mode = single.

- `TIMER_GPIO_CONTINUOUS_MODE = 0x1U`

  Timer capture mode = continuous.

20.10.4.8 _TIMER_GPIO_t

Location: Driver_TIMER.h:114

Timer GPIO interrupt selection.

Members

- `TIMER_GPIO_0 = 0x0U`

Timer GPIO interrupt 0.

- `TIMER_GPIO_1 = 0x1U`

  Timer GPIO interrupt 1.

- `TIMER_GPIO_2 = 0x2U`

  Timer GPIO interrupt 2.

- `TIMER_GPIO_3 = 0x3U`

  Timer GPIO interrupt 3.

20.10.4.9 _TIMER_SYSTICK_CLKSRC_t

Location: Driver_TIMER.h:124

Timer SysTick Clock sources.

Members

- `SYSTICK_CLKSOURCE_EXTREFCLK = 0x0U`

  SysTick Timer CLK src = external ref.

- `SYSTICK_CLKSOURCE_CORECLK = 0x1U`

  SysTick Timer CLK src = core CLK.

20.10.4.10 _ADC_EVENT_t

Location: Driver_TIMER.h:132

Timer interrupt events selection.

Members

- `TIMER_TIMER0_EVENT = 1 << TIMER_0`

  Timer0 event.

- `TIMER_TIMER1_EVENT = 1 << TIMER_1`

  Timer1 event.

- `TIMER_TIMER2_EVENT = 1 << TIMER_2`

  Timer2 event.

- `TIMER_TIMER3_EVENT = 1 << TIMER_3`

  Timer3 event.

- `TIMER_SYSTICK_EVENT = 1 << TIMER_SYSTICK`

  SysTick event.

20.10.5 CMSIS Timer Driver Macro Definition Documentation

20.10.5.1 ARM_TIMER_API_VERSION

```
#define ARM_TIMER_API_VERSION ARM_DRIVER_VERSION_MAJOR_MINOR(1,0)
```

Location: Driver_TIMER.h:36

Timer API version.

20.10.5.2 TIMER_ERROR_UNCONFIGURED

```
#define TIMER_ERROR_UNCONFIGURED (ARM_DRIVER_ERROR_SPECIFIC - 1)
```

Location: Driver_TIMER.h:141

Driver has not been configured yet.

20.10.6 CMSIS Timer Driver Function Documentation

20.10.6.1 TIMER_GetVersion

```
ARM_DRIVER_VERSION TIMER_GetVersion()
```

Location: Driver_Timer.c:21

Get driver version.

Returns:

ARM_DRIVER_VERSION

20.10.6.2 TIMER_Initialize

```
int32_t TIMER_Initialize(TIMER_SignalEvent_t cb)
```

Location: Driver_Timer.c:22

Initialize Timer driver with default configuration.

Returns:

execution_status

Parameters

Direction	Name	Description
in	*cb*	Pointer to TIMER_SignalEvent

20.10.6.3 TIMER_Configure

```
int32_t TIMER_Configure(TIMER_SEL_t sel, const TIMER_CFG_t * cfg)
```

Location: Driver_Timer.c:23

Configure particular Timer.

Returns:

execution_status

Parameters

Direction	Name	Description
in	*sel*	Timer to be configured (TIMER_SEL_t)
in	*cfg*	Pointer to TIMER_CFG_t

20.10.6.4 TIMER_SetInterruptPriority

```
int32_t TIMER_SetInterruptPriority(TIMER_SEL_t sel, const TIMER_PRI_CFG_t * cfg)
```

Location: Driver_Timer.c:24

Configure the Timer interrupt priority.

Returns:

execution_status

Parameters

Direction	Name	Description
in	*sel*	Timer to be configured (TIMER_SEL_t)
in	*cfg*	Pointer to TIMER_PRI_CFG_t

20.10.6.5 TIMER_Start

```
int32_t TIMER_Start(TIMER_SEL_t sel)
```

Location: Driver_Timer.c:25

Starts the Timer.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*sel*	Timer number to be started (TIMER_SEL_t)

20.10.6.6 TIMER_Stop

```
int32_t TIMER_Stop(TIMER_SEL_t sel)
```

Location: Driver_Timer.c:26

Stops the Timer.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*sel*	Timer number to be stopped (TIMER_SEL_t)

20.10.6.7 TIMER_SetValue

```
int32_t TIMER_SetValue(TIMER_SEL_t sel, uint32_t val)
```

Location: Driver_Timer.c:27

Sets the timeout / reload value of the selected Timer.

Returns:

execution_status of error status

Parameters

Direction	Name	Description
`in`	*sel*	Timer value to be read (TIMER_SEL_t)
`in`	*val*	Timer value to be set

20.10.6.8 TIMER_GetValue

```
uint32_t TIMER_GetValue(TIMER_SEL_t sel)
```

Location: Driver_Timer.c:29

Returns the current value of Timer.

Returns:

Timer value or 0 if Timer was not enabled

Parameters

Direction	Name	Description
`in`	*sel*	Timer value to be read (TIMER_SEL_t)

20.10.6.9 TIMER_GetValueCapture

```
uint32_t TIMER_GetValueCapture(TIMER_SEL_t sel)
```

Location: Driver_Timer.c:30

Returns the current value of Timer.

Returns:

Timer capture value or 0 if Timer was not enabled

Parameters

Direction	Name	Description
`in`	*sel*	Timer value to be read (TIMER_SEL_t)

20.10.6.10 TIMER_GetSysTickState

```
uint32_t TIMER_GetSysTickState()
```

Location: Driver_Timer.c:31

Returns 1 if SysTick has already reached 0.

Returns:

SysTick status or 0 if SysTick was not enabled

20.10.6.11 TIMER_SignalEvent

```
void TIMER_SignalEvent(uint32_t event)
```

Location: Driver_Timer.c:33

Signal Timer events.

Returns:

none

Parameters

Direction	Name	Description
in	*event*	Notification mask

20.10.6.12 TIMER_SetGPIOInterrupt

```
int32_t TIMER_SetGPIOInterrupt(TIMER_SEL_t sel)
```

Location: Driver_Timer.c:28

Set GPIO interrupt capture status.

Returns:

execution_status

Parameters

Direction	Name	Description
in	*sel*	Timer value to be read (TIMER_SEL_t)

20.11 CMSIS USART DRIVER

CMSIS USART Driver Reference.

20.11.1 Summary

Typedefs

- ARM_USART_STATUS : USART Status.
- ARM_USART_MODEM_CONTROL : USART Modem Control.
- ARM_USART_MODEM_STATUS : USART Modem Status.
- ARM_USART_SignalEvent_t : Pointer to ARM_USART_SignalEvent : Signal USART Event.
- ARM_USART_CAPABILITIES : USART Device Driver Capabilities.
- ARM_DRIVER_USART : Access structure of the USART Driver.

