S32SDK Documentation

S32K116 EAR 1.8.7

Generated by Doxygen 1.8.10

Contents

1 S32 SDK

Introduction

This topic provides an introduction to the S32 software development kit (S32 SDK), including intended audience, purpose and scope, and detailed sections on technical considerations.

Copyright © 2016 NXP Semiconductor

Intended Audience

S32 SDK documentation is written for software developers and system engineers who have a technical background, and a working knowledge of embedded programming. The audience for the S32 SDK are users of S32 Processors.

Purpose and Scope

The S32 SDK is a embedded oriented development kit. It allows users to

1. Evaluate and explore the features of the S32 processors; experience how they are supported by working "out of the box" on NXP development boards.

2. Develop embedded solutions; the NXP SDK is thoroughly tested from development to production.

S32 SDK Architecture Overview

The S32 SDK is an extensive suite of robust hardware interface and hardware abstraction layers, peripheral drivers, RTOS, stacks, and middleware designed to simplify and accelerate application development on NXP S32 SOCs. The addition of Processor Expert technology for software and board configuration provides unmatched ease of use and flexibility. Included in the S32 SDK is full source code under a permissive open-source license for all hardware abstraction and peripheral driver software. See the Release Notes for details. The S32 SDK consists of the following runtime software components written in C:
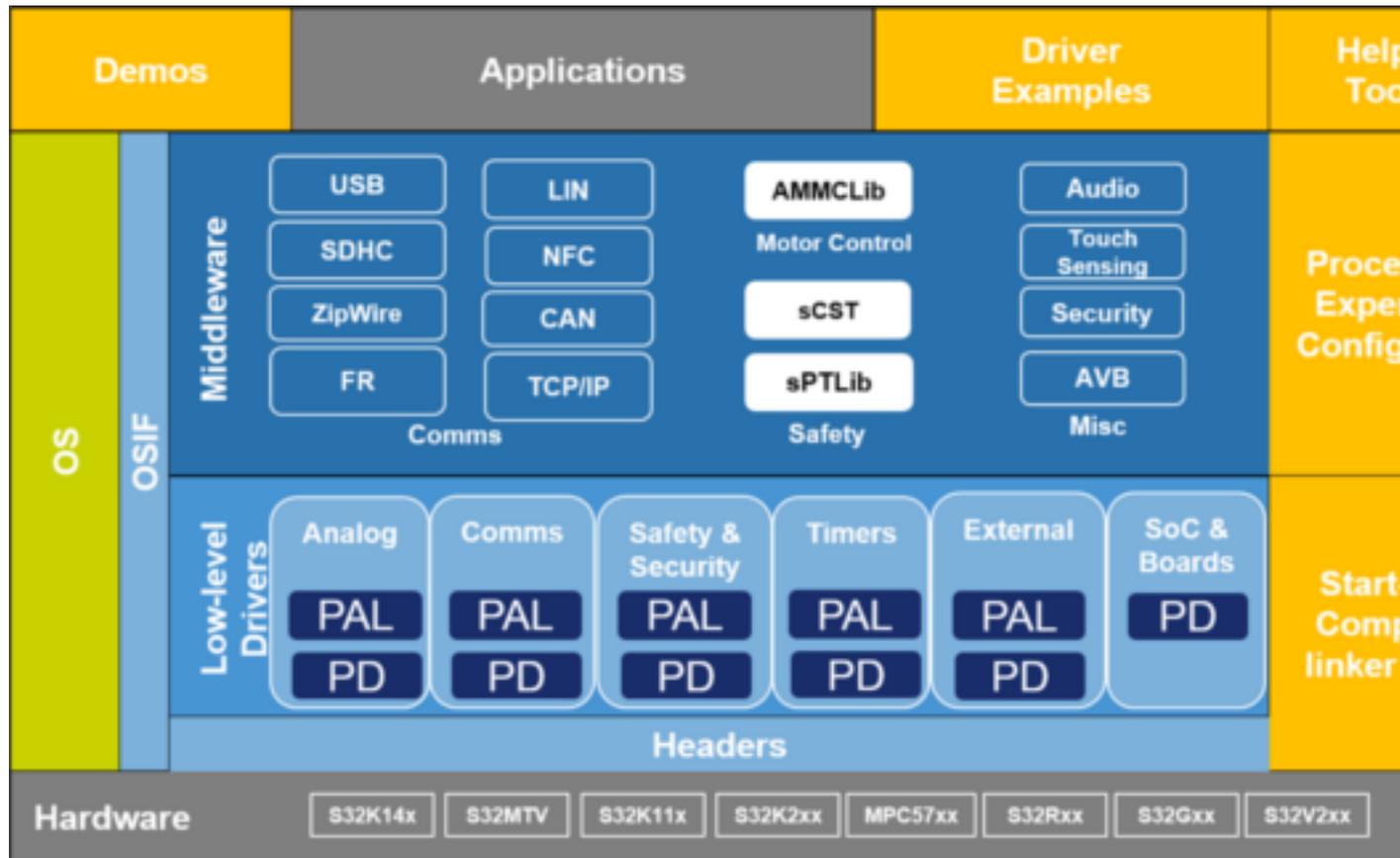
2 Components

Header file

The S32 SDK contains a device-specific header files which provide direct access to the peripheral registers. Each supported device in S32 SDK has an overall System-on-Chip (SoC) memory-mapped header file. This header file contains the memory map and register base address for each peripheral and the IRQ vector table with associated vector numbers.

Feature Header File

The PAL is designed to be reusable regardless of the peripheral configuration differences from one SOC device to another. An overall Peripheral Feature Header File is provided for device to define the feature or configuration differences for each SOC sub-family device.

Peripheral Abstraction Layer

The PAL provides unified interfaces for families of peripherals, allowing for cross-platform compatibility of application code. The main goal is to provide an application programming interface that is independent of the underlying peripheral implementation.

The PAL supports all instances of each peripheral from a certain family instantiated on the SOC by using a simple integer parameter for the peripheral instance number.

The PAL instances should be configured bearing in mind possible limitations of the underlying peripherals - some features may not be supported on some hardware modules. It is the user's responsibility to correctly handle hardware resources, especially when porting the application to a different platform.

The PAL drivers can be found in the platform/pal directory.

Peripheral Drivers

The Peripheral Drivers are high-level drivers that implement high-level logic transactions based on an internal register access abstraction layer, other Peripheral Drivers, and/or System Services. For example, the UART register access abstraction layer mainly focuses on byte-level basic functional primitives, while the UART Peripheral Driver operates on an interrupt-driven level using data buffers to transfer a stream of bytes. In general, if a driver, that is mainly based on one peripheral, interfaces with functions beyond the register access abstraction layer and/or requires interrupt servicing, the driver is considered a high-level Peripheral Driver.

The Peripheral Drivers support all instances of each peripheral instantiated on the SOC by using a simple integer parameter for the peripheral instance number. The user of the Peripheral Driver does not need to know the peripheral memory-mapped base address.

The Peripheral Drivers operate on a high-level logic that requires data storage for internal operation context handling. However, the Peripheral Drivers do not allocate this memory space. Rather, the user passes in the memory for the driver internal operation through the driver initialization function.

The Peripheral Drivers are designed to handle the entire functionality for a targeted use-case. An application should be able to use only the Peripheral Driver to accomplish its purpose.

The Peripheral Drivers can be found in the platform/drivers directory.

System Services

The System Services contain a set of software entities that can be used by the Peripheral Drivers. They may be used with PAL Drivers to build the Peripheral Drivers or they can be used by an application directly. The following sections describe each of the System Services software entities. These System Services are in the platform/drivers directory.

Interrupt Manager

The Interrupt Manager provides functions to enable and disable individual interrupts within the Nested Vector Interrupt Controller (NVIC). It also provides functions to enable and disable the ARM core global interrupt (via the CPSIE and CPSID instructions) for bare-metal critical section implementation. In addition to providing functions for interrupt enabling and disabling, the Interrupt Manager provides Interrupt Service Routine (ISR) registration that allows the application software to register or replace the interrupt handler for a specified IRQ vector. The drivers do not set interrupt priorities. The interrupt priority scheme is entirely determined by the specific application logic and its setting is handled by the user application. The user application manages the interrupt priorities.

Clock Manager

The Clock Manager provides centralized clock-related functions for the entire system. It can dynamically set the system clock and perform clock gating/un-gating for specific peripherals. The Clock Manager also maintains knowledge of the clock sources required for each peripheral and provides functions to obtain the clock frequency for each supported clock used by the peripheral. The Clock Manager provides a notification framework which the software components, such as drivers, uses to register callback functions and execute the predefined code flow during the clock mode transition.

Power Manager

The Power Manager provides centralized power-related functions for the entire system. It dynamically sets the system power mode. The Power Manager provides a notification framework which the software components, such as drivers, uses to register callback functions and execute the predefined code flow during the power mode transition.

Examples

The examples provided show how to build user applications using the S32 SDK. The examples can be found in the top-level example directory. For details please see Examples and Demos .

3   Supported Platforms

Supported board and SoC versions can be found in the Release Notes. (SDK\ReleaseNotes.pdf)

4   Installation

Prerequisites

SDK can be used in two ways: bundled in S32 Design Studio and standalone.

S32 SDK is delivered bundled in the S32 Design Studio. In this case it's already configured and ready to use.

S32 SDK is also delivered through a standalone installer. Using the standalone installer is recommended when using a compiler which is not supported in S32 Design Studio or when the graphical interface is not required. In this case the installer can configure an existing S32 Design Studio to use the configuration files delivered in the installer.

If the integration with the S32 Design Studio is not needed the path to S32 Design Studio can be left empty – and in this case only the S32 SDK will be installed and configured.

Steps

1. Start the installer S32_SDK_<ReleaseSpecifc>.exe

2. Set the destination folder for the SDK, give optional location of S32DS and install. Example of S32DS path: C:\NXP\S32ARMv1.3

3. Start using the SDK by creating a new project or importing a project

Background

The installer does the following things in background:

- Puts the SDK in the selected destination directory.

- Appends to S32SDK_PATH the path of the SDK.

   - Note: Please make sure you uninstall previous SDK so that this variable will be empty.

- Copies necessary files into S32 Design Studio installation location.

- Overwrites existing SDK from S32 Design Studio with the version from destination directory

Uninstaller

Use "uninst.exe" to uninstall the SDK.

Note: If you want to reinstall the SDK please use a clean copy of S32DS. When you uninstall this does not delete the copied files (ex: Config_01.pez), so a clean copy is needed.

5 Build Tools

Introduction

S32 SDK supports and is tested with multiple compiler toolchains.

Note

> The toolchain list, versions and their options specific for the platform and release can be found in the Release Notes. (SDK\ReleaseNotes.pdf)

Makefiles

Multiple makefile projects are provided in the 'examples' folder, for all supported compilers. These projects can be modified by adding application code, or the makefiles can be reused in different projects, after reconfiguring the paths/variables. Please note that these projects require the designated compiler to be already installed on the host; also, the makefile path to compiler executable must be updated before running make utility.

S32 Design Studio

S32 Design Studio for ARM is delivered with arm-gcc compiler support included ("{S32_Design_Studio_install_↩ path}\Cross_Tools\gcc-arm-none-eabi). Eclipse plugins for gcc are already installed in S32 Design Studio IDE, so new projects for this toolchain can be created and built directly from the IDE. To add S32 SDK source files to a clean S32 Design Studio project, eclipse "linked resources" feature can be used: project properties->New->Folder->↩ Advanced->'Link to alternate location' (e.g. "{S32_SDK_PATH}"). For S32 Design Studio project with Processor Expert support, please import a project from "{S32_SDK_PATH} Name".

6 IDE Support

S32 Design Studio

- S32 Design Studio for ARM is delivered with Processor Expert support included. Please see Configuration chapter.

- To configure the S32 SDK path of the project, eclipse "S32 SDK Specific" feature can be used: patch project properties->Processor Expert->S32 SDK Specific->SDK path

- Processor Expert repositories and paths can be configured as it follows: Window -> Preferences -> Processor Expert -> Repositories and Paths.

- S32 Design Studio projects can be imported from S32 SDK package. Please see Examples and Demos chapter.

IAR Embedded Workbench

- There is no configuration support for S32 SDK in IAR.

- IAR Embedded Workbench projects can be imported from S32 SDK package. Please see Examples and Demos chapter.

7 Configuration

Processor Expert software allows generation of configuration structures for peripheral drivers from S32 SDK. With the help of Eclipse based graphical interface where you can configure your driver and generate corresponding

configuration structure. This tool doesn't generate source code for S32 family, it only generates configurations data structures.

Processor Expert generates configuration header files that are included by application source code. The configuration data structures from these files are defined in S32 SDK. All these header files are generated by this tool in ${ProjName}/Generated_Code directory.

Peripheral drivers are not stored directly in the project directory, these drivers are stored in S32 SDK repository. Shared peripheral drivers repository is advantageous when more projects should share the same version of peripheral drivers. In this case, peripheral drivers are not physically placed in the project directory but each project is virtually linked with shared, common repository from S32 SDK. This way the management of the projects' drivers can be done in one place and any changes made in the shared repository is automatically distributed across all of the linked projects, for example in case of bug fixing or library update and also backup or archiving of the peripheral drivers versions is very simple.

8   Acronyms and Abbreviations

Acronym	Description
CPSIE, CPSID	Change Processor State Interrupt Enable / Disable
EAR	Early Access Release
EVB	Evaluation board
PAL	Peripheral Abstraction Layer
IRQ	Interrupt Request
ISR	Interrupt Service Routine
LLWU	Low Leakage Wakeup Unit
NVIC	Nested Vector Interrupt Controller
RTOS	Real Time Operating System
S32DS	S32 Design Studio
SDK	Software Development Kit
SOC	System-on-Chip
UART	Universal Asynchronous Receiver / Transmitter

9   MISRA Compliance

This section describes how the S32 SDK project addresses MISRA Compliance.

The S32 SDK SW components which are implemented to be compliant with MISRA C 2012 are:

- all drivers & PALs

- generated driver code (including Cpu.c & .h)

- main.c (generated via graphical configurator)

Violations of MISRA C 2012 guidelines which remain not fixed, shall be documented as deviations at file level.

Other SW components included in the S32 SDK package which are not subject to MISRA C 2012 compliance:

- demo_apps & driver examples

- FreeRTOS

10   Error detection and reporting

S32 SDK drivers can use a mechanism to validate data coming from upper software layers (application code) by performing a number of checks on input parameters' range or other invariants that can be statically checked (not

dependent on runtime conditions). A failed validation is indicative of a software bug in application code, therefore it is important to use this mechanism during development.

The validation is performed by using DEV_ASSERT macro. A default implementation of this macro is provided in this file. However, application developers can provide their own implementation in a custom file. This requires defining the CUSTOM_DEVASSERT symbol with the specific file name in the project configuration (for example: -DCUSTOM_DEVASSERT="custom_devassert.h")

The default implementation accommodates two behaviors, based on DEV_ERROR_DETECT symbol:

- When DEV_ERROR_DETECT symbol is defined in the project configuration (for example: -DDEV_ER↩ROR_DETECT), the validation performed by the DEV_ASSERT macro is enabled, and a failed validation triggers a software breakpoint and further execution is prevented (application spins in an infinite loop) This configuration is recommended for development environments, as it prevents further execution and allows investigating potential problems from the point of error detection.

- When DEV_ERROR_DETECT symbol is not defined, the DEV_ASSERT macro is implemented as no-op, therefore disabling all validations. This configuration can be used to eliminate the overhead of development-time checks.

It is the application developer's responsibility to decide the error detection strategy for production code: one can opt to disable development-time checking altogether (by not defining DEV_ERROR_DETECT symbol), or one can opt to keep the checks in place and implement a recovery mechanism in case of a failed validation, by defining CUSTOM_DEVASSERT to point to the file containing the custom implementation.

11 Examples and Demos

Applications that show the user how to initialize the peripherals for the basic use cases

11.1 Introduction

S32 SDK examples structure:

- Demo applications (SDK/examples/<CPU>/demo_apps), are demo applications for various IDEs and compilers. Also this examples are using more advanced use-cases - FreeRTOS integration, LIN Stack, FlexCAN usage and Clock Setup.

- Driver Examples (SDK/examples/<CPU>/driver_examples), are simple applications which exemplify a basic use-case for a specific driver.

Examples are available for:

- S32K116 Examples

11.2 Usage

11.2.1 How to build

For makefile project

There are makefile projects in all compilers supported. In order to used them:

- **Make** utility (eg. GNU Make)

- **Toolchain** (eg. GCC Toolchain)

- **Make sure the make and compiler are in Path (for Microsoft Windows : System -$>$ Environmental Variables)**

- From command line execute the makefile: **make all**

The makefiles generate binary files for both RAM and FLASH configurations.

For IAR Embedded Workbench

From IAR Workbench for ARM use File $>$ Open $>$ Workspace and browse to the desired project. After the project was opened you can see the files in "Workspace Files". Finally, the project can be executed from Project $>$ Download and Debug. Make sure that the debug probe you are using is selected and configured in Project options $>$ Debugger $>$ Driver.

For S32 Design Studio

From S32 Design Studio (See Release notes for the S32 Design Studio version), go to File -$>$ New -$>$ New Project from Example and select the example you wish to import. This will copy the example project into workspace. Next steps:

- Use Processor Expert to configure the components used in the example

- Use Project $>$ Generate Processor Expert for generating the configuration

- Use Project $>$ Build to build the project

- Use Project $>$ Debug and launch your preferred debug configuration

11.2.2   How to debug

This section explains how to upload and debug the binary files generated after build. This assumes that you have a debug probe(see release notes for supported debug probes) and a debug software installed on the machine.

Generic steps:

1. Launch the debug software

2. Load the binary file into the MCU

3. Execute the application

Loading with Segger JLink:

- Download and install the latest drivers and GDB server, named *Software and documentation pack*, from their `site`

- Download your favorite GDB client (eg. arm-none-eabi-gdb)

- Browse to JLink installation folder and launch **JLinkGDBServer**

- Select the appropriate part from the device list and click on **OK**

- Open the GDB client and connect to the configured port - by default localhost:2331

- Upload the file and execute (see GDB client user manual for details regarding the commands used)

The following table is a small list of commands used in GNU ARM GDB with JLinkGDBServer to connect and run the application:

Command	Description
target remote:PortNumber	Connect to the remote target at a specified port. Please replace PortNumber with the port configured in the GDB server.
monitor reset	Reset the target MCU
monitor halt	Halt the target MCU
file ApplicationName.elf	Load the file and symbols. Please change ApplicationName with your application name
load	Download the executable to the target MCU
continue	Begin executing the application

Loading with PEmicro OpenSDA/MultiLink:

- Download and install the latest drivers and GDB server, named *P&E GDB Server for Kinetis with Windows GUI*, from their `site`

- Download your favorite GDB client (eg. arm-none-eabi-gdb)

- Browse to PEmicro GDB Server installation folder and launch **P&E GDB Server for Kinetis**

- Select the appropriate part from the device list and click on **Connect**

- Open the GDB client and connect to the configured port - by default localhost:7224

- Upload the file and execute (see GDB client user manual for details regarding the commands used)

The following table is a small list of commands used in GNU ARM GDB with PEmicro GDB server to connect and run the application:

Command	Description
target remote:PortNumber	Connect to the remote target at a specified port. Please replace PortNumber with the port configured in the GDB server.
monitor reset	Reset the target MCU
file ApplicationName.elf	Load the file and symbols. Please change ApplicationName with your application name
load	Download the executable to the target MCU
continue	Begin executing the application

11.2.3 Using terminal emulator

To run the examples that use LPUART to help you visualize data you must download a terminal emulator (eg. Putty, Termite, TeraTerm) and configure it.