Data Structures

- _ARM_USART_STATUS : USART Status.
- _ARM_USART_MODEM_STATUS : USART Modem Status.
- _ARM_USART_CAPABILITIES : USART Device Driver Capabilities.
- _ARM_DRIVER_USART : Access structure of the USART Driver.

Enumerations

- _ARM_USART_MODEM_CONTROL : USART Modem Control.

Macros

- ARM_USART_API_VERSION : API version.
- ARM_USART_CONTROL_Pos : Position of the 0th bit of the USART control field in the ARM_USART structure.
- ARM_USART_CONTROL_Msk : Positioning of USART control field in the ARM_USART structure.
- ARM_USART_MODE_ASYNCHRONOUS : UART (Asynchronous); arg = Baudrate.
- ARM_USART_MODE_SYNCHRONOUS_MASTER : Synchronous Master (generates clock signal); arg = Baudrate.
- ARM_USART_MODE_SYNCHRONOUS_SLAVE : Synchronous Slave (external clock signal).
- ARM_USART_MODE_SINGLE_WIRE : UART Single-wire (half-duplex); arg = Baudrate.
- ARM_USART_MODE_IRDA : UART IrDA; arg = Baudrate.
- ARM_USART_MODE_SMART_CARD : UART Smart Card; arg = Baudrate.
- ARM_USART_DATA_BITS_Pos : Position of the 0th bit of the Data bits field in the ARM_USART structure.
- ARM_USART_DATA_BITS_Msk : Positioning of the Data bits field in the ARM_USART structure.
- ARM_USART_DATA_BITS_5 : 5 data bits.
- ARM_USART_DATA_BITS_6 : 6 data bits.
- ARM_USART_DATA_BITS_7 : 7 data bits.
- ARM_USART_DATA_BITS_8 : 8 data bits (default).
- ARM_USART_DATA_BITS_9 : 9 data bits.
- ARM_USART_PARITY_Pos : Position of the 0th bit of the Mode parameters Parity field in the ARM_USART structure.
- ARM_USART_PARITY_Msk : Positioning of the Mode parameters Parity field in the ARM_USART structure.
- ARM_USART_PARITY_NONE : No parity (default).
- ARM_USART_PARITY_EVEN : Even parity.
- ARM_USART_PARITY_ODD : Odd parity.
- ARM_USART_STOP_BITS_Pos : Position of the 0th bit of the Mode parameters Stop bits field in the ARM_USART structure.
- ARM_USART_STOP_BITS_Msk : Positioning of the Mode parameters Stop bits field in the ARM_USART structure.
- ARM_USART_STOP_BITS_1 : 1 stop bit (default).
- ARM_USART_STOP_BITS_2 : 2 stop bits.
- ARM_USART_STOP_BITS_1_5 : 1.5 stop bits.
- ARM_USART_STOP_BITS_0_5 : 0.5 stop bits.
- ARM_USART_FLOW_CONTROL_Pos : Position of the 0th bit of the Mode parameters Flow control field in the ARM_USART structure.
- ARM_USART_FLOW_CONTROL_Msk : Positioning of the Mode parameters Flow control field in the ARM_USART structure.

- ARM_USART_FLOW_CONTROL_NONE : No flow control (default).
- ARM_USART_FLOW_CONTROL_RTS : RTS flow control.
- ARM_USART_FLOW_CONTROL_CTS : CTS flow control.
- ARM_USART_FLOW_CONTROL_RTS_CTS : RTS/CTS flow control.
- ARM_USART_CPOL_Pos : Position of the 0th bit of the Mode parameters Clock polarity field in the ARM_USART structure.
- ARM_USART_CPOL_Msk : Positioning of the Mode parameters Clock polarity field in the ARM_USART structure.
- ARM_USART_CPOL0 : CPOL = 0 (default).
- ARM_USART_CPOL1 : CPOL = 1.
- ARM_USART_CPHA_Pos : Position of the 0th bit of the Mode parameters Clock phase field in the ARM_USART structure.
- ARM_USART_CPHA_Msk : Positioning of the Mode parameters Clock phase field in the ARM_USART structure.
- ARM_USART_CPHA0 : CPHA = 0 (default).
- ARM_USART_CPHA1 : CPHA = 1.
- ARM_USART_SET_DEFAULT_TX_VALUE : Set default transmit value (synchronous receive only); arg = value.
- ARM_USART_SET_IRDA_PULSE : Set IrDA Pulse in ns; arg: 0=3/16 of bit period.
- ARM_USART_SET_SMART_CARD_GUARD_TIME : Set smart card guard time; arg = number of bit periods.
- ARM_USART_SET_SMART_CARD_CLOCK : Set smart card clock in Hz; arg: 0=Clock not generated.
- ARM_USART_CONTROL_SMART_CARD_NACK : Smart card NACK generation; arg: 0=disabled, 1=enabled.
- ARM_USART_CONTROL_TX : Transmitter; arg: 0=disabled, 1=enabled.
- ARM_USART_CONTROL_RX : Receiver; arg: 0=disabled, 1=enabled.
- ARM_USART_CONTROL_BREAK : Continuous break transmission; arg: 0=disabled, 1=enabled.
- ARM_USART_ABORT_SEND : Abort ARM_USART_Send.
- ARM_USART_ABORT_RECEIVE : Abort ARM_USART_Receive.
- ARM_USART_ABORT_TRANSFER : Abort ARM_USART_Transfer.
- ARM_USART_ERROR_MODE : Specified mode not supported.
- ARM_USART_ERROR_BAUDRATE : Specified baudrate not supported.
- ARM_USART_ERROR_DATA_BITS : Specified number of data bits not supported.
- ARM_USART_ERROR_PARITY : Specified parity not supported.
- ARM_USART_ERROR_STOP_BITS : Specified number of stop bits not supported.
- ARM_USART_ERROR_FLOW_CONTROL : Specified flow control not supported.
- ARM_USART_ERROR_CPOL : Specified clock polarity not supported.
- ARM_USART_ERROR_CPHA : Specified clock phase not supported.
- ARM_USART_EVENT_SEND_COMPLETE : Send completed; however USART may still transmit data.
- ARM_USART_EVENT_RECEIVE_COMPLETE : Receive completed.
- ARM_USART_EVENT_TRANSFER_COMPLETE : Transfer completed.
- ARM_USART_EVENT_TX_COMPLETE : Transmit completed (optional).
- ARM_USART_EVENT_TX_UNDERFLOW : Transmit data not available (synchronous slave).
- ARM_USART_EVENT_RX_OVERFLOW : Receive data overflow.
- ARM_USART_EVENT_RX_TIMEOUT : Receive character timeout (optional).
- ARM_USART_EVENT_RX_BREAK : Break detected on receive.
- ARM_USART_EVENT_RX_FRAMING_ERROR : Framing error detected on receive.
- ARM_USART_EVENT_RX_PARITY_ERROR : Parity error detected on receive.
- ARM_USART_EVENT_CTS : CTS state changed (optional).

- ARM_USART_EVENT_DSR : DSR state changed (optional).
- ARM_USART_EVENT_DCD : DCD state changed (optional).
- ARM_USART_EVENT_RI : RI state changed (optional).

Functions

- ARM_USART_GetVersion : Get driver version.
- ARM_USART_GetCapabilities : Get driver capabilities.
- ARM_USART_Initialize : Initialize USART Interface.
- ARM_USART_Uninitialize : De-initialize USART Interface.
- ARM_USART_PowerControl : Control USART Interface Power.
- ARM_USART_Send : Start sending data to USART transmitter.
- ARM_USART_Receive : Start receiving data from USART receiver.
- ARM_USART_Transfer : Start sending/receiving data to/from USART transmitter/receiver.
- ARM_USART_GetTxCount : Get transmitted data count.
- ARM_USART_GetRxCount : Get received data count.
- ARM_USART_Control : Control USART Interface.
- ARM_USART_GetStatus : Get USART status.
- ARM_USART_SetModemControl : Set USART Modem Control line state.
- ARM_USART_GetModemStatus : Get USART Modem Status lines state.
- ARM_USART_SignalEvent : Signal USART Events.

20.11.2 CMSIS USART Driver Typedef Documentation

20.11.2.1 ARM_USART_STATUS

```
typedef struct _ARM_USART_STATUS ARM_USART_STATUS
```

Location: Driver_USART.h:176

USART Status.

20.11.2.2 ARM_USART_MODEM_CONTROL

```
typedef enum _ARM_USART_MODEM_CONTROL ARM_USART_MODEM_CONTROL
```

Location: Driver_USART.h:186

USART Modem Control.

20.11.2.3 ARM_USART_MODEM_STATUS

```
typedef struct _ARM_USART_MODEM_STATUS ARM_USART_MODEM_STATUS
```

Location: Driver_USART.h:197

USART Modem Status.

20.11.2.4 ARM_USART_SignalEvent_t

```
typedef void(* ARM_USART_SignalEvent_t
```

Location: Driver_USART.h:295

Pointer to ARM_USART_SignalEvent : Signal USART Event.

20.11.2.5 ARM_USART_CAPABILITIES

```
typedef struct _ARM_USART_CAPABILITIES ARM_USART_CAPABILITIES
```

Location: Driver_USART.h:324

USART Device Driver Capabilities.

20.11.2.6 ARM_DRIVER_USART

```
typedef struct _ARM_DRIVER_USART ARM_DRIVER_USART
```

Location: Driver_USART.h:347

Access structure of the USART Driver.

20.11.3 CMSIS USART Driver Data Structures Type Documentation

20.11.3.1 _ARM_USART_STATUS

Location: Driver_USART.h:167

USART Status.

Data Fields

Type	*Name*	Description
`uint32_t`	*tx_busy*	Transmitter busy flag.
`uint32_t`	*rx_busy*	Receiver busy flag.
`uint32_t`	*tx_underflow*	Transmit data underflow detected (cleared on start of next send operation).
`uint32_t`	*rx_overflow*	Receive data overflow detected (cleared on start of next receive operation).
`uint32_t`	*rx_break*	Break detected on receive (cleared on start of next receive operation).
`uint32_t`	*rx_framing_error*	Framing error detected on receive (cleared on start of next receive operation).
`uint32_t`	*rx_parity_error*	Parity error detected on receive (cleared on start of next receive operation).
`uint32_t`	*reserved*	(Reserved for future use)

20.11.3.2 _ARM_USART_MODEM_STATUS

Location: Driver_USART.h:191

USART Modem Status.

Data Fields

Type	*Name*	Description
`uint32_t`	*cts*	CTS state: 1=Active, 0=Inactive.
`uint32_t`	*dsr*	DSR state: 1=Active, 0=Inactive.
`uint32_t`	*dcd*	DCD state: 1=Active, 0=Inactive.
`uint32_t`	*ri*	RI state: 1=Active, 0=Inactive.
`uint32_t`	*reserved*	(Reserved for future use)

20.11.3.3 _ARM_USART_CAPABILITIES

Location: Driver_USART.h:301

USART Device Driver Capabilities.

Data Fields

`Type`	*Name*	Description
`uint32_t`	*asynchronous*	supports UART (asynchronous) mode.
`uint32_t`	*synchronous_master*	supports synchronous master mode.
`uint32_t`	*synchronous_slave*	supports synchronous slave mode.
`uint32_t`	*single_wire*	supports UART single-wire mode.
`uint32_t`	*irda*	supports UART IrDA mode.
`uint32_t`	*smart_card*	supports UART smart card mode.
`uint32_t`	*smart_card_clock*	Smart card clock generator available.
`uint32_t`	*flow_control_rts*	RTS flow control available.
`uint32_t`	*flow_control_cts*	CTS flow control available.
`uint32_t`	*event_tx_complete*	Transmit completed event: ARM_USART_EVENT_TX_COMPLETE.
`uint32_t`	*event_rx_timeout*	Signal receive character timeout event: ARM_USART_EVENT_RX_TIMEOUT.
`uint32_t`	*rts*	RTS Line: 0=not available, 1=available.
`uint32_t`	*cts*	CTS Line: 0=not available, 1=available.
`uint32_t`	*dtr*	DTR Line: 0=not available, 1=available.
`uint32_t`	*dsr*	DSR Line: 0=not available, 1=available.
`uint32_t`	*dcd*	DCD Line: 0=not available, 1=available.
`uint32_t`	*ri*	RI Line: 0=not available, 1=available.
`uint32_t`	*event_cts*	Signal CTS change event: ARM_USART_EVENT_CTS.
`uint32_t`	*event_dsr*	Signal DSR change event: ARM_USART_EVENT_DSR.
`uint32_t`	*event_dcd*	Signal DCD change event: ARM_USART_EVENT_DCD.
`uint32_t`	*event_ri*	Signal RI change event: ARM_USART_EVENT_RI.
`uint32_t`	*reserved*	Reserved (must be zero).

20.11.3.4 _ARM_DRIVER_USART

Location: Driver_USART.h:330

Access structure of the USART Driver.