Unless otherwise noted the standard communication parameters are:

- 115200 baud

- One stop bit

- No parity

- No flow control

Example configuration for Termite using OpenSDA

1) Download Termite from their `site`
2) Run the installer. Wait for the installation to be completed
3) Go to **Start** -> **All Programs** -> **Termite** and launch the program. The window from Fig.1 will appear ...

Termite window
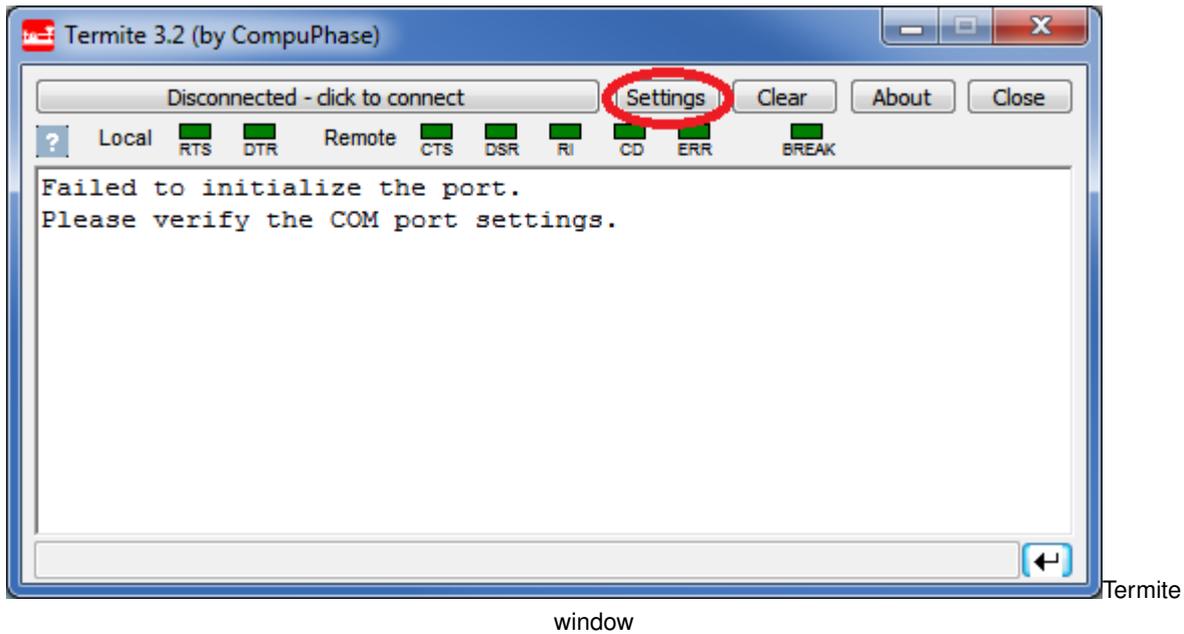
4) Click on **Settings**

5) As seen in Fig.2, configure the following communication parameters:

- **Port(1)** : COMx - where x must be replaced with the COM port number

- **Baud Rate(2)** : 115200

- **Data Bits(3)** : 8

- **Stop Bits(4)** : 1

- **Parity(5)** : None

- **Flow Control(6)** : None

- **Forward(7)** : None

Settings window
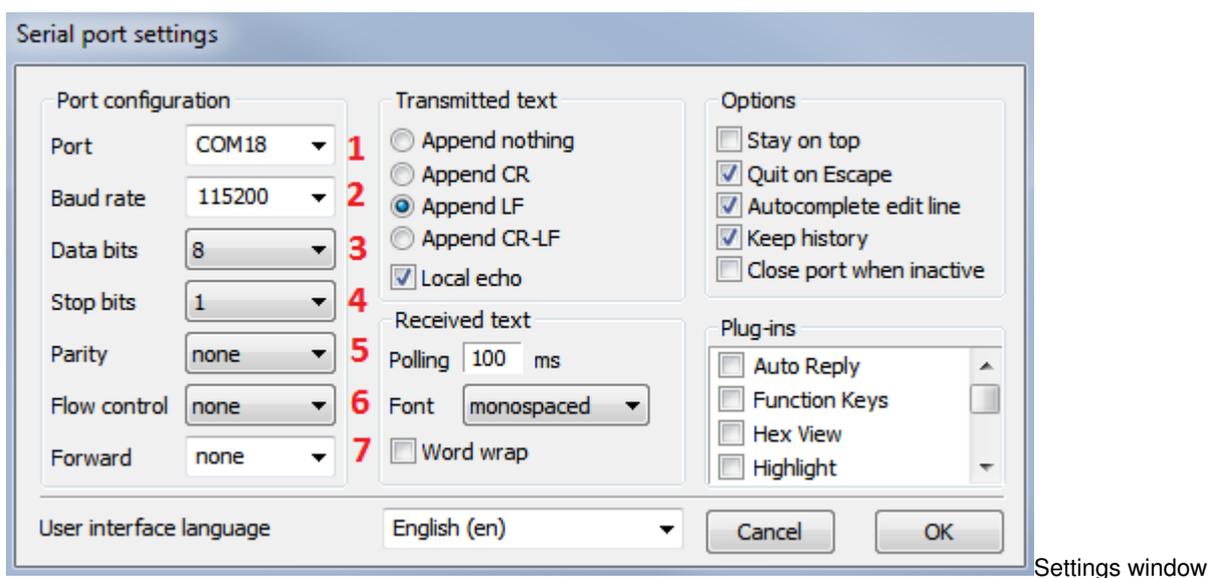
6) Click **OK**. Now the terminal should be configured

Note

> For further help consult the terminal's documentation

11.3 S32K116 Examples

Demo applications and driver examples for S32K116

Examples for S32K116 are separated into two groups:

- Demo Applications

- Driver Examples

11.3.1 Demo Applications

Applications that show more advanced use cases

Available demo applications:
Click on one of the project to see the corresponding documentation

- FlexCAN Encrypted

- Hello World - Makefile

- Hello World

- LIN MASTER

- LIN SLAVE

11.3.1.1 FlexCAN Encrypted

Demo application showing the FlexCAN functionalities

Note

> **If running the encrypted communication:** The encryption uses the first non-volatile user key, which needs to be configured by running the **CSEc Key Configuration** in the driver examples folder.
> **Encrypted communication works only for CSEc enabled parts.** SIM_SDID indicates whether CSEc is available on your device.
> **If one of the user keys was loaded using the CSEc Key Configuration, any further full erase of the Flash requires a Challenge-Authentication process.** This can be done by running the CSEc Key Configuration example again and setting the ERASE_ALL_KEYS macro to 1.

Application description

The purpose of this demo application is to show you the usage of the FlexCAN module configured to use Flexible Data Rate and the CSEc module from the S32K116 CPU using the S32 SDK API.

- In the first part, the application will setup the board clocks, pins and other system functions.

- Then it will configure the FlexCAN module features such as FD, Bitrate and Message buffers

- The application will wait for frames to be received on the configured message buffer or for an event raised by pressing one of the two buttons which will trigger a frame send to the recipient.

- Pressing SW3 button of board 1 shall trigger a CAN transfer that results in toggling the RED led on board 2.

- Pressing SW2 button of board 1 shall trigger a CAN transfer that results in toggling the GREEN led on board 2.

- Pressing both SW3 and SW2 buttons shall enable the encrypted communication. This event is signaled by the BLUE led being ON.

- The frames are sent in plain text by default.

- This demo application requires two boards, one configured as master and the other one configured as slave (see MASTER/SLAVE defines in application code).

Note

> Due to limited RAM size, this example contains only one build configuration for target flash memory (and the associated debug configuration).

Prerequisites

To run the example you will need to have the following items:

- 2 S32K116 boards

- 1 Power Adapter 12V

- 3 Dupont female to famale cables

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q100
CAN HIGH (∗)	CAN HIGH - J13.1
CAN LOW (∗)	CAN LOW - J13.2
GND (**GND**)	GND - J13.4
BUTTON 0 (**PTC13**)	SW3 - wired on the board
BUTTON 1 (**PTC12**)	SW2 - wired on the board
RED_LED (**PTD16**)	RGB_RED - wired on the board
GREEN_LED (**PTD15**)	RGB_GREEN - wired on the board
BLUE_LED (**PTE8**)	RGB_GREEN - wired on the board

(∗) **Those lines must be modulated using a transceiver, if it is not specified the boards already include the CAN transceiver**

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **flexcan_encrypted_↩ s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**flexcan_encrypted_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
flexcan_encrypted_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
flexcan_encrypted_s32k116_debug_flash_↩ pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.1.2 Hello World - Makefile

Basic application that presents the project scenarios for S32 SDK using makefiles for various compilers

Application description

The purpose of this demo is to provide the user with an out-of-the box example application for S32K146 platform, using S32 SDK. The demo uses Pins and Clock driver to initialize the MCU and to toggle two LEDs alternatively.

There are five projects delivered with this package:

- Makefile project (GCC compiler)

- Makefile project (GHS compiler)

- Makefile project (IAR compiler)

Note

> For information about how to run the makefile please refer to Usage

11.3.1.3 Hello World

Basic application that presents the project scenarios for S32 SDK

Application description

The purpose of this demo is to provide the user with an out-of-the box example application for S32K116 platform, using S32 SDK. The demo uses hardware abstraction layer primitives for PCC and PORT modules in order to toggle two LEDs alternatively.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K116-MB
RED_LED (**PTD15**)	RGB_RED - wired on the board	J12.17 - J11.31
GREEN_LED (**PTD16**)	RGB_GREEN - wired on the board	J12.16 - J11.30

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **hello_world_S32K116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**hello_world_S32K116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
hello_world_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
hello_world_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.1.4 LIN MASTER

Example that shows the usage of the LIN stack in master mode

Application description

This example demonstrates the LIN communication between S32K116 EVB Master and Slave using unconditional frames.

- The Master SeatECU is in NormalTable schedule table and it uses the LIN frame Motor1State_Cycl to receive temperature signal Motor1Temp from Slave Motor1 and send selection signal Motor1Selection to Slave Motor1 by frame Motor1Control.

- If value of temperature signal is higher than MOTOR1_OVER_TEMP value, Master SeatECU will send STOP command through Motor1Selection signal to stop motor.

- If value of temperature signal is in range from MOTOR1_MAX_TEMP value to MOTOR1_OVER_TEMP value, master SeatECU will send DECREASE MOTOR SPEED command through Motor1Selection signal to reduce motor speed.

- If value of temperature signal is lower than MOTOR1_MAX_TEMP value, master will send INCREASE MO←TOR SPEED command through Motor1Selection signal to increase motor speed.

- When users press button SW2 on the Master board, the Master SeatECU switches its schedule table to go-to-sleep table. So the Slave and Master enter sleep mode, RGB LEDS are off.

- When LIN cluster is in sleep mode, users press button SW3 on the Master board, the Master board sends a wakeup signal to wakeup slave nodes, then switches its table to NormalTable.

Note

This example will work only in FLASH target.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V

- 5 Dupont female to female cable

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K1xxCVD-Q48 with S32K116 chip

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K1xxCVD-Q48
BUTTON 2 (**PTD5**)	SW3 - wired on the board	BTN3 - wired on the board
BUTTON 1 (**PTD3**)	SW2 - wired on the board	BTN2 - wired on the board
RED_LED (**PTD16**)	RGB_RED - wired on the board	J12.15 - J11.31
GREEN_LED (**PTD15**)	RGB_GREEN - wired on the board	J12.18 - J11.30
BLUE_LED (**PTE8**)	RGB_BLUE - wired on the board	J13.23 - J11.29
GND (**GND**)	J11-4 - Slave GND	J6 - Slave GND
LIN ($*$)	J11-1 - Slave LIN	J48.4 - Slave LIN

($*$) **Those lines must be modulated using a transceiver, if it is not specified the boards already include the LIN transceiver**

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **lin_master_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**lin_master_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
lin_master_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
lin_master_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

This example isn't successful run on RAM because RAM size of S32K116 is small. For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.1.5 LIN SLAVE

Example that shows the usage of the LIN stack in slave mode

Application description

This example demonstrates the LIN communication between S32K116 EVB Master and Slave using unconditional frames.

- The Master SeatECU is in NormalTable schedule table and it uses the LIN frame Motor1State_Cycl to receive temperature signal Motor1Temp from Slave Motor1 and send selection signal Motor1Selection to Slave Motor1 by frame Motor1Control.

- If value of temperature signal is higher than MOTOR1_OVER_TEMP value, Master SeatECU will send STOP command through Motor1Selection signal to stop motor.

- If value of temperature signal is in range from MOTOR1_MAX_TEMP value to MOTOR1_OVER_TEMP value, master SeatECU will send DECREASE MOTOR SPEED command through Motor1Selection signal to reduce motor speed.

- If value of temperature signal is lower than MOTOR1_MAX_TEMP value, master will send INCREASE MO←TOR SPEED command through Motor1Selection signal to increase motor speed.

- When users press button SW2 on the Master board, the Master SeatECU switches its schedule table to go-to-sleep table. So the Slave and Master enter sleep mode, RGB LEDS are off.

- When LIN cluster is in sleep mode, users press button SW3 on the Master board, the Master board sends a wakeup signal to wakeup slave nodes, then switches its table to NormalTable.

Note

This example will work only in FLASH target.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V

- 5 Dupont female to female cable

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K1xxCVD-Q48 with S32K116 chip

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K1xxCVD-Q48
BUTTON 2 (**PTD5**)	SW3 - wired on the board	BTN3 - wired on the board
BUTTON 1 (**PTD3**)	SW2 - wired on the board	BTN2 - wired on the board
RED_LED (**PTD16**)	RGB_RED - wired on the board	J12.15 - J11.31
GREEN_LED (**PTD15**)	RGB_GREEN - wired on the board	J12.18 - J11.30
BLUE_LED (**PTE8**)	RGB_BLUE - wired on the board	J13.23 - J11.29
GND (**GND**)	J11-4 - Master GND	J6 - Master GND
LIN (∗)	J11-1 - Master LIN	J48.4 - Master LIN

(∗) **Those lines must be modulated using a transceiver, if it is not specified the boards already include the LIN transceiver**

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **lin_slave_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**lin_slave_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
lin_slave_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
lin_slave_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> This example isn't successful run on RAM because RAM size of S32K116 is small. For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.2 Driver Examples

Applications that show the user how to initialize the peripherals for the basic use cases

There are currently examples for the following categories:
Click on one of the categories to see the available projects

- Analog Driver Examples

- Communication Driver Examples

- System Driver Examples

- Timer Driver Examples

11.3.2.1 Analog Driver Examples

Applications that show the user how to initialize the analog peripherals

There are currently driver examples with the following modules:
Click on one of the module to see the available projects

- ADC Hardware Trigger

- ADC Software Trigger

- CMP DAC

- ADC PAL example

11.3.3 ADC Hardware Trigger

How to trigger the ADC by hardware

Application description

The purpose of this demo application is to show you the usage of the ADC module triggered in hardware by the Programmable Delay Block from the S32K116 CPU using the S32 SDK API.

- The application uses PDB to trigger ADC conversions every 1s.

- When the conversion is complete the data is sent to the host PC using LPUART.

See also

PDB_Example_group
For alternate ADC Hardware triggering scheme see ADC_LOW_POWER_group

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

- UART to USB converter if it is not included on the target board. (Please consult your boards documentation to check if UART-USB converter is present).

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048
LPUART0 TX (**PTB1**)	UART_TX - wired on the board
LPUART0 RX (**PTB0**)	UART_RX - wired on the board
ADC0 Input 0 (**PTA0**)	POT - solder R670 on the board

| Remove R682 if using PTB13 (Note: ADC0_SE8 not present on S32K116)

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **adc_hwtrigger_s32k116**.
Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**adc_hwtrigger_s32k116**). Then go to
Project and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding
to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
adc_hwtrigger_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
adc_hwtrigger_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

For more detailed information related to S32 Design Studio usage please consult the available documentation.

Notes

ADC0_SE8 not present on S32K116. For this example it is necessary to open a terminal emulator and configure it
with:

- 115200 baud

- One stop bit

- No parity

- No flow control

11.3.4   ADC Software Trigger

How to trigger ADC by software

Application description

The purpose of this demo application is to show you the usage of the ADC module triggered by software from the S32K116 CPU using the S32 SDK API.

- The application uses software to trigger ADC conversions every 1s.

- When the conversion is complete the data is sent to the host PC using LPUART.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

- UART to USB converter if it is not included on the target board. (Please consult your boards documentation to check if UART-USB converter is present).

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048
LPUART0 TX (**PTB1**)	UART_TX - wired on the board
LPUART0 RX (**PTB0**)	UART_RX - wired on the board
ADC0 Input 0 (**PTA0**)	POT - solder R670 on the board

| Remove R682 if using PTB13 (Note: ADC0_SE8 not present on S32K116)

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **adc_swtrigger_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**adc_swtrigger_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
adc_swtrigger_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
adc_swtrigger_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

Notes

ADC0_SE8 not present on S32K116. For this example it is necessary to open a terminal emulator and configure it with:

- 115200 baud

- One stop bit

- No parity

- No flow control

11.3.5   CMP DAC

Driver examples showing the basic usage scenario of the CMP

Application description

The purpose of this demo application is to show you how to use the Analog Comparator of the S32K116 MCU using the S32 SDK API.
The Comparator is configured to compare analog input 0(AIN0) with half the reference voltage generated with the internal DAC. Based on the input from the potentiometer the LEDs light by the following rules:

- 1) Vin $<$ DAC voltage : RED on, GREEN off

- 2) Vin $>$ DAC voltage : RED off, GREEN on

- 3) Unknown state : RED on, GREEN on

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048
RED_LED (**PTD15**)	RGB_RED - wired on the board
GREEN_LED (**PTD16**)	RGB_GREEN - wired on the board

CMP Input 0 (**PTA0**) Connect R670 resistor

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **cmp_dac_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**cmp_dac_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) or **RAM** (Debug_RAM) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
cmp_dac_s32k116_debug_ram_jlink	Debug the RAM configuration using Segger Jlink debuggers

cmp_dac_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
cmp_dac_s32k116_debug_ram_pemicro	Debug the RAM configuration using PEMicro debuggers
cmp_dac_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.

Use the controls to control the program flow.

Note

For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.6 ADC PAL example

Example for ADC PAL usage

Application description

The purpose of this demo application is to present the basic functionality of the Analog to Digital Converter Peripheral Abstraction Layer (ADC PAL) on S32K14x MCU.

The application uses ADC PAL to trigger multiple executions of two groups of ADC conversions: first group configured for SW triggering and second group for HW triggering. For each execution of a group of conversions, an average conversion value is computed in SW, and the average value is printed on UART.
example is divided in 2 parts:

- Part 1: SW triggered group of conversions
  After each complete execution of the group, results are read, the average value is calculated and printed to console. A delay is inserted and then the SW group is triggered again. The process is repeated for a fixed number of iterations.

- Part 2: HW triggered group of conversions
  LPTMR is configured to provide a trigger event with a fixed periodicity. The selected HW group is enabled. After each complete execution of the group, results are read, the average value is calculated and printed to console. After a fixed number of iterations, the HW trigger group of conversions is disabled, and the LPTMR is stopped.

Note: both HW and SW triggered groups are configured to run all conversions on a single ADC InputChannel, because the development board contains only a single potentiometer connected to the MCU. However, the ADC PAL supports different InputChannels to be used in the same group. For more details please refer to the ADC PAL documentation.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

- UART to USB converter if it is not included on the target board. (Please consult your boards documentation to check if UART-USB converter is present).

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048
LPUART0 TX (**PTB1**)	UART_TX - wired on the board
LPUART0 RX (**PTB0**)	UART_RX - wired on the board
ADC0 Input 0 (**PTA0**)	POT - R670 needs to be soldered and

| PTB13 disabled or R682 removed

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **adc_pal_example_↩ s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**adc_pal_example_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be two debug configurations for this project:

Configuration Name	Description
adc_pal_example_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
adc_pal_example_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> The Debug_RAM configuration could not be accommodated for the current example, due to the small RAM footprint present on S32K116.
> For more detailed information related to S32 Design Studio usage please consult the available documentation.