Data Fields

Type	Name	Description
ARM_DRIVER_VERSION(*	*GetVersion)(void)*	Pointer to ARM_USART_GetVersion : Get driver version.
ARM_USART_CAPABILITIES(*	*GetCapabilities)(void)*	Pointer to ARM_USART_GetCapabilities : Get driver capabilities.
int32_t(*	*Initialize)(ARM_USART_SignalEvent_t cb_event)*	Pointer to ARM_USART_Initialize : Initialize USART Interface.
int32_t(*	*Uninitialize)(void)*	Pointer to ARM_USART_Uninitialize : De-initialize USART Interface.
int32_t(*	*PowerControl)(ARM_POWER_STATE state)*	Pointer to ARM_USART_PowerControl : Control USART Interface Power.
int32_t(*	*Send)(const void *data, uint32_t num)*	Pointer to ARM_USART_Send : Start sending data to USART transmitter.
int32_t(*	*Receive)(void *data, uint32_t num)*	Pointer to ARM_USART_Receive : Start receiving data from USART receiver.
int32_t(*	*Transfer)(const void *data_out, void *data_in, uint32_t num)*	Pointer to ARM_USART_Transfer : Start sending/receiving data to/from USART.
uint32_t(*	*GetTxCount)(void)*	Pointer to ARM_USART_GetTxCount : Get transmitted data count.
uint32_t(*	*GetRxCount)(void)*	Pointer to ARM_USART_GetRxCount : Get received data count.
int32_t(*	*Control)(uint32_t control, uint32_t arg)*	Pointer to ARM_USART_Control : Control USART Interface.

ARM_USART_STATUS (*	*GetStatus)(void)*	Pointer to ARM_USART_GetStatus : Get USART status.
int32_t(*	*SetModemControl)(ARM_USART_MODEM_CONTROL control)*	Pointer to ARM_USART_SetModemControl : Set USART modem control line state.
ARM_USART_MODEM_STATUS(*	*GetModemStatus)(void)*	Pointer to ARM_USART_GetModemStatus : Get USART modem status lines state.

20.11.4 CMSIS USART Driver Enumeration Type Documentation

20.11.4.1 _ARM_USART_MODEM_CONTROL

Location: Driver_USART.h:181

USART Modem Control.

Members

- `ARM_USART_RTS_CLEAR`

  Deactivate RTS.

- `ARM_USART_RTS_SET`

  Activate RTS.

- `ARM_USART_DTR_CLEAR`

  Deactivate DTR.

- `ARM_USART_DTR_SET`

  Activate DTR.

20.11.5 CMSIS USART Driver Macro Definition Documentation

20.11.5.1 ARM_USART_API_VERSION

```
#define ARM_USART_API_VERSION ARM_DRIVER_VERSION_MAJOR_MINOR(2,3)
```

Location: Driver_USART.h:78

API version.

20.11.5.2 ARM_USART_CONTROL_Pos

```
#define ARM_USART_CONTROL_Pos 0
```

Location: Driver_USART.h:81

Position of the 0th bit of the USART control field in the ARM_USART structure.

20.11.5.3 ARM_USART_CONTROL_Msk

```
#define ARM_USART_CONTROL_Msk (0xFFUL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:82

Positioning of USART control field in the ARM_USART structure.

20.11.5.4 ARM_USART_MODE_ASYNCHRONOUS

```
#define ARM_USART_MODE_ASYNCHRONOUS (0x01UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:85

UART (Asynchronous); arg = Baudrate.

20.11.5.5 ARM_USART_MODE_SYNCHRONOUS_MASTER

```
#define ARM_USART_MODE_SYNCHRONOUS_MASTER (0x02UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:86

Synchronous Master (generates clock signal); arg = Baudrate.

20.11.5.6 ARM_USART_MODE_SYNCHRONOUS_SLAVE

```
#define ARM_USART_MODE_SYNCHRONOUS_SLAVE (0x03UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:87

Synchronous Slave (external clock signal).

20.11.5.7 ARM_USART_MODE_SINGLE_WIRE

```
#define ARM_USART_MODE_SINGLE_WIRE (0x04UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:88

UART Single-wire (half-duplex); arg = Baudrate.

20.11.5.8 ARM_USART_MODE_IRDA

```
#define ARM_USART_MODE_IRDA (0x05UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:89

UART IrDA; arg = Baudrate.

20.11.5.9 ARM_USART_MODE_SMART_CARD

```
#define ARM_USART_MODE_SMART_CARD (0x06UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:90

UART Smart Card; arg = Baudrate.

20.11.5.10 ARM_USART_DATA_BITS_Pos

```
#define ARM_USART_DATA_BITS_Pos 8
```

Location: Driver_USART.h:93

Position of the 0th bit of the Data bits field in the ARM_USART structure.

20.11.5.11 ARM_USART_DATA_BITS_Msk

```
#define ARM_USART_DATA_BITS_Msk (7UL << ARM_USART_DATA_BITS_Pos)
```

Location: Driver_USART.h:94

Positioning of the Data bits field in the ARM_USART structure.

20.11.5.12 ARM_USART_DATA_BITS_5

```
#define ARM_USART_DATA_BITS_5 (5UL << ARM_USART_DATA_BITS_Pos)
```

Location: Driver_USART.h:95

5 data bits.

20.11.5.13 ARM_USART_DATA_BITS_6

```
#define ARM_USART_DATA_BITS_6 (6UL << ARM_USART_DATA_BITS_Pos)
```

Location: Driver_USART.h:96

6 data bits.

20.11.5.14 ARM_USART_DATA_BITS_7

```
#define ARM_USART_DATA_BITS_7 (7UL << ARM_USART_DATA_BITS_Pos)
```

Location: Driver_USART.h:97

7 data bits.

20.11.5.15 ARM_USART_DATA_BITS_8

```
#define ARM_USART_DATA_BITS_8 (0UL << ARM_USART_DATA_BITS_Pos)
```

Location: Driver_USART.h:98

8 data bits (default).

20.11.5.16 ARM_USART_DATA_BITS_9

```
#define ARM_USART_DATA_BITS_9 (1UL << ARM_USART_DATA_BITS_Pos)
```

Location: Driver_USART.h:99

9 data bits.

20.11.5.17 ARM_USART_PARITY_Pos

```
#define ARM_USART_PARITY_Pos 12
```

Location: Driver_USART.h:102

Position of the 0th bit of the Mode parameters Parity field in the ARM_USART structure.

20.11.5.18 ARM_USART_PARITY_Msk

```
#define ARM_USART_PARITY_Msk (3UL << ARM_USART_PARITY_Pos)
```

Location: Driver_USART.h:103

Positioning of the Mode parameters Parity field in the ARM_USART structure.

20.11.5.19 ARM_USART_PARITY_NONE

```
#define ARM_USART_PARITY_NONE (0UL << ARM_USART_PARITY_Pos)
```

Location: Driver_USART.h:104

No parity (default).

20.11.5.20 ARM_USART_PARITY_EVEN

```
#define ARM_USART_PARITY_EVEN (1UL << ARM_USART_PARITY_Pos)
```

Location: Driver_USART.h:105

Even parity.

20.11.5.21 ARM_USART_PARITY_ODD

```
#define ARM_USART_PARITY_ODD (2UL << ARM_USART_PARITY_Pos)
```

Location: Driver_USART.h:106

Odd parity.

20.11.5.22 ARM_USART_STOP_BITS_Pos

```
#define ARM_USART_STOP_BITS_Pos 14
```

Location: Driver_USART.h:109

Position of the 0th bit of the Mode parameters Stop bits field in the ARM_USART structure.

20.11.5.23 ARM_USART_STOP_BITS_Msk

```
#define ARM_USART_STOP_BITS_Msk (3UL << ARM_USART_STOP_BITS_Pos)
```

Location: Driver_USART.h:110

Positioning of the Mode parameters Stop bits field in the ARM_USART structure.

20.11.5.24 ARM_USART_STOP_BITS_1

```
#define ARM_USART_STOP_BITS_1 (0UL << ARM_USART_STOP_BITS_Pos)
```

Location: Driver_USART.h:111

1 stop bit (default).

20.11.5.25 ARM_USART_STOP_BITS_2

```
#define ARM_USART_STOP_BITS_2 (1UL << ARM_USART_STOP_BITS_Pos)
```

Location: Driver_USART.h:112

2 stop bits.

20.11.5.26 ARM_USART_STOP_BITS_1_5

```
#define ARM_USART_STOP_BITS_1_5 (2UL << ARM_USART_STOP_BITS_Pos)
```

Location: Driver_USART.h:113

1.5 stop bits.

20.11.5.27 ARM_USART_STOP_BITS_0_5

```
#define ARM_USART_STOP_BITS_0_5 (3UL << ARM_USART_STOP_BITS_Pos)
```

Location: Driver_USART.h:114

0.5 stop bits.

20.11.5.28 ARM_USART_FLOW_CONTROL_Pos

```
#define ARM_USART_FLOW_CONTROL_Pos 16
```

Location: Driver_USART.h:117

Position of the 0th bit of the Mode parameters Flow control field in the ARM_USART structure.

20.11.5.29 ARM_USART_FLOW_CONTROL_Msk

```
#define ARM_USART_FLOW_CONTROL_Msk (3UL << ARM_USART_FLOW_CONTROL_Pos)
```

Location: Driver_USART.h:118

Positioning of the Mode parameters Flow control field in the ARM_USART structure.

20.11.5.30 ARM_USART_FLOW_CONTROL_NONE

```
#define ARM_USART_FLOW_CONTROL_NONE (0UL << ARM_USART_FLOW_CONTROL_Pos)
```

Location: Driver_USART.h:119

No flow control (default).

20.11.5.31 ARM_USART_FLOW_CONTROL_RTS

```
#define ARM_USART_FLOW_CONTROL_RTS (1UL << ARM_USART_FLOW_CONTROL_Pos)
```

Location: Driver_USART.h:120

RTS flow control.

20.11.5.32 ARM_USART_FLOW_CONTROL_CTS

```
#define ARM_USART_FLOW_CONTROL_CTS (2UL << ARM_USART_FLOW_CONTROL_Pos)
```

Location: Driver_USART.h:121

CTS flow control.

20.11.5.33 ARM_USART_FLOW_CONTROL_RTS_CTS

```
#define ARM_USART_FLOW_CONTROL_RTS_CTS (3UL << ARM_USART_FLOW_CONTROL_Pos)
```

Location: Driver_USART.h:122

RTS/CTS flow control.

20.11.5.34 ARM_USART_CPOL_Pos

```
#define ARM_USART_CPOL_Pos 18
```

Location: Driver_USART.h:125

Position of the 0th bit of the Mode parameters Clock polarity field in the ARM_USART structure.

20.11.5.35 ARM_USART_CPOL_Msk

```
#define ARM_USART_CPOL_Msk (1UL << ARM_USART_CPOL_Pos)
```

Location: Driver_USART.h:126

Positioning of the Mode parameters Clock polarity field in the ARM_USART structure.

20.11.5.36 ARM_USART_CPOL0

```
#define ARM_USART_CPOL0 (0UL << ARM_USART_CPOL_Pos)
```

Location: Driver_USART.h:127

CPOL = 0 (default).

20.11.5.37 ARM_USART_CPOL1

```
#define ARM_USART_CPOL1 (1UL << ARM_USART_CPOL_Pos)
```

Location: Driver_USART.h:128

CPOL = 1.

20.11.5.38 ARM_USART_CPHA_Pos

```
#define ARM_USART_CPHA_Pos 19
```

Location: Driver_USART.h:131

Position of the 0th bit of the Mode parameters Clock phase field in the ARM_USART structure.

20.11.5.39 ARM_USART_CPHA_Msk

```
#define ARM_USART_CPHA_Msk (1UL << ARM_USART_CPHA_Pos)
```

Location: Driver_USART.h:132

Positioning of the Mode parameters Clock phase field in the ARM_USART structure.

20.11.5.40 ARM_USART_CPHA0

```
#define ARM_USART_CPHA0 (0UL << ARM_USART_CPHA_Pos)
```

Location: Driver_USART.h:133

CPHA = 0 (default).

20.11.5.41 ARM_USART_CPHA1

```
#define ARM_USART_CPHA1 (1UL << ARM_USART_CPHA_Pos)
```

Location: Driver_USART.h:134

CPHA = 1.

20.11.5.42 ARM_USART_SET_DEFAULT_TX_VALUE

```
#define ARM_USART_SET_DEFAULT_TX_VALUE (0x10UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:138

Set default transmit value (synchronous receive only); arg = value.

20.11.5.43 ARM_USART_SET_IRDA_PULSE

```
#define ARM_USART_SET_IRDA_PULSE (0x11UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:139

Set IrDA Pulse in ns; arg: 0=3/16 of bit period.

20.11.5.44 ARM_USART_SET_SMART_CARD_GUARD_TIME

```
#define ARM_USART_SET_SMART_CARD_GUARD_TIME (0x12UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:140

Set smart card guard time; arg = number of bit periods.

20.11.5.45 ARM_USART_SET_SMART_CARD_CLOCK

```
#define ARM_USART_SET_SMART_CARD_CLOCK (0x13UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:141

Set smart card clock in Hz; arg: 0=Clock not generated.

20.11.5.46 ARM_USART_CONTROL_SMART_CARD_NACK

```
#define ARM_USART_CONTROL_SMART_CARD_NACK (0x14UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:142

Smart card NACK generation; arg: 0=disabled, 1=enabled.