Notes

For this example it is necessary to open a terminal emulator and configure it with:

- 115200 baud

- One stop bit

- No parity

- No flow control

11.3.6.1 Communication Driver Examples

Applications that show the user how to initialize the communication peripherals

There are currently driver examples with the following modules:
Click on one of the module to see the available projects

- CAN PAL

- FLEXIO I2C

- FLEXIO I2S

- FLEXIO SPI

- FLEXIO UART

- I2C PAL

- I2S PAL

- LPI2C MASTER

- LPI2C SLAVE

- LPSPI DMA Master

- LPSPI DMA Slave

- LPSPI Transfer Master

- LPSPI Transfer Slave

- LPUART Echo

- SPI PAL

- UART PAL ECHO

11.3.7 CAN PAL

Demo application showing the CAN PAL functionalities

Application description

The purpose of this demo application is to show you the usage of the CAN PAL module configured to use Flexible Data Rate from the S32K116 CPU using the S32 SDK API.

- In the first part, the application will setup the board clocks, pins and other system functions such as SBC if the board uses this module as a CAN transceiver.

- Then it will configure the CAN PAL module features such as FD, Bitrate and buffers

- The application will wait for frames to be received on the configured buffer or for an event raised by pressing one of the two buttons which will trigger a frame send to the recipient.

- Pressing SW3 button of board 1 shall trigger a CAN transfer that results in toggling the RED led on board 2.

- Pressing SW2 button of board 1 shall trigger a CAN transfer that results in toggling the GREEN led on board 2.

- This demo application requires two boards, one configured as master and the other one configured as slave (see MASTER/SLAVE defines in application code).

Note

Due to limited RAM size, this example contains only one build configuration for target flash memory (and the associated debug configuration).

Prerequisites

To run the example you will need to have the following items:

- 2 S32K116 boards

- 1 Power Adapter 12V

- 3 Dupont female to famale cables

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q100

CAN HIGH (∗)	CAN HIGH - J13.1
CAN LOW (∗)	CAN LOW - J13.2
GND (**GND**)	GND - J13.4
BUTTON 0 (**PTC13**)	SW3 - wired on the board
BUTTON 1 (**PTC12**)	SW2 - wired on the board
RED_LED (**PTD16**)	RGB_RED - wired on the board
GREEN_LED (**PTD15**)	RGB_GREEN - wired on the board

(∗) Those lines must be modulated using a transceiver, if it is not specified the boards already include the CAN transceiver

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **can_pal_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**can_pal_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
can_pal_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
can_pal_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.8   FLEXIO I2C

Example application showing FlexIO I2C driver usage

Application description

The purpose of this demo application is to show you the usage of the FlexIO I2C driver found on the S32K116 SoC using S32 SDK API.

The application uses FlexIO I2C driver to make a send and a receive data request. The slave device for this example is the LPI2C instance, which is configured to act as a bus slave. The slave and master buffers will be checked after each transfer by the application, RED or GREEN led will be lit depending on the check result.

Prerequisites

To run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board can't be powered from the USB)

- 2 Dupont male to male cable

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048
FLEXIO SDA (**PTD0**)	J1.5 - J1.1
FLEXIO SCL (**PTD1**)	J1.6 - J1.2
LPI2C SDA (**PTA2**)	J1.1 - J1.5
LPI2C SCL (**PTA3**)	J1.2 - J1.6

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **flexio_i2c_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**flexio_i2c_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Debugging the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
flexio_i2c_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers

flexio_i2c_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.9   FLEXIO I2S

Example application showing FlexIO I2S driver usage

Application description

The purpose of this demo application is to show you the usage of the FlexIO I2S driver found on the S32K116 SoC using S32 SDK API.

The application uses FlexIO I2S driver to make a data transfer of a defined size. The slave device for this example is a second FlexIO I2S driver using the same FlexIO instance, which is configured to act as a bus slave. The slave and master buffers will be checked after each transfer by the application, RED or GREEN led will be lit depending on the check result.

The MASTER I2S driver is configured to use DMA for transfers.

Data size is configured by TRANSFER_SIZE define, by default is configured to be 0.5 KB.

Note

> Since the driver is configured to tranfer 32 bit frames the data size must be modulo 4.

Prerequisites

To run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board can't be powered from the USB)

- 4 Dupont male to male cable

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048
FLEXIO_MASTER TX (**PTD0**)	J1.5 - J1.7
FLEXIO_MASTER RX (**PTD1**)	J1.6 - J1.2
FLEXIO_MASTER SCK (**PTA0**)	J2.9 - J3.10
FLEXIO_MASTER WS (**PTA1**)	J2.10 - J3.12
FLEXIO_SLAVE TX (**PTA3**)	J1.2 - J1.6
FLEXIO_SLAVE RX (**PTD2**)	J1.7 - J1.5
FLEXIO_SLAVE SCK (**PTE4**)	J3.10 - J2.9
FLEXIO_SLAVE WS (**PTE5**)	J3.12 - J2.10

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **flexio_i2s_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**flexio_i2s_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Debugging the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
flexio_i2s_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
flexio_i2s_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.10 FLEXIO SPI

Example application showing FlexIO SPI driver usage

Application description

The purpose of this demo application is to show you the usage of the FlexIO SPI driver found on the S32K116 SoC using S32 SDK API.

The application uses FlexIO SPI driver to make a data transfer of a defined size. The slave device for this example is a second FlexIO SPI driver using the same FlexIO instance, which is configured to act as a bus slave. The slave

and master buffers will be checked after each transfer by the application, RED or GREEN led will be lit depending on the check result.

Prerequisites

To run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board can't be powered from the USB)

- 4 Dupont male to male cable

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048
FLEXIO_MASTER MOSI (**PTD0**)	J1.5 - J1.7
FLEXIO_MASTER MISO (**PTD1**)	J1.6 - J1.2
FLEXIO_MASTER SCK (**PTA0**)	J2.9 - J3.10
FLEXIO_MASTER SS (**PTA1**)	J2.10 - J3.12
FLEXIO_SLAVE MOSI (**PTD2**)	J1.7 - J1.5
FLEXIO_SLAVE MISO (**PTA3**)	J1.2 - J1.6
FLEXIO_SLAVE SCK (**PTE4**)	J3.10 - J2.9
FLEXIO_SLAVE SS (**PTE5**)	J3.12 - J2.10

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **flexio_spi_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**flexio_spi_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Debugging the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
flexio_spi_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
flexio_spi_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.11 FLEXIO UART

Example application showing FlexIO UART driver usage

Application description

The purpose of this demo application is to show you the usage of the FlexIO UART driver found on the S32K116 SoC using S32 SDK API.

Two instances of the FlexIO UART driver are used to echo the data received from host.

Prerequisites

To run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board can't be powered from the USB)

- 2 Dupont male to male cable

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

- UART to USB converter if it is not included on the target board. (Please consult your boards documentation to check if UART-USB converter is present).

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048
FLEXIO_UART RX (**PTD1**)	J1.6 - J3.2

| FLEXIO_UART TX (**PTA0**) | J2.9 - J3.4 |

Note

>   The application uses on board USB - UART chips to transfer data from board to host PC

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **flexio_uart_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**flexio_uart_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Debugging the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
flexio_uart_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
flexio_uart_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

>   For more detailed information related to S32 Design Studio usage please consult the available documentation.

Notes

For this example it is necessary to open a terminal emulator and configure it with:

- 115200 baud

- One stop bit

- No parity

- No flow control

11.3.12   I2C PAL

Driver example using I2C

Application description

The purpose of this application is to show you how to use the LPI2C and FLEXIO Interfaces on the S32K116 using the S32 SDK API.

The application uses one board instance of LPI2C in slave configuration and one board instance of FLEXIO in master configuration to communicate data via the I2C bus using interrupts. If transfer succeeded, which means that slave rx buffer have the same elements as master tx buffer, the RGB_GREEN led wired on the board will be on, otherwise if the transfer failed the RGB_RED led will be on.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board can't be powered from the USB)

- 1 Personal Computer

- 4 Dupont male to male cable

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048
FLEXIO SDA (**PTD0**)	J1.5 - J1.1
FLEXIO SCL (**PTD1**)	J1.6 - J1.2
LPI2C SDA (**PTA2**)	J1.1 - J1.5
LPI2C SCL (**PTA3**)	J1.2 - J1.6

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **i2c_pal**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**i2c_pal**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
i2c_pal_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
i2c_pal_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.13   I2S PAL

Driver example using I2S

Application description

The purpose of this application is to show you how to use the I2S_PAL on the S32K116.

The application uses one instance of FLEXIO in slave configuration and one instance of FLEXIO in master configuration to communicate data via the I2S bus using interrupts.

Success or failed status when example ends will be stored in CheckResult variable.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Personal Computer

- 3 male to male jump wires

Boards supported

The following boards are supported by this application:

- S32K116-EVB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116-EVB
SCK	j1.5 - j1.7
WS	j1.6 - j3.16
TX - RX	j2.9 - j3.12

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **i2s_pal**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**i2s_pal**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
i2s_pal_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
i2s_pal_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.14 LPI2C MASTER

Driver example that will show the LPI2C Master functionality

Application description

The purpose of this demo application is to show you the usage of the LPI2C module available on the S32K116 MCU as a **master** using S32 SDK.

- The application uses S32 SDK API to initialize the LPI2C module as a master node and in Fast operation speed after configuring the clocks and pins needed to use the I2C. The example sends to requests to a slave, found at the configured address, the first being a TX request, while the other being a RX request.

To check if the transmission is successful the user has to verify that the master rx buffer has the same elements as slave tx buffer and slave rx buffer has the same elements as master tx buffer after running the example.

Prerequisites

To run the example you will need to have the following items:

- 1 S32K116 board

- 3 Dupont cables (male to male or female to female depending on the boards)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048
LPI2C SCL (**PTA3**)	J1-2 - Slave SCL
LPI2C SDA (**PTA2**)	J1-1 - Slave SDA

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **lpi2c_master_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**lpi2c_master_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
lpi2c_master_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
lpi2c_master_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.15   LPI2C SLAVE

Driver example that will show the LPI2C Slave functionality

Application description

The purpose of this demo application is to show you the usage of the LPI2C module available on the S32K116 MCU as a **slave** using S32 SDK.

- The application uses S32 SDK API to initialize the LPI2C module as a slave node and in Fast operation speed after configuring the clocks and pins needed to use the I2C. example uses the LPI2C callback to respond to requests such as:

    - data receive

    - data transmit

    - buffer full or empty.

To check if the transmission is successful the user has to verify that the master rx buffer has the same elements as slave tx buffer and slave rx buffer has the same elements as master tx buffer after running the example.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 2 Dupont cables (male to male or female to female depending on the boards)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048
LPI2C SCL (**PTA3**)	J1-2 - Master SCL
LPI2C SDA (**PTA2**)	J1-1 - Master SDA

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **lpi2c_slave_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**lpi2c_slave_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
lpi2c_slave_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
lpi2c_slave_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.16   LPSPI DMA Master

Driver example that will show the LPSPI Master functionalities

Application description

The purpose of this application is to show the user how to use the Low Power Serial Peripheral Interface on the S32K116 using the S32 SDK API.

- The application uses the on board instance of LPSPI in master configuration to communicate data via the SPI bus using DMA.
  The example sends a buffer of 16 elements as a master to a slave on the SPI Bus using DMA.

- To check if the transmission is successful the user has to verify that the masterDataSend buffer has the same elements as slaveDataReceive buffer and slaveDataSend buffer has the same elements as masterData↩
  Receive buffer after running the example.

Note

> The Slave device should be listening for data coming from the Master before the Master device starts sending the data.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 4 Dupont male to male cables

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q100
LPSPI0 CS (**PTB5**)	J2.3 - Slave CS
LPSPI0 SCK (**PTB2**)	J2.6 - Slave SCK
LPSPI0 MOSI (**PTB4**)	J2.4 - Slave MISO
LPSPI0 MISO (**PTB3**)	J2.5 - Slave MOSI

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **lpspi_dma_master_↩ s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**lpspi_dma_master_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be two debug configurations for this project:

Configuration Name	Description
lpspi_dma_master_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
lpspi_dma_master_s32k116_debug_flash_↩ pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.17   LPSPI DMA Slave

Driver example that will show the LPSPI Slave functionalities

Application description

The purpose of this application is to show the user how to use the Low Power Serial Peripheral Interface on the S32K116 using the S32 SDK API.

- The application uses the on board instance of LPSPI in slave configuration to communicate data via the SPI bus using DMA.
  The example sends a buffer of 16 elements as a slave to the master of the SPI Bus using DMA.

- To check if the transmission is successful the user has to verify that the masterDataSend buffer has the same elements as slaveDataReceive buffer and slaveDataSend buffer has the same elements as masterData↩ Receive buffer after running the example.

Note

> The Slave device should be listening for data coming from the Master before the Master device starts sending the data.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 4 Dupont male to male cables

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q100
LPSPI0 CS (**PTB5**)	J2.3 - Master CS
LPSPI0 SCK (**PTB2**)	J2.6 - Master SCK
LPSPI0 MOSI (**PTB4**)	J2.4 - Master MISO
LPSPI0 MISO (**PTB3**)	J2.5 - Master MOSI

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **lpspi_dma_slave_↩ s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**lpspi_dma_slave_s32k116**). Then go
to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding
to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be two debug configurations for this project:

Configuration Name	Description
lpspi_dma_slave_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
lpspi_dma_slave_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Per-
spective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.18   LPSPI Transfer Master

Driver example that will show the LPSPI Master functionalities

Application description

The purpose of this application is to show the user how to use the Low Power Serial Peripheral Interface on the
S32K116 using the S32 SDK API.

- The application uses the on board instance of LPSPI in master configuration to communicate data via the
  SPI bus.
  The example sends a buffer of 16 elements as a master to a slave on the SPI Bus.

- To check if the transmission is successful the user has to verify that the masterDataSend buffer has the same
  elements as slaveDataReceive buffer and slaveDataSend buffer has the same elements as masterData↩
  Receive buffer after running the example.

Note

> The Slave device should be listening for data coming from the Master before the Master device starts sending
> the data.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 4 Dupont male to male cables

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q100
LPSPI0 CS (**PTB5**)	J2.3 - Slave CS
LPSPI0 SCK (**PTB2**)	J2.6 - Slave SCK
LPSPI0 MOSI (**PTB4**)	J2.4 - Slave MISO
LPSPI0 MISO (**PTB3**)	J2.5 - Slave MOSI

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **lpspi_transfer_master↩ _s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**lpspi_transfer_master_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be two debug configurations for this project:

Configuration Name	Description
lpspi_transfer_master_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
lpspi_transfer_master_s32k116_debug_flash_↩ pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.19 LPSPI Transfer Slave

Driver example that will show the LPSPI Slave functionalities

Application description

The purpose of this application is to show the user how to use the Low Power Serial Peripheral Interface on the S32K116 using the S32 SDK API.

- The application uses the on board instance of LPSPI in slave configuration to communicate data via the SPI bus.
  The example sends a buffer of 16 elements as a slave to the master of the SPI Bus.

- To check if the transmission is successful the user has to verify that the masterDataSend buffer has the same elements as slaveDataReceive buffer and slaveDataSend buffer has the same elements as masterData↩ Receive buffer after running the example.

Note

The Slave device should be listening for data coming from the Master before the Master device starts sending the data.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 4 Dupont male to male cables

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q100
LPSPI0 CS (**PTB5**)	J2.3 - Master CS
LPSPI0 SCK (**PTB2**)	J2.6 - Master SCK
LPSPI0 MOSI (**PTB4**)	J2.4 - Master MISO
LPSPI0 MISO (**PTB3**)	J2.5 - Master MOSI

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **lpspi_transfer_slave_↩ s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**lpspi_transfer_slave_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be two debug configurations for this project:

Configuration Name	Description
lpspi_transfer_slave_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
lpspi_transfer_slave_s32k116_debug_flash_↩ pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.20 LPUART Echo

Example application using the LPUART driver

Application description

The purpose of this demo application is to show you how to use the Low Power UART from the S32K116 CPU using the S32 SDK API. The example will echo back any string that is being sent from the console; if 'Hello Board' is sent, the MCU will reply with 'Hello World!!' string.

Note

> Due to limited RAM size, this example contains only one build configuration for target flash memory (and the associated debug configuration).

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q100
LPUART0 TX (**PTB1**)	UART_TX - wired on the board
LPUART0 RX (**PTB0**)	UART_RX - wired on the board

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **lpuart_echo_s32k116**.
Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**lpuart_echo_s32k116**). Then go to
Project and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Click the **build** button(and wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be one debug configuration for this project:

Configuration Name	Description
lpuart_echo_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

Notes

For this example it is necessary to open a terminal emulator and configure it with:

- 115200 baud

- One stop bit

- No parity

- No flow control

- '\n' line ending

11.3.21   SPI PAL

Driver example using SPI

Application description

The purpose of this application is to show you how to use the LPSPI and FLEXIO Interfaces on the S32K116 using the S32 SDK API.

The application uses one board instance of LPSPI in slave configuration and one board instance of FLEXIO in master configuration to communicate data via the SPI bus using interrupts. It also verifies that the data sent is the same as the received data. If the transfer is successful the green LED is on.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board can't be powered from the USB)

- 1 Personal Computer

- 4 Dupont male to male cable

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K116-MB
FLEXIO_MASTER CS (**PTE5**)	J3.12 - J2.3	PTE5 - PTB5
FLEXIO_MASTER SCK (**PTE4**)	J3.10 - J2.6	PTE4 - PTB2
FLEXIO_MASTER MOSI (**PTD1**)	J1.6 - J2.5	PTD1 - PTB3
FLEXIO_MASTER MISO (**PTD0**)	J1.5 - J2.4	PTD0 - PTB4
LPSPI_SLAVE SS (**PTB5**)	J3.12 - J2.3	PTE5 - PTB5
LPSPI_SLAVE SCK (**PTB2**)	J3.10 - J2.6	PTE4 - PTB2
LPSPI_SLAVE MOSI (**PTB3**)	J1.6 - J2.5	PTD1 - PTB3
LPSPI_SLAVE MISO (**PTB4**)	J1.5 - J2.4	PTD0 - PTB4

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **spi_pal**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**spi_pal**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
spi_pal_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
spi_pal_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.22   UART PAL ECHO

Basic application that presents the project scenarios for S32 SDK

Application description

The purpose of this demo is to show the user how UART PAL works over FLEXIO_UART or LPUART peripherals. The user can choose whether to use FLEXIO_UART or LPUART (see USE_FLEXIO_UART define from example code). If 'Hello!' string is sent from the console, the board will reply with 'Hello World!'; when any other string is sent from the console, it will be echoed back.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

- UART to USB converter if it is not included on the target board. (Please consult your boards documentation to check if UART-USB converter is present).

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K1xxCVD-Q48 + S32K-MB(or S32K144-MB)

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K-MB or S32K144-MB
RED_LED (**PTD16**)	RGB_RED - wired on the board	
LED1 (**PTC1**)		JP50 - jump 50 on motherboard
LPUART0 TX (**PTB1**)	UART_TX - wired on the board	J10.32 - J20.2
LPUART0 RX (**PTB0**)	UART_RX - wired on the board	J10.31 - J20.5
FLEXIO_UART RX (**PTA1**)	J2.10 - J3.2	J9.32 - J20.5
FLEXIO_UART TX (**PTA0**)	J2.9 - J3.4	J9.31 - J20.2

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **uart_pal_echo**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**uart_pal_echo**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
uart_pal_echo_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
uart_pal_echo_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

Notes

For this example it is necessary to open a terminal emulator and configure it with:

- 115200 baud

- One stop bit

- No parity

- No flow control

- '\n' line ending

11.3.22.1 System Driver Examples

Applications that show the user how to initialize the communication peripherals

There are currently driver examples with the following modules:
Click on one of the module to see the available projects

- EIM

- WDG PAL Interrupt

- CRC Checksum

- CSEc key configuration

- CSEc key configuration

- MPU Memory Protection

- MPU PAL Memory Protection

- erm_report_s32k116_group

- WDOG Interrupt

- SECURITY PAL

- Power Mode Switch

- FLASH Partitioning

- Trigger MUX Control

11.3.23 EIM

Basic application that presents the project scenarios for S32 SDK

Application description

The purpose of this demo is to provide the user check able correction of ECC. Module EIM enable user addition error to RAM (up). And enable user can use module ERM to read address that user already error to region RAM. This example only debug on Flash.

Run the code

1. After reset LED_RED on, LED_GREEN off and initialize value for address used to test .

2. Press button SW2 to initialize ERM and EIM module .

3. Read address which initialized if value in address test same before value initialized -> LED_RED OFF and LED_GREEN ON .

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q48

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q48	S32K116-MB
RED_LED (**PTD16**)	RGB_RED - wired on the board	LED1 - wired on the board
GREEN_LED (**PTD15**)	RGB_GREEN - wired on the board	LED2 - wired on the board
SW (**PTD3**)	SW2_BTN0 - wired on the board	BUTTON0 - wired on the board

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **eim_injection_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**eim_injection_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) or **RAM** (Debug_RAM) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
eim_injection_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
eim_injection_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.24   WDG PAL Interrupt

Example application that will show the usage of the Watchdog

Application description

The purpose of this driver application is to show the user how to use the WDG PAL from the S32K116 using the S32 SDK API.

The example uses the SysTick timer from the ARM core to refresh the WDG PAL counter for 30 times. LED0 will toggle when WDG PAL counter is refreshed. After this the WDG PAL counter will expire, WDG PAL interrupt will happen and turn off LED0, LED1. Then the CPU will be reset. If the FLASH configuration will be used, then the program will use the Reset Control Module to detect if the reset was caused by the Watchdog and will stop the execution of the program and turn on LED0, LED1.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K116-MB

| LED0 (**PTD16**) | RGB_RED - wired on the boards | J12.17 - J11.31 |
| LED1 (**PTD15**) | RGB_GREEN - wired on the board | J12.16 - J11.30 |

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **wdg_pal_interrupt_↩ s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**wdg_pal_interrupt_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) or **RAM** (Debug_RAM) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
wdg_pal_interrupt_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
wdg_pal_interrupt_s32k116_debug_flash_↩ pemicro	Debug the FLASH configuration using PEMicro debuggers

Selg_palt the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> This example isn't successful run on RAM because RAM size of S32K116 is small. For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.25 CRC Checksum

Example application showing the usage of the CRC module

Application description

The purpose of this demo application is to show you how to use the Cyclic Redundancy Check of the S32K116 MCU with this SDK.

In this example, The CRC is configured to generate two configurations for CCITT standard following:

- CCITT 16 bits standard: { .crcWidth = CRC_BITS_16, .seed = 0xFFFFU, .polynomial = 0x1021U, .write↩ Transpose = CRC_TRANSPOSE_NONE, .readTranspose = CRC_TRANSPOSE_NONE, .complement↩ Checksum = false }

Run the example:

1. After reset, GREEN LED and RED LED is off.

2. Initialize CCITT 16 bits standard follow configuration above then RED LED is on.

3. Press SW to CRC calculator with CRC_data array in input_data file: If result calculator is correct then RED LED is off and GREEN LED is on and The program will stopped.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K116-MB
RED_LED (**PTD16**)	RGB_RED - wired on the board	LED0 - wired on the board
GREEN_LED (**PTD15**)	RGB_GREEN - wired on the board	LED1 - wired on the board
SW (**PTD3**)	SW2_BTN0 - wired on the board	BUTTON0 - wired on the board

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **crc_checksum_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**crc_checksum_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) or **RAM** (Debug_RAM) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
crc_checksum_s32k116_debug_ram_jlink	Debug the RAM configuration using Segger Jlink debuggers
crc_checksum_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
crc_checksum_s32k116_debug_ram_pemicro	Debug the RAM configuration using PEMicro debuggers
crc_checksum_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

Notes

The CRC module in S32K platform supports both big endian and little endian in source data.

11.3.26 CSEc key configuration

Basic application that presents Flash partitioning for CSEc usage

Note

> **This example works only for CSEc enabled parts.** SIM_SDID indicates whether CSEc is available on your device.
> **This example should only be ran from RAM.**

Application description

The purpose of this demo application is to show the user how to enable the Cryptographic Services Engine module from the S32K116 MCU with the S32 SDK API.

The implementation demonstrates the following:

- the enablement of the CSEc module, by showing how the Flash should be partitioned (using the Flash driver);

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board
- 1 Power Adapter 12V (if the board cannot be powered from the USB port)
- 1 Personal Computer
- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048
- S32K116-MB

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **csec_flash_part_s32k116**.
Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**csec_flash_part_s32k116**). Then go
to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) or **RAM** (Debug_RAM) by left clicking on the downward
arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next
step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
csec_flash_part_s32k116_debug_ram_jlink	Debug the RAM configuration using Segger Jlink debuggers
csec_flash_part_s32k116_debug_ram_pemicro	Debug the RAM configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Per-
spective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.27   CSEc key configuration

Basic application that presents basic usecases for the CSEc driver

Note

> **This example works only for CSEc enabled parts.** SIM_SDID indicates whether CSEc is available on your
> device.
> **Prior to running this example, the Flash must be enabled for CSEc operation. The can be done by
> running the csec_flash_part example.**
> **The user keys are non-volatile.** Once the key was loaded, in order to update it, the counter should be
> increased.
> **After the user key was loaded using this example, any further full erase of the Flash requires a
> Challenge-Authentication process.** This can be done by setting the ERASE_ALL_KEYS macro to 1.

Application description

The purpose of this demo application is to show the user how to use the Cryptographic Services Engine module
from the S32K116 MCU with the S32 SDK API.

The implementation demonstrates the following:

- configuring the MASTER_ECU key;

- configuring the first user key, using the MASTER_ECU key as an authorization;

- using the user key for an encryption. In order to update the user key after they were configured using the example the user should increase the counter used for loading the key. Erasing all the configured keys (including the MASTER_ECU key) can be done by changing the value of the ERASE_ALL_KEYS macro to 1. This will place the part back into factory status (the partition command will need to be run again).

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K116-MB

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **csec_keyconfig_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**csec_keyconfig_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) or **RAM** (Debug_RAM) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
csec_keyconfig_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers

csec_keyconfig_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.28   MPU Memory Protection

Example application that shows how to use the MPU module

Application description

The purpose of this demo application is to show you how to use the Memory Protection Unit of the S32K116 MCU with this SDK.

In this example, MPU regions are configured to have access rights as following:

Region	Core	Debugger	DMA	Address
1	rwx	rwx	rwx	0x00000000 - 0x0001FEFF

0 | 2 | -wx | rwx | rwx |0x0001FF00 - 0x0001FF1F | 3 | r– | rwx | rwx |0x0001FF00 - 0x0001FF1F | 4 | rwx | rwx | rwx |0x0001FF20 - 0xFFFFFFFF

Run the example

1. After reset, MPU will be initialized according to configuration above.

2. Read flash memory at address 0x0001FF04 is permitted.

3. Press button (SW) on the board to ignore read permission by disabling region 3.

4. Read flash memory at address 0x0001FF04 is violated.

5. MPU report the detail of error access on slave port 0 (Crossbar slave port 0 -> Flash Controller).

Verification

1. GREEN LED on indicate that MPU initialization successful.

2. RED LED on (GREEN LED off) indicate that there is violated read access reported by MPU.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q48

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q48	S32K116-MB
RED_LED (**PTD16**)	RGB_RED - wired on the board	LED1 - wired on the board
GREEN_LED (**PTD15**)	RGB_GREEN - wired on the board	LED2 - wired on the board
SW (**PTD3**)	SW2_BTN0 - wired on the board	BUTTON0 - wired on the board

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **mpu_memory_↩ protection_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**mpu_memory_protection_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) or **RAM** (Debug_RAM) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
mpu_memory_protection_s32k116 Debug_RAM Jlink	Debug the RAM configuration using Segger Jlink debuggers
mpu_memory_protection_s32k116 Debug_FLASH Jlink	Debug the FLASH configuration using Segger Jlink debuggers
mpu_memory_protection_s32k116 Debug_RAM PEmicro	Debug the RAM configuration using PEMicro debuggers
mpu_memory_protection_s32k116 Debug_FLASH PEmicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.29 MPU PAL Memory Protection

Example application that shows how to use the MPU PAL

Application description

The purpose of this demo application is to show you how to use the Memory Protection Unit PAL of the S32K116 MCU with this SDK.

In this example, MPU PAL regions are configured to have access rights as following:

Region	Core	Debugger	DMA	Address
1	rwx	rwx	rwx	0x00000000 - 0x0001FEFF

0 | 2 | -wx | rwx | rwx |0x0001FF00 - 0x0001FF1F | 3 | r-- | rwx | rwx |0x0001FF00 - 0x0001FF1F | 4 | rwx | rwx | rwx |0x0001FF20 - 0xFFFFFFFF

Run the example

1. After reset, MPU PAL will be initialized according to configuration above.

2. Read flash memory at address 0x0001FF04 is permitted.

3. Press button (SW) on the board to ignore read permission by disabling region 3.

4. Read flash memory at address 0x0001FF04 is violated.

5. MPU PAL report the detail of error access on slave port 0 (Crossbar slave port 0 -> Flash Controller).

Verification

1. GREEN LED on indicate that MPU PAL initialization successful.

2. RED LED on (GREEN LED off) indicate that there is violated read access reported by MPU PAL.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q48

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q48	S32K116-MB
RED_LED (**PTD16**)	RGB_RED - wired on the board	LED1 - wired on the board
GREEN_LED (**PTD15**)	RGB_GREEN - wired on the board	LED2 - wired on the board
SW (**PTD3**)	SW2_BTN0 - wired on the board	BUTTON0 - wired on the board

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **mpu_pal_memory_↩ protection_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**mpu_pal_memory_protection_↩ s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) or **RAM** (Debug_RAM) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
mpu_pal_memory_protection_s32k116 Debug_RAM Jlink	Debug the RAM configuration using Segger Jlink debuggers
mpu_pal_memory_protection_s32k116 Debug_FLASH Jlink	Debug the FLASH configuration using Segger Jlink debuggers
mpu_pal_memory_protection_s32k116 Debug_RAM PEMicro	Debug the RAM configuration using PEMicro debuggers
mpu_pal_memory_protection_s32k116 Debug_FLASH PEMicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.30 WDOG Interrupt

Example application that will show the usage of the Watchdog

Application description

The purpose of this driver application is to show the user how to use the WDOG from the S32K116 using the S32 SDK API.

The examples uses the SysTick timer from the ARM core to refresh the WDOG counter for 8 times. After this the Watchdog counter will expire and the CPU will be reset. If the FLASH configuration will be used, then the code will

use the Reset Control Module to detect if the reset was caused by the Watchdog and will stop the execution of the program.

Run the example:

1. After reset, GREEN LED and RED LED is off.

2. Initialize WDOG Interrupt above then RED LED is toggle 8 times(on 4 times and off 4 times).

3. Watchdog timeout happen then MCU reset and GREEN LED and RED LED is on and The program will stopped.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K116-MB
RED_LED (**PTD16**)	RGB_RED - wired on the board	LED0 - wired on the board
GREEN_LED (**PTD15**)	RGB_GREEN - wired on the board	LED1 - wired on the board

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **wdog_interrupt_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**wdog_interrupt_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) or **RAM** (Debug_RAM) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the bulid action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
wdog_interrupt_s32k116_debug_ram_jlink	Debug the RAM configuration using Segger Jlink debuggers
wdog_interrupt_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
wdog_interrupt_s32k116_debug_ram_pemicro	Debug the RAM configuration using PEMicro debuggers
wdog_interrupt_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.31 SECURITY PAL

Basic application that presents basic usecases for the Security PAL.

Note

> **This example works only for CSEc enabled parts.** SIM_SDID indicates whether CSEc is available on your device.
> **Prior to running this example, the Flash must be enabled for CSEc operation. The can be done by running the csec_flash_part example.**
> **This example generates a random number.**
> **This example demonstrates CBC Encryption/Decryption.**

Application description

The purpose of this demo application is to show the user how to use the Security PAL in conjuction with Cryptographic Services Engine module from the S32K1xx MCU with the S32 SDK API.

The implementation demonstrates the following:

- the enablement of the Security PAL, used over CSEc module, by showing how the Flash should be partitioned (using the Flash driver);

- initializing the Random Number Generator and generating a vector of 128 random bits;

- configuring the RAM key, with a 128-bit plaintext;

- using the user key for a CBC encryption and a CBC decryption;

If no errors occur during the cryptographic operations, the green LED will be turned on upon completion; if the red LED is lit, the program failed during one of the steps.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **security_pal**. Select "Copy projects into workspace" and then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**security_pal**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Click the **build** button(and wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug** Configurations. There will be one debug configuration for this project:

Configuration Name	Description
security_pal_s32k116 Debug_FLASH PEMicro	Debug the FLASH configuration using PEMicro debuggers

Select the debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**. Use the controls to control the program flow.

Note

For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.32 Power Mode Switch

Example application demonstrating S32K116 power modes

Application description

The purpose of the application is to show the user how to enter various power modes of the S32K116 SoC using the S32 SDK API.

The application displays on the host PC terminal a menu in which the user can select to enter:

- Normal Run (RUN)

- Very Low Power Run (VLPR)

- STOP mode 1 (STOP1)

- STOP mode 2 (STOP2)

- Very Low Power Stop (VLPS)

When user selects a mode, PC terminal will show result successfully or unsuccessfully. The CPU can be woken up from sleep modes by pressing button (PTD5).

Prerequisites

To run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

- UART to USB converter if it is not included on the target board. (Please consult your boards documentation to check if UART-USB converter is present).

- 2 Dupont female to female cable.

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048
LPUART1 TX (**PTB0**)	UART_RX - wired on the board
LPUART1 RX (**PTB1**)	UART_TX - wired on the board
BUTTON (**PTD5**)	SW3 - wired on the board

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **power_mode_switch_↩s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**power_mode_switch_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) or **RAM** (Debug_RAM) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the bulid action to be completed before continuing to the next step.

4. Building the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
power_mode_switch_s32k116 Debug_FLASH Jlink	Debug the FLASH configuration using Segger Jlink debuggers
power_mode_switch_s32k116 Debug_FLASH PEMicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

For more detailed information related to S32 Design Studio usage please consult the available documentation.

Notes

Due to limited RAM size, this example contains only one build configuration for target flash memory (and the associated debug configuration). For this example it is necessary to open a terminal emulator and configure it with:

- 115200 baud

- One stop bit

- No parity

- No flow control

- '\n' line ending

11.3.33 FLASH Partitioning

Example application which shows the basic operations of the FLASH driver

Application description

The purpose of this demo application is to show you the usage of the FLASH driver with the S32 SDK API.

The examples does the following operations:

- Erases flash

- Partitions the flash

- Configures FlexNVM region as EEPROM

    Note

    The FlexNVM memory is partitioned to EEPROM use and is blocked for some erase commands (Erase Sector and Erase Block). As a consequence, loading the program to flash memory may fail on some debuggers. Please perform a mass erase operation on Flash to remove this partitioning after running the example to be able to update your application on target.

    Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q48

- S32K116-MB

Hardware Wiring

No connections are required for this example.

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **flash_partitioning_↩ s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**flash_partitioning_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) or **RAM** (Debug_RAM) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
flash_partitioning_s32k116_debug_ram_jlink	Debug the RAM configuration using Segger Jlink debuggers
flash_partitioning_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
flash_partitioning_s32k116_debug_ram_pemicro	Debug the RAM configuration using PEMicro debuggers
flash_partitioning_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.34   Trigger MUX Control

Example application showing the usage of the TRGMUX module

Application description

The purpose of this demo application is to show you how to use the Trigger MUX Control of the S32K116 MCU with this SDK.

The examples use TRGMUX to connect Pin Trigger Mux In4 and LPIT channel 0.

- Initialize TRGMUX with source trigger from TRGMUX_IN4 and target module is LPIT_CH0

- Initialize the LPIT Channel 0.

- LED RED on EVB board or LED ORANGE on Motherboard is used to blink led

- Each time when user presses button SW2 on EVB board or SW4 on Motherboard will generate a trigger signal that actives LPIT channel 0 via TRGMUX. After 1ms, LPIT will create an event interrupt and toggle LED

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K116-MB

Hardware Wiring

PIN FUNCTION	S32K116EVB-Q48	S32K116-MB
RED_LED (**PTD16**)	RGB_RED - wired on the board	LED1 - wired on the board
GREEN_LED (**PTD15**)	RGB_GREEN - wired on the board	LED2 - wired on the board
SW (**PTD3**)	SW2_BTN0 - wired on the board	BUTTON0 - wired on the board

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **trgmux_lpit_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**trgmux_lpit_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) or **RAM** (Debug_RAM) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
trgmux_lpit_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
trgmux_lpit_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Per-spective**.
Use the controls to control the program flow.

Note

For more detailed information related to S32 Design Studio usage please consult the available documentation.

Notes

The TRGMUX module in S32K platform supports both big endian and little endian in source data.

11.3.34.1   Timer Driver Examples

Applications that show the user how to initialize the timer peripherals

There are currently driver examples with the following modules:
Click on one of the module to see the available projects

- FTM Combined PWM

- FTM Periodic Interrupt

- FTM PWM

- FTM Signal Measurement

- LPIT Periodic Interrupt

- LPTMR Periodic Interrupt

- LPTMR Pulse Counter

- PDB Periodic Interrupt

- RTC Alarm

- TIMING PAL

- IC PAL Signal Measurement

- OC PAL

- PWM PAL

11.3.35   FTM Combined PWM

Example application showing the FTM's combined PWM functionality

Application description

The purpose of this demo application is to show you the usage of the Combined PWM mode of the FlexTimer module found on the S32K116 using S32 SDK API.

Note

> Due to limited RAM size, this example contains only one build configuration for target flash memory (and the associated debug configuration).

The examples does the following operations:

- increment or decrement duty cycle

- Update channel duty cycle

- Wait for a number of cycles to make the change visible

Run the example

1. After reset, The RED LED and GREEN LED of EVB board will increment or decrement light intensity

2. Use oscilloscope to verify the output signal

Prerequisites

To run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 2 Dupont female to female cable

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K116-MB
FTM0 Channel 0 (**PTD15**)	RGB_GREEN - wired on the board	J12.18 - J11.31
FTM0 Channel 1 (**PTD16**)	RGB_RED - wired on the board	J12.15 - J11.29

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **ftm_combined_pwm_↵ s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**ftm_combined_pwm_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be two debug configurations for this project:

Configuration Name	Description
ftm_combined_pwm_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
ftm_combined_pwm_s32k116_debug_flash_↵ pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.36 FTM Periodic Interrupt

Example application showing the FTM's Timer functionality

Application description

The purpose of this demo application is to show you the usage of the FlexTimer's Timer functionality from the S32K116 CPU using the S32 SDK API.

- The application configures FTM0 to generate an interrupt every 1 second. The interrupt will toggle the configured LED.

Note

> Due to limited RAM size, this example contains only one build configuration for target flash memory (and the associated debug configuration).