20.11.5.47 ARM_USART_CONTROL_TX

```
#define ARM_USART_CONTROL_TX (0x15UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:143

Transmitter; arg: 0=disabled, 1=enabled.

20.11.5.48 ARM_USART_CONTROL_RX

```
#define ARM_USART_CONTROL_RX (0x16UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:144

Receiver; arg: 0=disabled, 1=enabled.

20.11.5.49 ARM_USART_CONTROL_BREAK

```
#define ARM_USART_CONTROL_BREAK (0x17UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:145

Continuous break transmission; arg: 0=disabled, 1=enabled.

20.11.5.50 ARM_USART_ABORT_SEND

```
#define ARM_USART_ABORT_SEND (0x18UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:146

Abort ARM_USART_Send.

20.11.5.51 ARM_USART_ABORT_RECEIVE

```
#define ARM_USART_ABORT_RECEIVE (0x19UL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:147

Abort ARM_USART_Receive.

20.11.5.52 ARM_USART_ABORT_TRANSFER

```
#define ARM_USART_ABORT_TRANSFER (0x1AUL << ARM_USART_CONTROL_Pos)
```

Location: Driver_USART.h:148

Abort ARM_USART_Transfer.

20.11.5.53 ARM_USART_ERROR_MODE

```
#define ARM_USART_ERROR_MODE (ARM_DRIVER_ERROR_SPECIFIC - 1)
```

Location: Driver_USART.h:153

Specified mode not supported.

20.11.5.54 ARM_USART_ERROR_BAUDRATE

```
#define ARM_USART_ERROR_BAUDRATE (ARM_DRIVER_ERROR_SPECIFIC - 2)
```

Location: Driver_USART.h:154

Specified baudrate not supported.

20.11.5.55 ARM_USART_ERROR_DATA_BITS

```
#define ARM_USART_ERROR_DATA_BITS (ARM_DRIVER_ERROR_SPECIFIC - 3)
```

Location: Driver_USART.h:155

Specified number of data bits not supported.

20.11.5.56 ARM_USART_ERROR_PARITY

```
#define ARM_USART_ERROR_PARITY (ARM_DRIVER_ERROR_SPECIFIC - 4)
```

Location: Driver_USART.h:156

Specified parity not supported.

20.11.5.57 ARM_USART_ERROR_STOP_BITS

```
#define ARM_USART_ERROR_STOP_BITS (ARM_DRIVER_ERROR_SPECIFIC - 5)
```

Location: Driver_USART.h:157

Specified number of stop bits not supported.

20.11.5.58 ARM_USART_ERROR_FLOW_CONTROL

```
#define ARM_USART_ERROR_FLOW_CONTROL (ARM_DRIVER_ERROR_SPECIFIC - 6)
```

Location: Driver_USART.h:158

Specified flow control not supported.

20.11.5.59 ARM_USART_ERROR_CPOL

```
#define ARM_USART_ERROR_CPOL (ARM_DRIVER_ERROR_SPECIFIC - 7)
```

Location: Driver_USART.h:159

Specified clock polarity not supported.

20.11.5.60 ARM_USART_ERROR_CPHA

```
#define ARM_USART_ERROR_CPHA (ARM_DRIVER_ERROR_SPECIFIC - 8)
```

Location: Driver_USART.h:160

Specified clock phase not supported.

20.11.5.61 ARM_USART_EVENT_SEND_COMPLETE

```
#define ARM_USART_EVENT_SEND_COMPLETE (1UL << 0)
```

Location: Driver_USART.h:201

Send completed; however USART may still transmit data.

USART Event

20.11.5.62 ARM_USART_EVENT_RECEIVE_COMPLETE

```
#define ARM_USART_EVENT_RECEIVE_COMPLETE (1UL << 1)
```

Location: Driver_USART.h:202

Receive completed.

20.11.5.63 ARM_USART_EVENT_TRANSFER_COMPLETE

```
#define ARM_USART_EVENT_TRANSFER_COMPLETE (1UL << 2)
```

Location: Driver_USART.h:203

Transfer completed.

20.11.5.64 ARM_USART_EVENT_TX_COMPLETE

```
#define ARM_USART_EVENT_TX_COMPLETE (1UL << 3)
```

Location: Driver_USART.h:204

Transmit completed (optional).

20.11.5.65 ARM_USART_EVENT_TX_UNDERFLOW

```
#define ARM_USART_EVENT_TX_UNDERFLOW (1UL << 4)
```

Location: Driver_USART.h:205

Transmit data not available (synchronous slave).

20.11.5.66 ARM_USART_EVENT_RX_OVERFLOW

```
#define ARM_USART_EVENT_RX_OVERFLOW (1UL << 5)
```

Location: Driver_USART.h:206

Receive data overflow.

20.11.5.67 ARM_USART_EVENT_RX_TIMEOUT

```
#define ARM_USART_EVENT_RX_TIMEOUT (1UL << 6)
```

Location: Driver_USART.h:207

Receive character timeout (optional).

20.11.5.68 ARM_USART_EVENT_RX_BREAK

```
#define ARM_USART_EVENT_RX_BREAK (1UL << 7)
```

Location: Driver_USART.h:208

Break detected on receive.

20.11.5.69 ARM_USART_EVENT_RX_FRAMING_ERROR

```
#define ARM_USART_EVENT_RX_FRAMING_ERROR (1UL << 8)
```

Location: Driver_USART.h:209

Framing error detected on receive.

20.11.5.70 ARM_USART_EVENT_RX_PARITY_ERROR

```
#define ARM_USART_EVENT_RX_PARITY_ERROR (1UL << 9)
```

Location: Driver_USART.h:210

Parity error detected on receive.

20.11.5.71 ARM_USART_EVENT_CTS

```
#define ARM_USART_EVENT_CTS (1UL << 10)
```

Location: Driver_USART.h:211

CTS state changed (optional).

20.11.5.72 ARM_USART_EVENT_DSR

```
#define ARM_USART_EVENT_DSR (1UL << 11)
```

Location: Driver_USART.h:212

DSR state changed (optional).

20.11.5.73 ARM_USART_EVENT_DCD

```
#define ARM_USART_EVENT_DCD (1UL << 12)
```

Location: Driver_USART.h:213

DCD state changed (optional).

20.11.5.74 ARM_USART_EVENT_RI

```
#define ARM_USART_EVENT_RI (1UL << 13)
```

Location: Driver_USART.h:214

RI state changed (optional).

20.11.6 CMSIS USART Driver Function Documentation

20.11.6.1 ARM_USART_GetVersion

```
ARM_DRIVER_VERSION ARM_USART_GetVersion()
```

Location: Driver_USART.c:59

Get driver version.

Returns:

ARM_DRIVER_VERSION

20.11.6.2 ARM_USART_GetCapabilities