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K116-MB
GPIO PIN	(**PTD15**) RGB_GREEN - wired on the board	(**PTC1**) LED1 - wired on the board

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **ftm_periodic_interrupt↩_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**ftm_periodic_interrupt_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button (Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be two debug configurations for this project:

Configuration Name	Description
ftm_periodic_interrupt_s32k116_debug_flash_↩jlink	Debug the FLASH configuration using Segger Jlink debuggers
ftm_periodic_interrupt_s32k116_debug_flash_↩pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.37 FTM PWM

Example application showing the FTM's PWM functionality

Application description

The purpose of this demo application is to show you the usage of the PWM mode of the FlexTimer module found on the S32K116 using S32 SDK API. The examples does the following operations:

- increment or decrement duty cycle

- Update channel duty cycle

- Wait for a number of cycles to make the change visible

Run the example

1. After reset, The GREEN LED of EVB board will increment or decrement light intensity

2. Use oscilloscope to verify the output signal

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Dupont female to female cable

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32k116-MB
FTM0 Channel 0 (**PTD15**)	RGB_GREEN - wired on the board	J12.18 - J11.30

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **ftm_pwm_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**ftm_pwm_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be two debug configurations for this project:

Configuration Name	Description
ftm_pwm_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
ftm_pwm_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

   For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.38   FTM Signal Measurement

Example application showing the FTM's Signal Measurement functionality

Application description

The purpose of this demo application is to show you the usage of the FlexTimer's Signal Measurement functionality from the S32K116 CPU using the S32 SDK API.

  • The application is configured to generate a PWM signal with a variable frequency which will be measured another FTM instance. The frequency will range from 300 Hz to 3000 Hz. Each step changes 100 Hz. The measurement result will be sent to the host PC via LPUART. User is able to compare pwm frequency and measurement frequency.

Note

   Due to limited RAM size, this example contains only one build configuration for target flash memory (and the associated debug configuration).

Prerequisites

The run the example you will need to have the following items:

  • 1 S32K116 board

  • 1 Power Adapter 12V (if the board cannot be powered from the USB port)

  • 1 Dupont male to male

- 3 Dupont female to female

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

- UART to USB converter if it is not included on the target board. (Please consult your boards documentation to check if UART-USB converter is present).

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K116-MB
FTM0 Out Channel 0 (**PTD15**)	J4.12 - J1.4	J12.18 - J9.19
FTM1 Input Channel 6 (**PTA12**)	J4.12 - J1.4	J12.18 - J9.19
LPUART0 TX (**PTB1**)	UART_TX - wired on the board	J10.32 - J20.3
LPUART0 RX (**PTB0**)	UART_RX - wired on the board	J10.31 - J20.6

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **ftm_signal_↩ measurement_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**ftm_signal_measurement_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be two debug configurations for this project:

Configuration Name	Description
ftm_signal_measurement_s32k116_debug_↩ flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
ftm_signal_measurement_s32k116_debug_↩ flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

Notes

For this example it is necessary to open a terminal emulator and configure it with:

- 115200 baud

- One stop bit

- No parity

- No flow control

- '\n' line ending

11.3.39   LPIT Periodic Interrupt

Driver example that will show the LPIT functionality

Application description

The purpose of this demo application is to show you how to use the Low Power Interrupt Timer from the S32K116 using the S32 SDK API.

- The example is configured to trigger an interrupt every second, which toggles a LED.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q48

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q48	S32K116-MB

| GREEN_LED (**PTD15**) | RGB_GREEN - wired on the board | LED1 - J11.32-J12.18 |

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **lpit_periodic_interrupt↩ _s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**lpit_periodic_interrupt_s32k116**).
Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
lpit_periodic_interrupt_s32k116_debug_flash_↩ jlink	Debug the FLASH configuration using Segger Jlink debuggers
lpit_periodic_interrupt_s32k116_debug_flash_↩ pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.40   LPTMR Periodic Interrupt

Example application that shows the LPTMR's Timer feature

Application description

The purpose of this demo application is to show you how to use the LPTMR's Timer functionality from the S32K116 using the S32 SDK API.

- The LPTMR is configured to generate a periodic interrupt at 1 seconds which toggles a LED.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V

- 1 Dupont male to male cable

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q48

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q48	S32K116-MB
GREEN_LED (**PTD15**)	RGB_GREEN - wired on the board	LED1 - J11.32-J12.18

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **lptmr_periodic_↩ interrupt_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**lptmr_periodic_interrupt_s32k116**).
Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
lptmr_periodic_interrupt_s32k116_debug_flash↩ _jlink	Debug the FLASH configuration using Segger Jlink debuggers
lptmr_periodic_interrupt_s32k116_debug_flash↩ _pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

    For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.41 LPTMR Pulse Counter

Example application that shows the LPTMR's Pulse Counting feature

Application description

The purpose of this demo application is to show you how to use the Low Power Timer's Pulse Counter functionality from the S32K144 using the S32 SDK API.

- The example is configured to trigger an interrupt after three pulses, sourced from one of the board's buttons.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V

- 1 Dupont male to male cable

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q48

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q48	S32K116-MB
GREEN_LED (**PTD15**)	RGB_GREEN - wired on the board	LED1 - J11.32-J12.18
BTN1 (**PTD5**)	SW3 - wired on the board	SW6 - J12.28-JP38.1

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **lptmr_pulse_counter_↩ s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**lptmr_pulse_counter_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
lptmr_pulse_counter_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
lptmr_pulse_counter_s32k116_debug_flash_↵ pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.42   PDB Periodic Interrupt

Driver example using PDB

Application description

The purpose of this demo application is to show you how to use the Programmable Delay Block from the S32K116 using the S32 SDK API.

The PDB is configured to generate a periodic interrupt which toggles a LED.

See also

> adc_hwtrigger_group

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K116-MB
GPIO Pin (**PTD16**)	RGB_RED - wired on the board	J12.17 - J11.31

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **pdb_periodic_interrupt↩_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**pdb_periodic_interrupt_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) or **RAM** (Debug_RAM) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
pdb_periodic_interrupt_s32k116_debug_ram_jlink	Debug the RAM configuration using Segger Jlink debuggers
pdb_periodic_interrupt_s32k116_debug_flash_↩jlink	Debug the FLASH configuration using Segger Jlink debuggers
pdb_periodic_interrupt_s32k116_debug_ram_↩pemicro	Debug the RAM configuration using PEMicro debuggers
pdb_periodic_interrupt_s32k116_debug_flash_↩pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

>   For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.43   RTC Alarm

Example application showing basic use cases for the RTC module

Application description

The purpose of this demo application is to show you how to use the Real Time Clock module from the S32K116 MCU with the S32 SDK API.

The RTC is configured to generate an interrupt every 1 second(RED_LED). If the alarm button is pressed an alarm interrupt toggles the alarm led(GREEN_LED) after 5 seconds.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V

- 3 Dupont male to male cable

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application

- S32K116EVB-Q48

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q48
RED_LED (**PTD15**)	RGB_RED - wired on the board
GREEN_LED (**PTD16**)	RGB_GREEN - wired on the board
BUTTON (**PTD3**)	SW2 - wired on the board

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **rtc_alarm_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**rtc_alarm_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
rtc_alarm_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers
rtc_alarm_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

Notes

If the example doesn't work, please Flash the Debug_FLASH configuration and enforce a power on reset of the board. This is caused by the fact that the register which configures the RTC clock source is can only be written once.

11.3.44   TIMING PAL

Driver example using TIMING PAL

Application description

The purpose of this application is to show you how to use the TIMING PAL over LPIT, LPTMR and FTM timers on the S32K116 using the S32 SDK API.

The application uses one board instance of LPIT, LPTMR and FTM to periodically toggle 3 leds.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board can't be powered from the USB)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q48

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q48	S32K116-MB
GREEN_LED (**PTD15**)	RGB_GREEN - wired on the board	LED1 - J11.32-J12.18
RED_LED (**PTD16**)	RGB_RED - wired on the board	LED2 - J11.29-J12.15
BLUE_LED (**PTE8**)	RGB_BLUE - wired on the board	LED3 - J11.30-J13.23

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **timing_pal_s32k116**.
Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**timing_pal_s32k116**). Then go to
Project and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding
to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
timing_pal_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers
timing_pal_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.45 IC PAL Signal Measurement

Example application showing the IC_PAL's Signal Measurement functionality

Application description

The purpose of this demo application is to show you the usage of the IC_PAL's Signal Measurement functionality
from the S32K116 CPU using the S32 SDK API.

- The application is configured to generate a PWM signal with a variable frequency which will be measured
  another FTM instance. The frequency will range from 300 Hz to 3000 Hz. Each step changes 100 Hz. The
  measurement result will be sent to the host PC via LPUART. User is able to compare pwm frequency and
  measurement frequency.

Note

> Due to limited RAM size, this example contains only one build configuration for target flash memory (and the
> associated debug configuration).

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board cannot be powered from the USB port)

- 1 Dupont male to male

- 3 Dupont female to female

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

- UART to USB converter if it is not included on the target board. (Please consult your boards documentation to check if UART-USB converter is present).

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K116-MB
FTM0 Out Channel 0 (**PTD15**)	J4.12 - J2.6	J12.18 - J10.29
FTM1 Input Channel 0 (**PTB2**)	J4.12 - J2.6	J12.18 - J10.29
LPUART0 TX (**PTB1**)	UART_TX - wired on the board	J10.32 - J20.3
LPUART0 RX (**PTB0**)	UART_RX - wired on the board	J10.31 - J20.6

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **ic_pal_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**ic_pal_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be two debug configurations for this project:

Configuration Name	Description
ic_pal_s32k116 Debug_FLASH Jlink	Debug the FLASH configuration using Segger Jlink debuggers
ic_pal_s32k116 Debug_FLASH PEMicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

Notes

For this example it is necessary to open a terminal emulator and configure it with:

- 115200 baud

- One stop bit

- No parity

- No flow control

- '\n' line ending

11.3.46   OC PAL

Driver example using OC PAL

Application description

The purpose of this application is to show you how to use the OC PAL over FTM_OC on the S32K116 using the S32 SDK API.

The application uses one board instance of FTM to periodically toggle 1 led with period 2 second.

Prerequisites

The run the example you will need to have the following items:

- 1 S32K116 board

- 1 Power Adapter 12V (if the board can't be powered from the USB)

- 1 Personal Computer

- 1 Jlink Lite Debugger (optional, users can use Open SDA)

Boards supported

The following boards are supported by this application:

- S32K116EVB-Q048

- S32K116-MB

Hardware Wiring

The following connections must be done to for this example application to work:

PIN FUNCTION	S32K116EVB-Q048	S32K116-MB
FTM0 Channel 0 (**PTD15**)	RGB_GREEN - wired on the board	wired on the board with connect J12.16 - J11.30

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **oc_pal_S32K116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**oc_pal_S32K116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Select the configuration to be built **FLASH** (Debug_FLASH) by left clicking on the downward arrow corresponding to the **build** button(. Wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be four debug configurations for this project:

Configuration Name	Description
oc_pal_S32K116 Debug_FLASH Jlink	Debug the FLASH configuration using Segger Jlink debuggers
oc_pal_S32K116 Debug_FLASH PEMicro	Debug the FLASH configuration using PEMicro debuggers

Select the desired debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**.
Use the controls to control the program flow.

Note

> For more detailed information related to S32 Design Studio usage please consult the available documentation.

11.3.47   PWM PAL

Example application using the PWM PAL

Application description

The purpose of this demo application is to show you how to use the PWM PAL from the S32K116 CPU using the S32 SDK API. The example will increase gradually the intensity of the GREEN LED.

Note

> Due to limited RAM size, this example contains only one build configuration for target flash memory (and the associated debug configuration).

Prerequisites

The run the example you will need to have the following items:

• 1 S32K116 board

• 1 Power Adapter 12V (if the board cannot be powered from the USB port)

• 1 Personal Computer

Boards supported

The following boards are supported by this application:

• S32K116EVB-Q048

How to run

1. Importing the project into the workspace

After opening S32 Design Studio, go to **File** -> **New S32DS Project From**... and select **pwm_pal_s32k116**. Then click on **Finish**.
The project should now be copied into you current workspace.

2. Generating the Processor Expert configuration

First go to **Project Explorer** View in S32 DS and select the current project(**lpuart_echo_s32k116**). Then go to **Project** and click on **Generate Processor Expert Code**
Wait for the code generation to be completed before continuing to the next step.

3. Building the project

Click the **build** button(and wait for the build action to be completed before continuing to the next step.

4. Running the project

Go to **Run** and select **Debug Configurations**. There will be one debug configuration for this project:

Configuration Name	Description
pwm_pal_s32k116_debug_flash_pemicro	Debug the FLASH configuration using PEMicro debuggers
pwm_pal_s32k116_debug_flash_jlink	Debug the FLASH configuration using Segger Jlink debuggers

Select the debug configuration and click on **Launch**. Now the perspective will change to the **Debug Perspective**. Use the controls to control the program flow.

Note

For more detailed information related to S32 Design Studio usage please consult the available documentation.

12   iseled_freemaster

Basic application that presents the project scenarios for S32 SDK

Application description

This example is only a placeholder for the fully featured ISELED Demo application. Please contact your FAE or sales representative for details on how to obtain the ISELED Library.

13 touchsense

Basic application that presents the project scenarios for S32 SDK

Application description

This example is only a placeholder for the fully featured Touch Sense Demo application. Please contact your FAE or sales representative for details on how to obtain the Touch Sense Library.

14 Module Index

14.1 Modules

Here is a list of all modules:

15 Data Structure Index

15.1 Data Structures

Here are the data structures with brief descriptions:

16 Module Documentation

16.1 ADC Driver

16.1.1 Detailed Description

Analog to Digital Converter Peripheral Driver.

The ADC is a configurable 12-bit (selectable to between 8-bit, 10-bit and 12-bit resolution) single-ended SA↩
R converter.

Features of the ADC include:

- up to 32 control channels (depending on the device variant), with configurable triggers

- up to 32 selectable external input sources (depending on the device variant) and multiple internal input sources

- hardware compare and average functions

- auto-calibration feature

Hardware background

The ADC included in the S32K14x series is a selectable resolution (8, 10, 12-bit), single-ended, SAR converter. Depending on the device variant, each ADC instance has up to 40 selectable input channels (up to 32 external and up to 8 internal) and up to 32 control channels (each with a result register, an input channel selection register and interrupt enable).

Sample time is configurable through selection of A/D clock and a configurable sample time (in A/D clocks).

Also provided are the Hardware Average and Hardware Compare Features.

Hardware Average will sample a selectable number of measurements and average them before signaling a Conversion Complete.

Hardware Compare can be used to signal if an input channel goes outside (or inside) of a predefined range.

The **Calibration** features can be used to automatically calibrate or fine-tune the ADC before use.

Driver consideration

The ADC Driver provides access to all features, but not all need to be configured to use the ADC. The user application can use the default for most settings, changing only what is necessary. For example, if Compare or Average features are not used, the user does not need to configure them.

The Driver uses structures for configuration. Each structure contains members that are specific to its respective functionality. There is a **converter** structure, a hardware **compare** structure, a hardware **average** structure and a **calibration** structure. Each struct has a corresponding `InitStruct()` method that can be used to initialize the members to reset values, so the user can change only the values that are specific to the application.

The Driver also includes support for configuring the Trigger Latching and Arbitration Unit controlled from a separate hardware module - System Integration Module (SIM).

Interrupt handling

The ADC Driver in S32 SDK does not use interrupts internally. These can be defined by the user application. There are two ways to add an ADC interrupt:

1. Using the weak symbols defined by start-up code. If the methods ADC**x**_Handler(void) (x denotes instance number) are not defined, the linker uses a default ISR. An error will be generated if methods with the same name are defined multiple times. This method works regardless of the placement of the interrupt vector table (Flash or RAM).

2. Using the Interrupt Manager's INT_SYS_InstallHandler() method. This can be used to dynamically change the ISR at run-time. This method works only if the interrupt vector table is located in RAM (S32 SDK behavior). To get the ADC instance's interrupt number, use ADC_DRV_GetInterruptNumber().

Clocking and pin configuration

The ADC Driver does not handle clock setup (from PCC) or any kind of pin configuration (done by PORT module). This is handled by the Clock Manager and PORT module, respectively. The driver assumes that correct clock configurations have been made, so it is the user's responsibility to set up clocking and pin configurations correctly.

Triggering a conversion

There are two separate ways for triggering an ADC conversion from a control channel:

1. Software triggering Only conversion from first control channel may be triggered from software - must enabled at converter configuration Initiated by writing a valid input channel ID to the first control channel - use ADC↩_DRV_ConfigChan().

2. Hardware triggering Conversion from any control channel may be hardware triggered - however for first control channel it must be enabled at converter configuration.

Data Structures

- struct adc_converter_config_t

    Defines the converter configuration. More...
- struct adc_compare_config_t

    Defines the hardware compare configuration. More...
- struct adc_average_config_t

    Defines the hardware average configuration. More...
- struct adc_chan_config_t

    Defines the control channel configuration. More...
- struct adc_calibration_t

    Defines the user calibration configuration. More...

Enumerations

- enum adc_clk_divide_t { ADC_CLK_DIVIDE_1 = 0x00U, ADC_CLK_DIVIDE_2 = 0x01U, ADC_CLK_DIVI↩DE_4 = 0x02U, ADC_CLK_DIVIDE_8 = 0x03U }

    Clock Divider selection.
- enum adc_resolution_t { ADC_RESOLUTION_8BIT = 0x00U, ADC_RESOLUTION_12BIT = 0x01U, ADC↩_RESOLUTION_10BIT = 0x02U }

    Conversion resolution selection.

- enum [adc_input_clock_t](#) { [ADC_CLK_ALT_1](#) = 0x00U, [ADC_CLK_ALT_2](#) = 0x01U, [ADC_CLK_ALT_3](#) = 0x02U, [ADC_CLK_ALT_4](#) = 0x03U }

    Input clock source selection.
- enum [adc_trigger_t](#) { [ADC_TRIGGER_SOFTWARE](#) = 0x00U, [ADC_TRIGGER_HARDWARE](#) = 0x01U }

    Trigger type selection.
- enum [adc_pretrigger_sel_t](#) { [ADC_PRETRIGGER_SEL_PDB](#) = 0x00U, [ADC_PRETRIGGER_SEL_TRGM↩](#) [UX](#) = 0x01U, [ADC_PRETRIGGER_SEL_SW](#) = 0x02U }

    Pretrigger types selectable from Trigger Latching and Arbitration Unit.
- enum [adc_trigger_sel_t](#) { [ADC_TRIGGER_SEL_PDB](#) = 0x00U, [ADC_TRIGGER_SEL_TRGMUX](#) = 0x01U }

    Trigger source selectable from Trigger Latching and Arbitration Unit.
- enum [adc_sw_pretrigger_t](#) {
    [ADC_SW_PRETRIGGER_DISABLED](#) = 0x00U, [ADC_SW_PRETRIGGER_0](#) = 0x04U, [ADC_SW_PRETR↩](#) [IGGER_1](#) = 0x05U, [ADC_SW_PRETRIGGER_2](#) = 0x06U,
    [ADC_SW_PRETRIGGER_3](#) = 0x07U }

    Software pretriggers which may be set from Trigger Latching and Arbitration Unit.
- enum [adc_voltage_reference_t](#) { [ADC_VOLTAGEREF_VREF](#) = 0x00U, [ADC_VOLTAGEREF_VALT](#) = 0x01U }

    Voltage reference selection.
- enum [adc_average_t](#) { [ADC_AVERAGE_4](#) = 0x00U, [ADC_AVERAGE_8](#) = 0x01U, [ADC_AVERAGE_16](#) = 0x02U, [ADC_AVERAGE_32](#) = 0x03U }

    Hardware average selection.
- enum [adc_inputchannel_t](#) {
    [ADC_INPUTCHAN_EXT0](#) = 0x00U, [ADC_INPUTCHAN_EXT1](#) = 0x01U, [ADC_INPUTCHAN_EXT3](#) = 0x03U,
    [ADC_INPUTCHAN_EXT4](#) = 0x04U,
    [ADC_INPUTCHAN_EXT5](#) = 0x05U, [ADC_INPUTCHAN_EXT6](#) = 0x06U, [ADC_INPUTCHAN_EXT7](#) = 0x07U,
    [ADC_INPUTCHAN_EXT9](#) = 0x09U,
    [ADC_INPUTCHAN_EXT10](#) = 0x0AU, [ADC_INPUTCHAN_EXT11](#) = 0x0BU, [ADC_INPUTCHAN_EXT12](#) = 0x0CU, [ADC_INPUTCHAN_EXT13](#) = 0x0DU,
    [ADC_INPUTCHAN_EXT14](#) = 0x0EU, [ADC_INPUTCHAN_DISABLED](#) = ADC_SC1_ADCH_MASK, [ADC_I↩](#) [NPUTCHAN_INT0](#) = 0x15, [ADC_INPUTCHAN_INT1](#) = 0x16,
    [ADC_INPUTCHAN_INT2](#) = 0x17, [ADC_INPUTCHAN_INT3](#) = 0x1C, [ADC_INPUTCHAN_TEMP](#) = 0x1A, [A↩](#) [DC_INPUTCHAN_BANDGAP](#) = 0x1B,
    [ADC_INPUTCHAN_VREFSH](#) = 0x1D, [ADC_INPUTCHAN_VREFSL](#) = 0x1E, [ADC_INPUTCHAN_SUPPLY↩](#) [_VDD](#) = 0xF00U, [ADC_INPUTCHAN_SUPPLY_VDDA](#) = 0xF01U,
    [ADC_INPUTCHAN_SUPPLY_VREFH](#) = 0xF02U, [ADC_INPUTCHAN_SUPPLY_VDD_3V](#) = 0xF03U, [ADC↩](#) [_INPUTCHAN_SUPPLY_VDD_FLASH_3V](#) = 0xF04U, [ADC_INPUTCHAN_SUPPLY_VDD_LV](#) = 0xF05U }

    Enumeration of input channels assignable to a control channel.
    Note 0: *entries in this enum are affected by ::FEATURE_ADC_NUM_EXT_CHANS, which is device dependent and controlled from "device_name"_features.h file.*
- enum [adc_latch_clear_t](#) { [ADC_LATCH_CLEAR_WAIT](#), [ADC_LATCH_CLEAR_FORCE](#) }

    Defines the trigger latch clear method Implements : adc_latch_clear_t_Class.

Converter

Converter specific methods.   These are used to configure and use the A/D Converter specific functionality, including:

- clock input and divider

- sample time in A/D clocks

- resolution

- trigger source

- voltage reference

- enable DMA

- enable continuous conversion on one channel

To start a conversion, a control channel (see Channel Configuration) and a trigger source must be configured. Once a conversion is started, the user application can wait for it to be finished by calling the ADC_DRV_WaitConvDone() function.

Only the first control channel can be triggered by software. To start a conversion in this case, an input channel must be written in the channel selection register using the ADC_DRV_ConfigChan() method. Writing a value to the control channel while a conversion is being performed on that channel will start a new conversion.

- void ADC_DRV_InitConverterStruct (adc_converter_config_t ∗const config)

    Initializes the converter configuration structure.
- void ADC_DRV_ConfigConverter (const uint32_t instance, const adc_converter_config_t ∗const config)

    Configures the converter with the given configuration structure.
- void ADC_DRV_GetConverterConfig (const uint32_t instance, adc_converter_config_t ∗const config)

    Gets the current converter configuration.
- void ADC_DRV_Reset (const uint32_t instance)

    Resets the converter (sets all configurations to reset values)
- void ADC_DRV_WaitConvDone (const uint32_t instance)

    Waits for a conversion/calibration to finish.
- bool ADC_DRV_GetConvCompleteFlag (const uint32_t instance, const uint8_t chanIndex)

    Gets the control channel Conversion Complete Flag state.

Hardware Compare

The Hardware Compare feature of the S32K144 ADC is a versatile mechanism that can be used to monitor that a value is within certain values. Measurements can be monitored to be within certain ranges:

- less than/ greater than a fixed value

- inside or outside of a certain range

Two compare values can be configured (the second value is used only for range function mode). The compare values must be written in 12-bit resolution mode regardless of the actual used resolution mode.

Once the hardware compare feature is enabled, a conversion is considered complete only when the measured value is within the allowable range set by the configuration.

- void ADC_DRV_InitHwCompareStruct (adc_compare_config_t ∗const config)

    Initializes the Hardware Compare configuration structure.
- void ADC_DRV_ConfigHwCompare (const uint32_t instance, const adc_compare_config_t ∗const config)

    Configures the Hardware Compare feature with the given configuration structure.
- void ADC_DRV_GetHwCompareConfig (const uint32_t instance, adc_compare_config_t ∗const config)

    Gets the current Hardware Compare configuration.

Hardware Average

The Hardware Average feature of the S32K144 allows for a set of measurements to be averaged together as a single conversion. The number of samples to be averaged is selectable (4, 8, 16 or 32 samples).

- void ADC_DRV_InitHwAverageStruct (adc_average_config_t ∗const config)

    Initializes the Hardware Average configuration structure.
- void ADC_DRV_ConfigHwAverage (const uint32_t instance, const adc_average_config_t ∗const config)

    Configures the Hardware Average feature with the given configuration structure.
- void ADC_DRV_GetHwAverageConfig (const uint32_t instance, adc_average_config_t ∗const config)

    Gets the current Hardware Average configuration.

Channel configuration

Control register specific functions. These functions control configurations for each control channel (input channel selection and interrupt enable).

When software triggering is enabled, calling the ADC_DRV_ConfigChan() method for control channel 0 starts a new conversion.

After a conversion is finished, the result can be retrieved using the ADC_DRV_GetChanResult() method.

- void ADC_DRV_InitChanStruct (adc_chan_config_t *const config)

  Initializes the control channel configuration structure.
- void ADC_DRV_ConfigChan (const uint32_t instance, const uint8_t chanIndex, const adc_chan_config_↩ t *const config)

  Configures the selected control channel with the given configuration structure.
- void ADC_DRV_GetChanConfig (const uint32_t instance, const uint8_t chanIndex, adc_chan_config_↩ t *const config)

  Gets the current control channel configuration for the selected channel index.
- void ADC_DRV_SetSwPretrigger (const uint32_t instance, const adc_sw_pretrigger_t swPretrigger)

  This function sets the software pretrigger - affects only first 4 control channels.
- void ADC_DRV_GetChanResult (const uint32_t instance, const uint8_t chanIndex, uint16_t *const result)

  Gets the last result for the selected control channel.

Automatic Calibration

These methods control the Calibration feature of the ADC.

The ADC_DRV_AutoCalibration() method can be called to execute a calibration sequence, or a calibration can be retrieved with the ADC_DRV_GetUserCalibration() and saved to non-volatile storage, to avoid calibration on every power-on. The calibration structure can be written with the ADC_DRV_ConfigUserCalibration() method.

- void ADC_DRV_AutoCalibration (const uint32_t instance)

  Executes an Auto-Calibration.
- void ADC_DRV_InitUserCalibrationStruct (adc_calibration_t *const config)

  Initializes the User Calibration configuration structure.
- void ADC_DRV_ConfigUserCalibration (const uint32_t instance, const adc_calibration_t *const config)

  Configures the User Calibration feature with the given configuration structure.
- void ADC_DRV_GetUserCalibration (const uint32_t instance, adc_calibration_t *const config)

  Gets the current User Calibration configuration.

Interrupts

This method returns the interrupt number for an ADC instance, which can be used to configure the interrupt, like in Interrupt Manager.

- IRQn_Type ADC_DRV_GetInterruptNumber (const uint32_t instance)

  Returns the interrupt number for the ADC instance.

Latched triggers processing

These functions provide basic operations for using the trigger latch mechanism.

- void ADC_DRV_ClearLatchedTriggers (const uint32_t instance, const adc_latch_clear_t clearMode)

  Clear latched triggers under processing.

- void ADC_DRV_ClearTriggerErrors (const uint32_t instance)

    Clear all latch trigger error.
- uint32_t ADC_DRV_GetTriggerErrorFlags (const uint32_t instance)

    Get the trigger error flags bits of the ADC instance.

16.1.2 Data Structure Documentation

16.1.2.1 struct adc_converter_config_t

Defines the converter configuration.

This structure is used to configure the ADC converter

Implements : adc_converter_config_t_Class

Definition at line 252 of file adc_driver.h.

Data Fields

- adc_clk_divide_t clockDivide
- uint8_t sampleTime
- adc_resolution_t resolution
- adc_input_clock_t inputClock
- adc_trigger_t trigger
- adc_pretrigger_sel_t pretriggerSel
- adc_trigger_sel_t triggerSel
- bool dmaEnable
- adc_voltage_reference_t voltageRef
- bool continuousConvEnable
- bool supplyMonitoringEnable

Field Documentation

16.1.2.1.1 adc_clk_divide_t clockDivide

Divider of the input clock for the ADC

Definition at line 254 of file adc_driver.h.

16.1.2.1.2 bool continuousConvEnable

Enable Continuous conversions

Definition at line 263 of file adc_driver.h.

16.1.2.1.3 bool dmaEnable

Enable DMA for the ADC

Definition at line 261 of file adc_driver.h.

16.1.2.1.4 adc_input_clock_t inputClock

Input clock source

Definition at line 257 of file adc_driver.h.

16.1.2.1.5 adc_pretrigger_sel_t pretriggerSel

Pretrigger source selected from Trigger Latching and Arbitration Unit - affects only the first 4 control channels

Definition at line 259 of file adc_driver.h.

16.1.2.1.6 adc_resolution_t resolution

ADC resolution (8,10,12 bit)

Definition at line 256 of file adc_driver.h.

16.1.2.1.7 uint8_t sampleTime

Sample time in AD Clocks

Definition at line 255 of file adc_driver.h.

16.1.2.1.8 bool supplyMonitoringEnable

Only available for ADC 0. Enable internal supply monitoring - enables measurement of ADC_INPUTCHAN_SUP↩
PLY_ sources.

Definition at line 264 of file adc_driver.h.

16.1.2.1.9 adc_trigger_t trigger

ADC trigger type (software, hardware) - affects only the first control channel

Definition at line 258 of file adc_driver.h.

16.1.2.1.10 adc_trigger_sel_t triggerSel

Trigger source selected from Trigger Latching and Arbitration Unit

Definition at line 260 of file adc_driver.h.

16.1.2.1.11 adc_voltage_reference_t voltageRef

Voltage reference used

Definition at line 262 of file adc_driver.h.

16.1.2.2 struct adc_compare_config_t

Defines the hardware compare configuration.

This structure is used to configure the hardware compare feature for the ADC

Implements : adc_compare_config_t_Class

Definition at line 275 of file adc_driver.h.

Data Fields

- bool compareEnable
- bool compareGreaterThanEnable
- bool compareRangeFuncEnable
- uint16_t compVal1
- uint16_t compVal2

Field Documentation

16.1.2.2.1 bool compareEnable

Enable the compare feature

Definition at line 277 of file adc_driver.h.

16.1.2.2.2 bool compareGreaterThanEnable

Enable Greater-Than functionality

Definition at line 278 of file adc_driver.h.

16.1.2.2.3   bool compareRangeFuncEnable

Enable Range functionality

Definition at line 279 of file adc_driver.h.

16.1.2.2.4   uint16_t compVal1

First Compare Value

Definition at line 280 of file adc_driver.h.

16.1.2.2.5   uint16_t compVal2

Second Compare Value

Definition at line 281 of file adc_driver.h.

16.1.2.3   struct adc_average_config_t

Defines the hardware average configuration.

This structure is used to configure the hardware average feature for the ADC

Implements : adc_average_config_t_Class

Definition at line 292 of file adc_driver.h.

Data Fields

- bool hwAvgEnable
- adc_average_t hwAverage

Field Documentation

16.1.2.3.1   adc_average_t hwAverage

Selection for number of samples used for averaging

Definition at line 295 of file adc_driver.h.

16.1.2.3.2   bool hwAvgEnable

Enable averaging functionality

Definition at line 294 of file adc_driver.h.

16.1.2.4   struct adc_chan_config_t

Defines the control channel configuration.

This structure is used to configure a control channel of the ADC

Implements : adc_chan_config_t_Class

Definition at line 306 of file adc_driver.h.

Data Fields

- bool interruptEnable
- adc_inputchannel_t channel

Field Documentation

16.1.2.4.1   adc_inputchannel_t channel

Selection of input channel for measurement

Definition at line 309 of file adc_driver.h.

16.1.2.4.2   bool interruptEnable

Enable interrupts for this channel

Definition at line 308 of file adc_driver.h.

16.1.2.5   struct adc_calibration_t

Defines the user calibration configuration.

This structure is used to configure the user calibration parameters of the ADC.

Implements : adc_calibration_t_Class

Definition at line 320 of file adc_driver.h.

Data Fields

- uint16_t userGain
- uint16_t userOffset

Field Documentation

16.1.2.5.1   uint16_t userGain

User-configurable gain

Definition at line 322 of file adc_driver.h.

16.1.2.5.2   uint16_t userOffset

User-configurable Offset (2's complement, subtracted from result)

Definition at line 323 of file adc_driver.h.

16.1.3   Enumeration Type Documentation

16.1.3.1   enum adc_average_t

Hardware average selection.

Implements : adc_average_t_Class

Enumerator

   ADC_AVERAGE_4   Hardware average of 4 samples.

   ADC_AVERAGE_8   Hardware average of 8 samples.

   ADC_AVERAGE_16   Hardware average of 16 samples.

   ADC_AVERAGE_32   Hardware average of 32 samples.

Definition at line 157 of file adc_driver.h.

16.1.3.2   enum adc_clk_divide_t

Clock Divider selection.

Implements : adc_clk_divide_t_Class

Enumerator

    ADC_CLK_DIVIDE_1   Input clock divided by 1.

    ADC_CLK_DIVIDE_2   Input clock divided by 2.

    ADC_CLK_DIVIDE_4   Input clock divided by 4.

    ADC_CLK_DIVIDE_8   Input clock divided by 8.

Definition at line 60 of file adc_driver.h.

16.1.3.3   enum **adc_input_clock_t**

Input clock source selection.

Implements : adc_input_clock_t_Class

Enumerator

    ADC_CLK_ALT_1   Input clock alternative 1.

    ADC_CLK_ALT_2   Input clock alternative 2.

    ADC_CLK_ALT_3   Input clock alternative 3.

    ADC_CLK_ALT_4   Input clock alternative 4.

Definition at line 85 of file adc_driver.h.

16.1.3.4   enum **adc_inputchannel_t**

Enumeration of input channels assignable to a control channel.
Note 0: entries in this enum are affected by ::FEATURE_ADC_NUM_EXT_CHANS, which is device dependent and controlled from *"device_name"_features.h* file.

Note 1: the actual number of external channels may differ between device packages and ADC instances. Reading a channel that is not connected externally, will return a random value within the range. Please refer to the Reference Manual for the maximum number of external channels for each device variant and ADC instance.

Note 2: ADC_INPUTCHAN_SUPPLY_ select which internal supply channel to be measured. They are only available for ADC0 and measured internally via internal input channel 0. Please note that supply monitoring needs to be enabled separately via dedicated flag in adc_converter_config_t.

Implements : adc_inputchannel_t_Class

Enumerator

    ADC_INPUTCHAN_EXT0   External input channel 0

    ADC_INPUTCHAN_EXT1   External input channel 1

    ADC_INPUTCHAN_EXT3   External input channel 3

    ADC_INPUTCHAN_EXT4   External input channel 4

    ADC_INPUTCHAN_EXT5   External input channel 5

    ADC_INPUTCHAN_EXT6   External input channel 6

    ADC_INPUTCHAN_EXT7   External input channel 7

    ADC_INPUTCHAN_EXT9   External input channel 9

    ADC_INPUTCHAN_EXT10   External input channel 10

    ADC_INPUTCHAN_EXT11   External input channel 11

    ADC_INPUTCHAN_EXT12   External input channel 12

    ADC_INPUTCHAN_EXT13   External input channel 13

    ADC_INPUTCHAN_EXT14   External input channel 14

> ***ADC_INPUTCHAN_DISABLED*** Channel disabled
>
> ***ADC_INPUTCHAN_INT0*** Internal input channel 0
>
> ***ADC_INPUTCHAN_INT1*** Internal input channel 1
>
> ***ADC_INPUTCHAN_INT2*** Internal input channel 2
>
> ***ADC_INPUTCHAN_INT3*** Internal input channel 3
>
> ***ADC_INPUTCHAN_TEMP*** Temperature Sensor
>
> ***ADC_INPUTCHAN_BANDGAP*** Band Gap
>
> ***ADC_INPUTCHAN_VREFSH*** Voltage Reference Select High
>
> ***ADC_INPUTCHAN_VREFSL*** Voltage Reference Select Low
>
> ***ADC_INPUTCHAN_SUPPLY_VDD*** Monitor internal supply 5 V input VDD supply.
>
> ***ADC_INPUTCHAN_SUPPLY_VDDA*** Monitor internal supply 5 V input analog supply.
>
> ***ADC_INPUTCHAN_SUPPLY_VREFH*** Monitor internal supply ADC reference supply.
>
> ***ADC_INPUTCHAN_SUPPLY_VDD_3V*** Monitor internal supply 3.3 V oscillator regulator output.
>
> ***ADC_INPUTCHAN_SUPPLY_VDD_FLASH_3V*** Monitor internal supply 3.3 V flash regulator output.
>
> ***ADC_INPUTCHAN_SUPPLY_VDD_LV*** Monitor internal supply 1.2 V core regulator output.

Definition at line 180 of file adc_driver.h.

16.1.3.5 enum adc_latch_clear_t

Defines the trigger latch clear method Implements : adc_latch_clear_t_Class.

Enumerator

> ***ADC_LATCH_CLEAR_WAIT*** Clear by waiting all latched triggers to be processed
>
> ***ADC_LATCH_CLEAR_FORCE*** Process current trigger and clear all latched

Definition at line 330 of file adc_driver.h.

16.1.3.6 enum adc_pretrigger_sel_t

Pretrigger types selectable from Trigger Latching and Arbitration Unit.

Implements : adc_pretrigger_sel_t_Class

Enumerator

> ***ADC_PRETRIGGER_SEL_PDB*** PDB pretrigger selected.
>
> ***ADC_PRETRIGGER_SEL_TRGMUX*** TRGMUX pretrigger selected.
>
> ***ADC_PRETRIGGER_SEL_SW*** Software pretrigger selected.

Definition at line 109 of file adc_driver.h.

16.1.3.7 enum adc_resolution_t

Conversion resolution selection.

Implements : adc_resolution_t_Class

Enumerator

> ***ADC_RESOLUTION_8BIT*** 8-bit resolution mode
>
> ***ADC_RESOLUTION_12BIT*** 12-bit resolution mode
>
> ***ADC_RESOLUTION_10BIT*** 10-bit resolution mode

Definition at line 73 of file adc_driver.h.

16.1.3.8 enum adc_sw_pretrigger_t

Software pretriggers which may be set from Trigger Latching and Arbitration Unit.

Implements : adc_sw_pretrigger_t_Class

Enumerator

> **ADC_SW_PRETRIGGER_DISABLED** SW pretrigger disabled.
> **ADC_SW_PRETRIGGER_0** SW pretrigger 0.
> **ADC_SW_PRETRIGGER_1** SW pretrigger 1.
> **ADC_SW_PRETRIGGER_2** SW pretrigger 2.
> **ADC_SW_PRETRIGGER_3** SW pretrigger 3.