```
ARM_USART_CAPABILITIES ARM_USART_GetCapabilities()
```

Location: Driver_USART.c:64

Get driver capabilities.

Returns:

ARM_USART_CAPABILITIES

20.11.6.3 ARM_USART_Initialize

```
int32_t ARM_USART_Initialize(ARM_USART_SignalEvent_t cb_event)
```

Location: Driver_USART.c:69

Initialize USART Interface.

Returns:

execution_status

Parameters

Direction	Name	Description
in	*cb_event*	Pointer to ARM_USART_SignalEvent

20.11.6.4 ARM_USART_Uninitialize

```
int32_t ARM_USART_Uninitialize()
```

Location: Driver_USART.c:73

De-initialize USART Interface.

Returns:

execution_status

20.11.6.5 ARM_USART_PowerControl

```
int32_t ARM_USART_PowerControl(ARM_POWER_STATE state)
```

Location: Driver_USART.c:77

Control USART Interface Power.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*state*	Power state

20.11.6.6 ARM_USART_Send

```
int32_t ARM_USART_Send(const void * data, uint32_t num)
```

Location: Driver_USART.c:93

Start sending data to USART transmitter.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*data*	Pointer to buffer with data to send to USART transmitter
`in`	*num*	Number of data items to send

20.11.6.7 ARM_USART_Receive

```
int32_t ARM_USART_Receive(void * data, uint32_t num)
```

Location: Driver_USART.c:97

Start receiving data from USART receiver.

Returns:

execution_status

Parameters

Direction	Name	Description
`out`	*data*	Pointer to buffer for data to receive from USART receiver
`in`	*num*	Number of data items to receive

20.11.6.8 ARM_USART_Transfer

```
int32_t ARM_USART_Transfer(const void * data_out, void * data_in, uint32_t num)
```

Location: Driver_USART.c:101

Start sending/receiving data to/from USART transmitter/receiver.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*data_out*	Pointer to buffer with data to send to USART transmitter
`out`	*data_in*	Pointer to buffer for data to receive from USART receiver
`in`	*num*	Number of data items to transfer

20.11.6.9 ARM_USART_GetTxCount

```
uint32_t ARM_USART_GetTxCount()
```

Location: Driver_USART.c:105

Get transmitted data count.

Returns:

number of data items transmitted

20.11.6.10 ARM_USART_GetRxCount

```
uint32_t ARM_USART_GetRxCount()
```

Location: Driver_USART.c:109

Get received data count.

Returns:

number of data items received

20.11.6.11 ARM_USART_Control

```
int32_t ARM_USART_Control(uint32_t control, uint32_t arg)
```

Location: Driver_USART.c:113

Control USART Interface.

Returns:

common execution_status and driver specific usart_execution_status

Parameters

Direction	Name	Description
`in`	*control*	Operation
`in`	*arg*	Argument of operation (optional)

20.11.6.12 ARM_USART_GetStatus

```
ARM_USART_STATUS ARM_USART_GetStatus()
```

Location: Driver_USART.c:117

Get USART status.

Returns:

USART status ARM_USART_STATUS

20.11.6.13 ARM_USART_SetModemControl

```
int32_t ARM_USART_SetModemControl(ARM_USART_MODEM_CONTROL control)
```

Location: Driver_USART.c:121

Set USART Modem Control line state.

Returns:

execution_status

Parameters

Direction	Name	Description
`in`	*control*	ARM_USART_MODEM_CONTROL

20.11.6.14 ARM_USART_GetModemStatus

```
ARM_USART_MODEM_STATUS ARM_USART_GetModemStatus()
```

Location: Driver_USART.c:125

Get USART Modem Status lines state.

Returns:

modem status ARM_USART_MODEM_STATUS

20.11.6.15 ARM_USART_SignalEvent

```
void ARM_USART_SignalEvent(uint32_t event)
```

Location: Driver_USART.c:129

Signal USART Events.

Returns:

none

Parameters

Direction	Name	Description
`in`	*event*	USART_events notification mask

CHAPTER 21

swmTrace Reference

The swmTrace library functions and macros provide a logging utility that helps you to debug an application running on the Arm Cortex-M33 core.

21.1 Summary

Macros

- swmLogVerbose : Shortcut macro for verbose logging.
- swmLogInfo : Shortcut macro for informational logging.
- swmLogWarn : Shortcut macro for warnings.
- swmLogError : Shortcut macro for errors.
- swmLogFatal : Shortcut macro for fatal errors.
- swmLogTestPass : Shortcut macro for test PASS indicators.
- swmLogTestFail : Shortcut macro for test FAIL indicators.

Functions

- swmTrace_init : Trace initialization function.
- swmTrace_txInProgress : Provides indication if transmission is in progress.
- swmTrace_printf : This provides a printf-like implementation for all possible trace mechanisms.
- swmTrace_vprintf : This provides a vprintf-like implementation for all possible trace mechanisms.
- swmTrace_getch : A method to allow characters to be passed from the logging target to the traced application.
- swmLog : A general logging method that allows us to output only trace messages if a particular log level has been selected.

21.2 swmTrace Reference Macro Definition Documentation

21.2.1 swmLogVerbose

```
#define swmLogVerbose swmLog(SWM_LOG_LEVEL_VERBOSE, __VA_ARGS__)
```

Location: swmTrace_api.h:137

Shortcut macro for verbose logging.

21.2.2 swmLogInfo

```
#define swmLogInfo swmLog(SWM_LOG_LEVEL_INFO, __VA_ARGS__)
```

Location: swmTrace_api.h:142

Shortcut macro for informational logging.

21.2.3 swmLogWarn

```
#define swmLogWarn swmLog(SWM_LOG_LEVEL_WARNING, __VA_ARGS__)
```

Location: swmTrace_api.h:147

Shortcut macro for warnings.

21.2.4 swmLogError

```
#define swmLogError swmLog(SWM_LOG_LEVEL_ERROR, __VA_ARGS__)
```

Location: swmTrace_api.h:152

Shortcut macro for errors.

21.2.5 swmLogFatal

```
#define swmLogFatal swmLog(SWM_LOG_LEVEL_FATAL, __VA_ARGS__)
```

Location: swmTrace_api.h:157

Shortcut macro for fatal errors.

21.2.6 swmLogTestPass

```
#define swmLogTestPass swmLog(SWM_LOG_TEST_PASS, __VA_ARGS__)
```

Location: swmTrace_api.h:162

Shortcut macro for test PASS indicators.

21.2.7 swmLogTestFail

```
#define swmLogTestFail swmLog(SWM_LOG_TEST_FAIL, __VA_ARGS__)
```

Location: swmTrace_api.h:167

Shortcut macro for test FAIL indicators.

21.3 SWMTRACE REFERENCE FUNCTION DOCUMENTATION

21.3.1 swmTrace_init