Definition at line 132 of file adc_driver.h.

16.1.3.9 enum adc_trigger_sel_t

Trigger source selectable from Trigger Latching and Arbitration Unit.

Implements : adc_trigger_sel_t_Class

Enumerator

> **ADC_TRIGGER_SEL_PDB** PDB trigger selected.
> **ADC_TRIGGER_SEL_TRGMUX** TRGMUX trigger selected.

Definition at line 121 of file adc_driver.h.

16.1.3.10 enum adc_trigger_t

Trigger type selection.

Implements : adc_trigger_t_Class

Enumerator

> **ADC_TRIGGER_SOFTWARE** Software trigger.
> **ADC_TRIGGER_HARDWARE** Hardware trigger.

Definition at line 98 of file adc_driver.h.

16.1.3.11 enum adc_voltage_reference_t

Voltage reference selection.

Implements : adc_voltage_reference_t_Class

Enumerator

> **ADC_VOLTAGEREF_VREF** VrefH and VrefL as Voltage reference.
> **ADC_VOLTAGEREF_VALT** ValtH and ValtL as Voltage reference.

Definition at line 146 of file adc_driver.h.

16.1.4 Function Documentation

16.1.4.1 void ADC_DRV_AutoCalibration (const uint32_t *instance*)

Executes an Auto-Calibration.

This functions executes an Auto-Calibration sequence. It is recommended to run this sequence before using the ADC converter.

Parameters

in	*instance*	instance number

Definition at line 528 of file adc_driver.c.

16.1.4.2 void ADC_DRV_ClearLatchedTriggers (const uint32_t *instance,* const adc_latch_clear_t *clearMode*)

Clear latched triggers under processing.

This function clears all trigger latched flags of the ADC instance. This function must be called after the hardware trigger source for the ADC has been deactivated.

Parameters

in	*instance*	instance number of the ADC
in	*clearMode*	The clearing method for the latched triggers • ADC_LATCH_CLEAR_WAIT : Wait for all latched triggers to be processed. • ADC_LATCH_CLEAR_FORCE : Clear latched triggers and wait for trigger being process to finish.

Definition at line 643 of file adc_driver.c.

16.1.4.3 void ADC_DRV_ClearTriggerErrors (const uint32_t *instance*)

Clear all latch trigger error.

This function clears all trigger error flags of the ADC instance.

Parameters

in	*instance*	instance number of the ADC

Definition at line 668 of file adc_driver.c.

16.1.4.4 void ADC_DRV_ConfigChan (const uint32_t *instance,* const uint8_t *chanIndex,* const adc_chan_config_t ∗const *config*)

Configures the selected control channel with the given configuration structure.

When Software Trigger mode is enabled, configuring control channel index 0, implicitly triggers a new conversion on the selected ADC input channel. Therefore, ADC_DRV_ConfigChan can be used for sw-triggering conversions.

Configuring any control channel while it is actively controlling a conversion (sw or hw triggered) will implicitly abort the on-going conversion.

Parameters

in	*instance*	instance number
in	*chanIndex*	the control channel index
in	*config*	the configuration structure

Definition at line 362 of file adc_driver.c.

16.1.4.5 void ADC_DRV_ConfigConverter (const uint32_t *instance,* const adc_converter_config_t ∗const *config*)

Configures the converter with the given configuration structure.

This function configures the ADC converter with the options provided in the provided structure.

Parameters

in	*instance*	instance number
in	*config*	the configuration structure

Definition at line 84 of file adc_driver.c.

16.1.4.6 void ADC_DRV_ConfigHwAverage (const uint32_t *instance,* const **adc_average_config_t ∗const *config*)**

Configures the Hardware Average feature with the given configuration structure.

This function sets the configuration for the Hardware Average feature.

Parameters

in	*instance*	instance number
in	*config*	the configuration structure

Definition at line 299 of file adc_driver.c.

16.1.4.7 void ADC_DRV_ConfigHwCompare (const uint32_t *instance,* const **adc_compare_config_t ∗const *config*)**

Configures the Hardware Compare feature with the given configuration structure.

This functions sets the configuration for the Hardware Compare feature using the configuration structure.

Parameters

in	*instance*	instance number
in	*config*	the configuration structure

Definition at line 236 of file adc_driver.c.

16.1.4.8 void ADC_DRV_ConfigUserCalibration (const uint32_t *instance,* const **adc_calibration_t ∗const *config*)**

Configures the User Calibration feature with the given configuration structure.

This function sets the configuration for the user calibration registers.

Parameters

in	*instance*	instance number
in	*config*	the configuration structure

Definition at line 589 of file adc_driver.c.

16.1.4.9 void ADC_DRV_GetChanConfig (const uint32_t *instance,* const uint8_t *chanIndex,* **adc_chan_config_t ∗const *config*)**

Gets the current control channel configuration for the selected channel index.

This function returns the configuration for a control channel

Parameters

in	*instance*	instance number
in	*chanIndex*	the control channel index
out	*config*	the configuration structure

Definition at line 387 of file adc_driver.c.

16.1.4.10 void ADC_DRV_GetChanResult (const uint32_t *instance,* const uint8_t *chanIndex,* uint16_t ∗const *result*)

Gets the last result for the selected control channel.

This function returns the conversion result from a control channel.

Parameters

in	*instance*	instance number
in	*chanIndex*	the converter control channel index
out	*result*	the result raw value

Definition at line 494 of file adc_driver.c.

16.1.4.11   bool ADC_DRV_GetConvCompleteFlag (const uint32_t *instance,* const uint8_t *chanIndex*)

Gets the control channel Conversion Complete Flag state.

This function returns the state of the Conversion Complete flag for a control channel. This flag is set when a conversion is complete or the condition generated by the Hardware Compare feature is evaluated to true.

Parameters

in	*instance*	instance number
in	*chanIndex*	the adc control channel index

Returns

the Conversion Complete Flag state

Definition at line 467 of file adc_driver.c.

16.1.4.12   void ADC_DRV_GetConverterConfig (const uint32_t *instance,* adc_converter_config_t ∗const *config*)

Gets the current converter configuration.

This functions returns the configuration for converter in the form of a configuration structure.

Parameters

in	*instance*	instance number
out	*config*	the configuration structure

Definition at line 121 of file adc_driver.c.

16.1.4.13   void ADC_DRV_GetHwAverageConfig (const uint32_t *instance,* adc_average_config_t ∗const *config*)

Gets the current Hardware Average configuration.

This function returns the configuration for the Hardware Average feature.

Parameters

in	*instance*	instance number
out	*config*	the configuration structure

Definition at line 318 of file adc_driver.c.

16.1.4.14   void ADC_DRV_GetHwCompareConfig (const uint32_t *instance,* adc_compare_config_t ∗const *config*)

Gets the current Hardware Compare configuration.

This function returns the configuration for the Hardware Compare feature.

Parameters

in	*instance*	instance number
out	*config*	the configuration structure

Definition at line 258 of file adc_driver.c.

16.1.4.15 IRQn_Type ADC_DRV_GetInterruptNumber (const uint32_t *instance*)

Returns the interrupt number for the ADC instance.

This function returns the interrupt number for the specified ADC instance.

Parameters

in	*instance*	instance number of the ADC

Returns

> irq_number: the interrupt number (index) of the ADC instance, used to configure the interrupt

Definition at line 626 of file adc_driver.c.

16.1.4.16 uint32_t ADC_DRV_GetTriggerErrorFlags (const uint32_t *instance*)

Get the trigger error flags bits of the ADC instance.

This function returns the trigger error flags bits of the ADC instance.

Parameters

in	*instance*	instance number of the ADC

Returns

> trigErrorFlags The Trigger Error Flags bit-mask

Definition at line 684 of file adc_driver.c.

16.1.4.17 void ADC_DRV_GetUserCalibration (const uint32_t *instance,* adc_calibration_t ∗const *config*)

Gets the current User Calibration configuration.

This function returns the current user calibration register values.

Parameters

in	*instance*	instance number
out	*config*	the configuration structure

Definition at line 608 of file adc_driver.c.

16.1.4.18 void ADC_DRV_InitChanStruct (adc_chan_config_t ∗const *config*)

Initializes the control channel configuration structure.

This function initializes the control channel configuration structure to default values (Reference Manual resets). This function should be called on a structure before using it to configure a channel (ADC_DRV_ConfigChan), otherwise all members must be written by the caller. This function insures that all members are written with safe values, so the user can modify only the desired members.

Parameters

out	*config*	the configuration structure

Definition at line 340 of file adc_driver.c.

16.1.4.19 void ADC_DRV_InitConverterStruct (adc_converter_config_t ∗const *config*)

Initializes the converter configuration structure.

This function initializes the members of the adc_converter_config_t structure to default values (Reference Manual resets). This function should be called on a structure before using it to configure the converter with ADC_DRV←

_ConfigConverter(), otherwise all members must be written (initialized) by the user. This function insures that all members are written with safe values, so the user can modify only the desired members.

Parameters

out	*config*	the configuration structure

Definition at line 59 of file adc_driver.c.

16.1.4.20 **void ADC_DRV_InitHwAverageStruct (adc_average_config_t ∗const *config*)**

Initializes the Hardware Average configuration structure.

This function initializes the Hardware Average configuration structure to default values (Reference Manual resets). This function should be called before configuring the Hardware Average feature (ADC_DRV_ConfigHwAverage), otherwise all members must be written by the caller. This function insures that all members are written with safe values, so the user can modify the desired members.

Parameters

out	*config*	the configuration structure

Definition at line 283 of file adc_driver.c.

16.1.4.21 **void ADC_DRV_InitHwCompareStruct (adc_compare_config_t ∗const *config*)**

Initializes the Hardware Compare configuration structure.

This function initializes the Hardware Compare configuration structure to default values (Reference Manual resets). This function should be called before configuring the Hardware Compare feature (ADC_DRV_ConfigHwCompare), otherwise all members must be written by the caller. This function insures that all members are written with safe values, so the user can modify the desired members.

Parameters

out	*config*	the configuration structure

Definition at line 217 of file adc_driver.c.

16.1.4.22 **void ADC_DRV_InitUserCalibrationStruct (adc_calibration_t ∗const *config*)**

Initializes the User Calibration configuration structure.

This function initializes the User Calibration configuration structure to default values (Reference Manual resets). This function should be called on a structure before using it to configure the User Calibration feature (ADC_DRV_↩ ConfigUserCalibration), otherwise all members must be written by the caller. This function insures that all members are written with safe values, so the user can modify only the desired members.

Parameters

out	*config*	the configuration structure

Definition at line 573 of file adc_driver.c.

16.1.4.23 **void ADC_DRV_Reset (const uint32_t *instance*)**

Resets the converter (sets all configurations to reset values)

This function resets all the internal ADC registers to reset values.

Parameters

in	*instance*	instance number

Definition at line 159 of file adc_driver.c.

16.1.4.24 **void ADC_DRV_SetSwPretrigger (const uint32_t *instance,* const adc_sw_pretrigger_t *swPretrigger*)**

This function sets the software pretrigger - affects only first 4 control channels.

Parameters

in	*instance*	instance number
in	*swPretrigger*	the swPretrigger to be enabled

Definition at line 407 of file adc_driver.c.

16.1.4.25   void ADC_DRV_WaitConvDone (const uint32_t *instance*)

Waits for a conversion/calibration to finish.

This functions waits for a conversion to complete by continuously polling the Conversion Active Flag.

Parameters

in	*instance*	instance number

Definition at line 447 of file adc_driver.c.

16.2 Backward Compatibility Symbols for S32K116

This module covers backward compatibility symbols.

16.3   CRC Driver

16.3.1   Detailed Description

This section describes the programming interface of the CRC driver.

Basic Operations of CRC

1. To initialize the CRC module, call CRC_DRV_Init() function and pass the user configuration data structure to it.
   This is example code to configure the CRC driver use CRC_DRV_GetDefaultConfig function:

   ```
   #define INST_CRC1 (0U)

   /* Configuration structure crc1_InitConfig0 */
   crc_user_config_t crc1_InitConfig0;

   /* Get default configuration for CRC module: CRC-16-CCITT (0x1021) standard */
   CRC_DRV_GetDefaultConfig(&crc1_InitConfig0);

   /* Initializes the CRC */
   CRC_DRV_Init(INST_CRC1, &crc1_InitConfig0);
   ```

2. To configuration and operation CRC module:
   Function CRC_DRV_Configure() shall be used to write user configuration to CRC hardware module before starting operation by calling CRC_DRV_WriteData().
   Finally, using CRC_DRV_GetCrcResult() function to get the result of CRC calculation.
   This is example code to Configure and get CRC block for S32K1xx:

   ```
   #define INST_CRC1 (0U)

   uint8_t buffer[] = { 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x30 };
   uint32_t result;

   /* Set the CRC configuration: CRC-16-CCITT (0x1021) standard */
   CRC_DRV_Configure(INST_CRC1, &crc1_InitConfig0);
   /* Write data to the current CRC calculation */
   CRC_DRV_WriteData(INST_CRC1, buffer, 10U);
   /* Get result of CRC calculation (0x3218U) */
   result = CRC_DRV_GetCrcResult(INST_CRC1);

   /* De-init */
   CRC_DRV_Deinit(INST_CRC1);
   ```

3. To Get result of 32-bit data then call CRC_DRV_GetCrc32() function.

   ```
   #define INST_CRC1 (0U)

   uint32_t seed = 0xFFFFU;
   uint32_t data = 0x12345678U;
   uint32_t result;

   /* Get result of 32-bit data (0x30EC) at CRC-16-CCITT (0x1021) standard configuration mode */
   result = CRC_DRV_GetCrc32(INST_CRC1, data, true, seed);
   ```

4. To Get result of 16-bit data then call CRC_DRV_GetCrc16() function.

   ```
   #define INST_CRC1 (0U)

   uint32_t seed = 0xFFFFU;
   uint16_t data = 0x1234U;
   uint32_t result;

   /* Get result of 16-bit data (0x0EC9) at CRC-16-CCITT (0x1021) standard configuration mode */
   result = CRC_DRV_GetCrc16(INST_CRC1, data, true, seed);
   ```

5. To Get current configuration of the CRC module, just call CRC_DRV_GetConfig() function.

```
#define INST_CRC1 (0U)
crc_user_config_t crc1_InitConfig0;

/* Get current configuration of the CRC module */
CRC_DRV_GetConfig(INST_CRC1, &crc1_InitConfig0);
```

6. To Get default configuration of the CRC module, just call CRC_DRV_GetDefaultConfig() function.

```
#define INST_CRC1 (0U)
crc_user_config_t crc1_InitConfig0;

/* Get default configuration of the CRC module */
CRC_DRV_GetDefaultConfig(&crc1_InitConfig0);
```

Data Structures

- struct crc_user_config_t

    CRC configuration structure. Implements : crc_user_config_t_Class. More...

Enumerations

- enum crc_transpose_t { CRC_TRANSPOSE_NONE = 0x00U, CRC_TRANSPOSE_BITS = 0x01U, CRC_↩
  TRANSPOSE_BITS_AND_BYTES = 0x02U, CRC_TRANSPOSE_BYTES = 0x03U }

    CRC type of transpose of read write data Implements : crc_transpose_t_Class.

CRC DRIVER API

- status_t CRC_DRV_Init (uint32_t instance, const crc_user_config_t ∗userConfigPtr)

    Initializes the CRC module.

- status_t CRC_DRV_Deinit (uint32_t instance)

    Sets the default configuration.

- uint32_t CRC_DRV_GetCrc32 (uint32_t instance, uint32_t data, bool newSeed, uint32_t seed)

    Appends 32-bit data to the current CRC calculation and returns new result.

- uint32_t CRC_DRV_GetCrc16 (uint32_t instance, uint16_t data, bool newSeed, uint32_t seed)

    Appends 16-bit data to the current CRC calculation and returns new result.

- uint32_t CRC_DRV_GetCrc8 (uint32_t instance, uint8_t data, bool newSeed, uint32_t seed)

    Appends 8-bit data to the current CRC calculation and returns new result.

- void CRC_DRV_WriteData (uint32_t instance, const uint8_t ∗data, uint32_t dataSize)

    Appends a block of bytes to the current CRC calculation.

- uint32_t CRC_DRV_GetCrcResult (uint32_t instance)

    Returns the current result of the CRC calculation.

- status_t CRC_DRV_Configure (uint32_t instance, const crc_user_config_t ∗userConfigPtr)

    Configures the CRC module from a user configuration structure.

- status_t CRC_DRV_GetConfig (uint32_t instance, crc_user_config_t ∗const userConfigPtr)

    Get configures of the CRC module currently.

- status_t CRC_DRV_GetDefaultConfig (crc_user_config_t ∗const userConfigPtr)

    Get default configures the CRC module for configuration structure.

16.3.2 Data Structure Documentation

16.3.2.1 struct crc_user_config_t

CRC configuration structure. Implements : crc_user_config_t_Class.

Definition at line 85 of file crc_driver.h.

Data Fields

- crc_transpose_t writeTranspose
- bool complementChecksum
- uint32_t seed

Field Documentation

16.3.2.1.1 bool complementChecksum

True if the result shall be complement of the actual checksum.

Definition at line 97 of file crc_driver.h.

16.3.2.1.2 uint32_t seed

Starting checksum value.

Definition at line 98 of file crc_driver.h.

16.3.2.1.3 crc_transpose_t writeTranspose

Type of transpose when writing CRC input data.

Definition at line 96 of file crc_driver.h.

16.3.3 Enumeration Type Documentation

16.3.3.1 enum crc_transpose_t

CRC type of transpose of read write data Implements : crc_transpose_t_Class.

Enumerator

CRC_TRANSPOSE_NONE   No transpose

CRC_TRANSPOSE_BITS   Transpose bits in bytes

CRC_TRANSPOSE_BITS_AND_BYTES   Transpose bytes and bits in bytes

CRC_TRANSPOSE_BYTES   Transpose bytes

Definition at line 46 of file crc_driver.h.

16.3.4 Function Documentation

16.3.4.1 status_t CRC_DRV_Configure (uint32_t *instance,* const crc_user_config_t ∗ *userConfigPtr*)

Configures the CRC module from a user configuration structure.

This function configures the CRC module from a user configuration structure

Parameters

in	*instance*	The CRC instance number
in	*userConfigPtr*	Pointer to structure of initialization

Returns

Execution status (success)

Definition at line 239 of file crc_driver.c.

16.3.4.2 status_t CRC_DRV_Deinit (uint32_t *instance*)

Sets the default configuration.

This function sets the default configuration

Parameters

in	*instance*	The CRC instance number

Returns

Execution status (success)

Definition at line 91 of file crc_driver.c.

16.3.4.3 status_t CRC_DRV_GetConfig (uint32_t *instance,* crc_user_config_t ∗const *userConfigPtr*)

Get configures of the CRC module currently.

This function Get configures of the CRC module currently

Parameters

in	*instance*	The CRC instance number
out	*userConfigPtr*	Pointer to structure of initialization

Returns

Execution status (success)

Definition at line 271 of file crc_driver.c.

16.3.4.4 uint32_t CRC_DRV_GetCrc16 (uint32_t *instance,* uint16_t *data,* bool *newSeed,* uint32_t *seed*)

Appends 16-bit data to the current CRC calculation and returns new result.

This function appends 16-bit data to the current CRC calculation and returns new result. If the newSeed is true, seed set and result are calculated from the seed new value (new CRC calculation)

Parameters

in	*instance*	The CRC instance number
in	*data*	Input data for CRC calculation
in	*newSeed*	Sets new CRC calculation • true: New seed set and used for new calculation. • false: Seed argument ignored, continues old calculation.
in	*seed*	New seed if newSeed is true, else ignored

Returns

> New CRC result

Definition at line 142 of file crc_driver.c.