```
void swmTrace_init(const uint32_t * configuration, uint32_t size)
```

Location: swmTrace_api.h:80

Trace initialization function.

This method allows the tracing functions to be initialized in a general way, allowing different trace options to be supplied, depending on the type of trace library selected. configuration Consists of an array of 32-bit words that define the selected initialization options. size Indicates the number of options provided.

NOTE: The list of options can be set up as a superset of all the possible options, and only the ones required for a given trace library are used. The possible options are defined in the swmTrace_options.h file.

NOTE: The list of options can be set up as a superset of all the possible options, and only the ones required for a given trace library are used. The possible options are defined in the swmTrace_options.h file.

Parameters

Direction	Name	Description
`in`	*configuration*	Consists of an array of 32-bit words that define the selected initialization options.
`in`	*size*	Indicates the number of options provided.

21.3.2 swmTrace_txInProgress

```
bool swmTrace_txInProgress()
```

Location: swmTrace_api.h:86

Provides indication if transmission is in progress.

Returns:

True if a string is being transmitted; false otherwise.

21.3.3 swmTrace_printf

```
void swmTrace_printf(const char * sFormat, ... )
```

Location: swmTrace_api.h:94

This provides a printf-like implementation for all possible trace mechanisms.

Parameters

Direction	Name	Description
in	*sFormat*	Defines the format of the string to print. This is followed by a variable number of arguments.

21.3.4 swmTrace_vprintf

```
void swmTrace_vprintf(const char * sFormat, va_list * pParamList)
```

Location: swmTrace_api.h:103

This provides a vprintf-like implementation for all possible trace mechanisms.

Parameters

Direction	Name	Description
`in`	*sFormat*	Defines the format of the string to print.
`in`	*pParamList*	Defines a pointer to a va_list object in a form similar to vprintf.

21.3.5 swmTrace_getch

```
bool swmTrace_getch(char * ch)
```

Location: swmTrace_api.h:112

A method to allow characters to be passed from the logging target to the traced application.

Returns:

True if a valid character has been returned; false otherwise.

Parameters

Direction	Name	Description
`in`	*ch*	A pointer to a character object which holds the returned character.

21.3.6 swmLog

```
void swmLog(uint32_t level, const char * sFormat, ... )
```

Location: swmTrace_api.h:121

A general logging method that allows us to output only trace messages if a particular log level has been selected.

Parameters

Direction	Name	Description
`in`	*level*	The level of this log message. Only messages which have a level equal to or higher than the currently selected level are output.
`in`	*sFormat*	The format of the output string, as per printf.

APPENDIX A

Glossary

The following abbreviations and terms are used in this manual:

ADC	analog-to-digital converter
Arm Cortex-M3 processor	processing core that can be used to configure the overall system after initial boot, and is typically used to coordinate the system interaction with external components
ASCC	audio sink clock counter
ASRC	asynchronous sample rate converter
CCO	current-controlled oscillator
CFX Digital Signal Processor	processing core that is used to configure the system at initial boot; this core can also be used to configure the overall system, and coordinate the flow of signal data progressing through the system
CID	chip identifier
CMSIS	Cortex Microcontroller Software Interface Standard
CRC	cyclic redundancy check
DAC	digital-to-analog converter
DIO	digital input/output
DMA	direct memory access
DSP	digital signal processor
ECC	error correcting code
EEPROM	electrically-erasable, programmable, read-only memory, a type of non-volatile memory

eMMC	embedded multi-media controller
FIFO	A section of memory that is treated as a buffer where components that use this memory use pointers to handle data in a first-in, first-out manner.
Filter Engine	a specialized processing core that is optimized for basic audio filtering of both decimated and undecimated data; commonly used in the creation of a low-delay path
GPIO	general-purpose input/output
HEAR Configurable Accelerator	a microcode configurable signal processing engine used to execute a variety of signal processing functions
HIZ	high impedance
I^2C	inter-IC communication protocol
I^2S	inter-IC sound protocol
I^3C	improved inter-IC communication protocol (MIPI Alliance)
INL	integral non-linearity
IOC	input/output controller
JTAG	joint test action group (developer of IEEE standard 1149.1-1990)
LDP	low delay path
LPDSP32	low-power digital signal processor with a 32-bit data path; a processing core that is used to accelerate codec, neural network, and similar functions
LSAD	low-speed analog-to-digital
LSB	

	least significant bit
LUT	look up table
MCU	microcontroller unit
memory block instance	an instance of RAM, ROM, or flash memory
memory bus	a communication system that transfers data between memory or memory-mapped components and the cores that operate on that data
memory space	a mapping of memory and memory-mapped items to a memory address; synonym for address space
MSB	most significant bit
MUX	multiplexer, selector of one signal from many
NNA	Neural Network Accelerator
NVIC	nested vectored interrupt controller
NVM	non-volatile memory
PMU	power management unit
POR	power-on-reset
PSU	power supervisory unit
QMF	quadrature mirror filter
RAM	random-access memory
ROM	read-only memory

SCL	serial clock (part of an I^2C or I^3C bus)
SDA	serial data (part of an I^2C or I^3C bus)
SDM	sigma-delta modulator
SPI	Serial Peripheral Interface
SWD	serial wire debug, two-wire interface used for communication with Arm cores
SWJ-DP	serial wire and JTAG debug port
UART	universal asynchronous receiver-transmitter
VBAT	supply voltage to the system (typically supplied from a battery)
VDDA	supply voltage for the non-RF analog blocks
VDDC	supply voltage for the digital logic
VDDIF	supply voltage supporting external components and their interfaces
VDDM	supply voltage for the memories
*VDDO**	the four input supplies for the digital I/O pads, including the debug ports (I^2C, SWJ-DP)
VMIC	supply voltage for microphone units
VREF	bandgap reference voltage
VREG	regulated reference supply voltage
VSSA	analog ground supply reference

WDF	
	wave digital filter

PUBLICATION ORDERING INFORMATION

LITERATURE FULFILLMENT:
Literature Distribution Center for onsemi
19521 E. 32nd Pkwy, Aurora, Colorado 80011 USA
Phone: 303-675-2175 or 800-344-3860 Toll Free USA/Canada
Fax: 303-675-2176 or 800-344-3867 Toll Free USA/Canada
Email: orderlit@onsemi.com

N. American Technical Support:
800-282-9855 Toll Free USA/Canada
Europe, Middle East and Africa Technical Support: Phone: 421 33 790 2910

onsemi Website: www.onsemi.com

Order Literature: http://www.onsemi.com/orderlit

For additional information, please contact your local Sales Representative

M-20875-004