16.3.4.5 uint32_t CRC_DRV_GetCrc32 (uint32_t *instance,* uint32_t *data,* bool *newSeed,* uint32_t *seed*)

Appends 32-bit data to the current CRC calculation and returns new result.

This function appends 32-bit data to the current CRC calculation and returns new result. If the newSeed is true, seed set and result are calculated from the seed new value (new CRC calculation)

Parameters

in	*instance*	The CRC instance number
in	*data*	Input data for CRC calculation
in	*newSeed*	Sets new CRC calculation • true: New seed set and used for new calculation. • false: Seed argument ignored, continues old calculation.
in	*seed*	New seed if newSeed is true, else ignored

Returns

> New CRC result

Definition at line 111 of file crc_driver.c.

16.3.4.6 uint32_t CRC_DRV_GetCrc8 (uint32_t *instance,* uint8_t *data,* bool *newSeed,* uint32_t *seed*)

Appends 8-bit data to the current CRC calculation and returns new result.

This function appends 8-bit data to the current CRC calculation and returns new result. If the newSeed is true, seed set and result are calculated from the seed new value (new CRC calculation)

Parameters

in	*instance*	The CRC instance number
in	*data*	Input data for CRC calculation
in	*newSeed*	Sets new CRC calculation • true: New seed set and used for new calculation. • false: Seed argument ignored, continues old calculation.
in	*seed*	New seed if newSeed is true, else ignored

Returns

> New CRC result

Definition at line 172 of file crc_driver.c.

16.3.4.7 uint32_t CRC_DRV_GetCrcResult (uint32_t *instance*)

Returns the current result of the CRC calculation.

This function returns the current result of the CRC calculation

Parameters

in	*instance*	The CRC instance number

Returns

> Result of CRC calculation

Definition at line 223 of file crc_driver.c.

16.3.4.8   status_t CRC_DRV_GetDefaultConfig (crc_user_config_t ∗const *userConfigPtr*)

Get default configures the CRC module for configuration structure.

This function Get default configures the CRC module for user configuration structure

Parameters

out	*userConfigPtr*	Pointer to structure of initialization

Returns

> Execution status (success)

Definition at line 303 of file crc_driver.c.

16.3.4.9   status_t CRC_DRV_Init (uint32_t *instance,* const crc_user_config_t ∗ *userConfigPtr*)

Initializes the CRC module.

This function initializes CRC driver based on user configuration input. The user must make sure that the clock is enabled

Parameters

in	*instance*	The CRC instance number
in	*userConfigPtr*	Pointer to structure of initialization

Returns

> Execution status (success)

Definition at line 68 of file crc_driver.c.

16.3.4.10   void CRC_DRV_WriteData (uint32_t *instance,* const uint8_t ∗ *data,* uint32_t *dataSize*)

Appends a block of bytes to the current CRC calculation.

This function appends a block of bytes to the current CRC calculation

Parameters

in	*instance*	The CRC instance number
in	*data*	Data for current CRC calculation
in	*dataSize*	Length of data to be calculated

Definition at line 200 of file crc_driver.c.

16.4 CSEc Driver

16.4.1 Detailed Description

Cryptographic Services Engine Peripheral Driver.

How to use the CSEc driver in your application

To access the command feature set, the part must be configured for EEE operation, using the PGMPART command. This can be implemented by using the Flash driver. By enabling security features and configuring a number of user keys, the total size of the 4 KByte EEERAM will be reduced by the space required to store the user keys. The user key space will then effectively be unaddressable space in the EEERAM.

At the bottom of this page is an example of making this configuration using the Flash driver. For more details related to the FLASH_DRV_DEFlashPartition function, please refer to the Flash driver documentation. Please note that this configuration is required only once and should not be lanched from Flash memory.

In order to use the CSEc driver in your application, the **CSEC_DRV_Init** function should be called prior to using the rest of the API. The parameter of this function is used for holding the internal state of the driver throughout the lifetime of the application.

Key/seed/random number generation

This is the high level flow in which to initialize and generate random numbers.

1. Run **CSEC_DRV_InitRNG** to initialize a random seed from the internal TRNG

    - **CSEC_DRV_InitRNG** must be run after every POR, and before the first execution of CSEC_DRV_↩ GenerateRND

    - Note that if the next step (run **CSEC_DRV_GenerateRND**) is run without initializing the seed, **CSEC↩ _RNG_SEED** will be returned.

2. Run **CSEC_DRV_GenerateRND** to generate a random numer The PRNG uses the PRNG_STATE/KEY and Seed per SHE spec and the AIS20 standard.

3. For additional random numbers the user may continue executing **CSEC_DRV_GenerateRND** unless a POR event occurred.

Memory update protocol

In order to update a key, the user must have knowledge of a valid authentication secret, i.e. another key (AuthID). If the key AuthID is empty, the key update will only work if AuthID = ID (the key that will be updated will represent the AuthID from now on), otherwise **CSEC_KEY_EMPTY** is returned.

The M1-M3 values need to be computed according to the SHE Specification in order to update a key slot. The **CSEC_DRV_LoadKey** function will require those values. After successfully updating the key slot, two verification values will be returned: M4 and M5. The user can compute the two values and compare them with the ones returned by the **CSEC_DRV_LoadKey** function in order to ensure the slot was updated as desired. Please refer to the CSEc driver example for a reference implementation of the memory update protocol.

Examples:

Using the Flash driver to partition Flash for CSEc operation

```
flash_ssd_config_t flashSSDConfig;

FLASH_DRV_Init(&flash1_InitConfig0, &flashSSDConfig);

/* Configure the part for EEE operation, with 20 keys for CSEc */
FLASH_DRV_DEFlashPartition(&flashSSDConfig, 0x2, 0x4, 0x3, false);
```

Encryption using AES EBC mode

```
uint8_t plainText[16] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88,
    0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF};
uint8_t plainKey[16] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

csec_error_code_t stat;
uint8_t cipherText[16];

csec_state_t csecState;

CSEC_DRV_Init(&csecState);

stat = CSEC_DRV_LoadPlainKey(plainKey);
if (stat != CSEC_NO_ERROR)
{
    /* Loading the key failed, encryption will not have the expected result */
    return false;
}

stat = CSEC_DRV_EncryptECB(CSEC_RAM_KEY, plainText, 16U, cipherText, 1U);
if (stat != CSEC_NO_ERROR)
{
    /* Encryption was successful */
    return true;
}
```

Generating and verifying CMAC for a message

```
uint8_t plainKey[16] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab,
    0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
uint8_t msg[16] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
uint8_t cmac[16];
bool verifStatus;
csec_error_code_t stat;

csec_state_t csecState;

CSEC_DRV_Init(&csecState);

stat = CSEC_DRV_LoadPlainKey(plainKey);
if (stat != CSEC_NO_ERROR)
    return false;

stat = CSEC_DRV_GenerateMAC(CSEC_RAM_KEY, msg, 128U, cmac, 1U);
if (stat != CSEC_NO_ERROR)
    return false;

stat = CSEC_DRV_VerifyMAC(CSEC_RAM_KEY, msg, 128U, cmac, 128U, &verifStatus,
    1U);
if (stat != CSEC_NO_ERROR)
    return false;

if (!verifStatus)
{
    /* The given CMAC did not matched with the one computed internally */
    return false;
}
```

Generating random bits

```
csec_error_code_t stat;
csec_status_t status;
uint8_t rnd[16];

csec_state_t csecState;

CSEC_DRV_Init(&csecState);

stat = CSEC_DRV_InitRNG();
if (stat != CSEC_NO_ERROR)
    return false;

/* Check RNG is initialized */
status = CSEC_DRV_GetStatus();
if (!(status & CSEC_STATUS_RND_INIT))
    return false;

stat = CSEC_DRV_GenerateRND(rnd);
if (stat != CSEC_NO_ERROR)
    return false;
```

Data Structures

- struct csec_state_t

    Internal driver state information. More...

Macros

- #define CSEC_STATUS_BUSY (0x1U)

    The bit is set whenever SHE is processing a command.
- #define CSEC_STATUS_SECURE_BOOT (0x2U)

    The bit is set if the secure booting is activated.
- #define CSEC_STATUS_BOOT_INIT (0x4U)

    The bit is set if the secure booting has been personalized during the boot sequence.
- #define CSEC_STATUS_BOOT_FINISHED (0x8U)

    The bit is set when the secure booting has been finished by calling either CMD_BOOT_FAILURE or CMD_BOOT←
    _OK or if CMD_SECURE_BOOT failed in verifying BOOT_MAC.
- #define CSEC_STATUS_BOOT_OK (0x10U)

    The bit is set if the secure booting (CMD_SECURE_BOOT) succeeded. If CMD_BOOT_FAILURE is called the bit is
    erased.
- #define CSEC_STATUS_RND_INIT (0x20U)

    The bit is set if the random number generator has been initialized.
- #define CSEC_STATUS_EXT_DEBUGGER (0x40U)

    The bit is set if an external debugger is connected to the chip.
- #define CSEC_STATUS_INT_DEBUGGER (0x80U)

    The bit is set if the internal debugging mechanisms of SHE are activated.

Typedefs

- typedef uint8_t csec_status_t

    Represents the status of the CSEc module. Provides one bit for each status code as per SHE specification. CSE←
    C_STATUS_∗ masks can be used for verifying the status.

Enumerations

- enum csec_key_id_t {
    CSEC_SECRET_KEY = 0x0U, CSEC_MASTER_ECU, CSEC_BOOT_MAC_KEY, CSEC_BOOT_MAC,
    CSEC_KEY_1, CSEC_KEY_2, CSEC_KEY_3, CSEC_KEY_4,
    CSEC_KEY_5, CSEC_KEY_6, CSEC_KEY_7, CSEC_KEY_8,
    CSEC_KEY_9, CSEC_KEY_10, CSEC_RAM_KEY = 0xFU, CSEC_KEY_11 = 0x14U,
    CSEC_KEY_12, CSEC_KEY_13, CSEC_KEY_14, CSEC_KEY_15,
    CSEC_KEY_16, CSEC_KEY_17 }

    Specify the KeyID to be used to implement the requested cryptographic operation.
- enum csec_cmd_t {
    CSEC_CMD_ENC_ECB = 0x1U, CSEC_CMD_ENC_CBC, CSEC_CMD_DEC_ECB, CSEC_CMD_DEC_←
    CBC,
    CSEC_CMD_GENERATE_MAC, CSEC_CMD_VERIFY_MAC, CSEC_CMD_LOAD_KEY, CSEC_CMD_L←
    OAD_PLAIN_KEY,
    CSEC_CMD_EXPORT_RAM_KEY, CSEC_CMD_INIT_RNG, CSEC_CMD_EXTEND_SEED, CSEC_CM←
    D_RND,
    CSEC_CMD_RESERVED_1, CSEC_CMD_BOOT_FAILURE, CSEC_CMD_BOOT_OK, CSEC_CMD_GE←
    T_ID,
    CSEC_CMD_BOOT_DEFINE, CSEC_CMD_DBG_CHAL, CSEC_CMD_DBG_AUTH, CSEC_CMD_RESE←
    RVED_2,
    CSEC_CMD_RESERVED_3, CSEC_CMD_MP_COMPRESS }

CSEc commands which follow the same values as the SHE command definition.

- enum csec_call_sequence_t { CSEC_CALL_SEQ_FIRST, CSEC_CALL_SEQ_SUBSEQUENT }

    Specifies if the information is the first or a following function call.

- enum csec_boot_flavor_t { CSEC_BOOT_STRICT, CSEC_BOOT_SERIAL, CSEC_BOOT_PARALLEL, C↩
SEC_BOOT_NOT_DEFINED }

    Specifies the boot type for the BOOT_DEFINE command.

Functions

- void CSEC_DRV_Init (csec_state_t *state)

    Initializes the internal state of the driver and enables the FTFC interrupt.

- void CSEC_DRV_Deinit (void)

    Clears the internal state of the driver and disables the FTFC interrupt.

- status_t CSEC_DRV_EncryptECB (csec_key_id_t keyId, const uint8_t *plainText, uint32_t length, uint8_t *cipherText, uint32_t timeout)

    Performs the AES-128 encryption in ECB mode.

- status_t CSEC_DRV_DecryptECB (csec_key_id_t keyId, const uint8_t *cipherText, uint32_t length, uint8_t *plainText, uint32_t timeout)

    Performs the AES-128 decryption in ECB mode.

- status_t CSEC_DRV_EncryptCBC (csec_key_id_t keyId, const uint8_t *plainText, uint32_t length, const uint8_t *iv, uint8_t *cipherText, uint32_t timeout)

    Performs the AES-128 encryption in CBC mode.

- status_t CSEC_DRV_DecryptCBC (csec_key_id_t keyId, const uint8_t *cipherText, uint32_t length, const uint8_t *iv, uint8_t *plainText, uint32_t timeout)

    Performs the AES-128 decryption in CBC mode.

- status_t CSEC_DRV_GenerateMAC (csec_key_id_t keyId, const uint8_t *msg, uint32_t msgLen, uint8_↩
t *cmac, uint32_t timeout)

    Calculates the MAC of a given message using CMAC with AES-128.

- status_t CSEC_DRV_GenerateMACAddrMode (csec_key_id_t keyId, const uint8_t *msg, uint32_t msgLen, uint8_t *cmac)

    Calculates the MAC of a given message (located in Flash) using CMAC with AES-128.

- status_t CSEC_DRV_VerifyMAC (csec_key_id_t keyId, const uint8_t *msg, uint32_t msgLen, const uint8_t *mac, uint16_t macLen, bool *verifStatus, uint32_t timeout)

    Verifies the MAC of a given message using CMAC with AES-128.

- status_t CSEC_DRV_VerifyMACAddrMode (csec_key_id_t keyId, const uint8_t *msg, uint32_t msgLen, const uint8_t *mac, uint16_t macLen, bool *verifStatus)

    Verifies the MAC of a given message (located in Flash) using CMAC with AES-128.

- status_t CSEC_DRV_LoadKey (csec_key_id_t keyId, const uint8_t *m1, const uint8_t *m2, const uint8_t *m3, uint8_t *m4, uint8_t *m5)

    Updates an internal key per the SHE specification.

- status_t CSEC_DRV_LoadPlainKey (const uint8_t *plainKey)

    Updates the RAM key memory slot with a 128-bit plaintext.

- status_t CSEC_DRV_ExportRAMKey (uint8_t *m1, uint8_t *m2, uint8_t *m3, uint8_t *m4, uint8_t *m5)

    Exports the RAM_KEY into a format protected by SECRET_KEY.

- status_t CSEC_DRV_InitRNG (void)

    Initializes the seed and derives a key for the PRNG.

- status_t CSEC_DRV_ExtendSeed (const uint8_t *entropy)

    Extends the seed of the PRNG.

- status_t CSEC_DRV_GenerateRND (uint8_t *rnd)

    Generates a vector of 128 random bits.

- status_t CSEC_DRV_BootFailure (void)

    Signals a failure detected during later stages of the boot process.

- status_t CSEC_DRV_BootOK (void)

    Marks a successful boot verification during later stages of the boot process.
- status_t CSEC_DRV_BootDefine (uint32_t bootSize, csec_boot_flavor_t bootFlavor)

    Implements an extension of the SHE standard to define both the user boot size and boot method.
- static csec_status_t CSEC_DRV_GetStatus (void)

    Returns the content of the status register.
- status_t CSEC_DRV_GetID (const uint8_t ∗challenge, uint8_t ∗uid, uint8_t ∗sreg, uint8_t ∗mac)

    Returns the identity (UID) and the value of the status register protected by a MAC over a challenge and the data.
- status_t CSEC_DRV_DbgChal (uint8_t ∗challenge)

    Obtains a random number which the user shall use along with the MASTER_ECU_KEY and UID to return an authorization request.
- status_t CSEC_DRV_DbgAuth (const uint8_t ∗authorization)

    Erases all keys (actual and outdated) stored in NVM Memory if the authorization is confirmed by CSEc.
- status_t CSEC_DRV_MPCompress (const uint8_t ∗msg, uint16_t msgLen, uint8_t ∗mpCompress, uint32_t timeout)

    Compresses the given messages by accessing the Miyaguchi-Prenell compression feature with in the CSEc feature set.
- status_t CSEC_DRV_EncryptECBAsync (csec_key_id_t keyId, const uint8_t ∗plainText, uint32_t length, uint8_t ∗cipherText)

    Asynchronously performs the AES-128 encryption in ECB mode.
- status_t CSEC_DRV_DecryptECBAsync (csec_key_id_t keyId, const uint8_t ∗cipherText, uint32_t length, uint8_t ∗plainText)

    Asynchronously performs the AES-128 decryption in ECB mode.
- status_t CSEC_DRV_EncryptCBCAsync (csec_key_id_t keyId, const uint8_t ∗plainText, uint32_t length, const uint8_t ∗iv, uint8_t ∗cipherText)

    Asynchronously performs the AES-128 encryption in CBC mode.
- status_t CSEC_DRV_DecryptCBCAsync (csec_key_id_t keyId, const uint8_t ∗cipherText, uint32_t length, const uint8_t ∗iv, uint8_t ∗plainText)

    Asynchronously performs the AES-128 decryption in CBC mode.
- status_t CSEC_DRV_GenerateMACAsync (csec_key_id_t keyId, const uint8_t ∗msg, uint32_t msgLen, uint8_t ∗cmac)

    Asynchronously calculates the MAC of a given message using CMAC with AES-128.
- status_t CSEC_DRV_VerifyMACAsync (csec_key_id_t keyId, const uint8_t ∗msg, uint32_t msgLen, const uint8_t ∗mac, uint16_t macLen, bool ∗verifStatus)

    Asynchronously verifies the MAC of a given message using CMAC with AES-128.
- status_t CSEC_DRV_GetAsyncCmdStatus (void)

    Checks the status of the execution of an asynchronous command.
- void CSEC_DRV_InstallCallback (security_callback_t callbackFunc, void ∗callbackParam)

    Installs a callback function which will be invoked when an asynchronous command finishes its execution.
- void CSEC_DRV_CancelCommand (void)

    Cancels a previously launched asynchronous command.

16.4.2 Data Structure Documentation

16.4.2.1 struct csec_state_t

Internal driver state information.

Note

> The contents of this structure are internal to the driver and should not be modified by users. Also, contents of the structure are subject to change in future releases.

Implements : csec_state_t_Class

Definition at line 186 of file csec_driver.h.

Data Fields

- bool cmdInProgress
- csec_cmd_t cmd
- const uint8_t ∗ inputBuff
- uint8_t ∗ outputBuff
- uint32_t index
- uint32_t fullSize
- uint32_t partSize
- csec_key_id_t keyId
- status_t errCode
- const uint8_t ∗ iv
- csec_call_sequence_t seq
- uint32_t msgLen
- bool ∗ verifStatus
- bool macWritten
- const uint8_t ∗ mac
- uint32_t macLen
- security_callback_t callback
- void ∗ callbackParam

Field Documentation

16.4.2.1.1   security_callback_t callback

The callback invoked when an asynchronous command is completed

Definition at line 203 of file csec_driver.h.

16.4.2.1.2   void∗ callbackParam

User parameter for the command completion callback

Definition at line 204 of file csec_driver.h.

16.4.2.1.3   csec_cmd_t cmd

Specifies the type of the command in execution

Definition at line 188 of file csec_driver.h.

16.4.2.1.4   bool cmdInProgress

Specifies if a command is in progress

Definition at line 187 of file csec_driver.h.

16.4.2.1.5   status_t errCode

Specifies the error code of the last executed command

Definition at line 195 of file csec_driver.h.

16.4.2.1.6   uint32_t fullSize

Specifies the size of the input of the command in execution

Definition at line 192 of file csec_driver.h.

16.4.2.1.7   uint32_t index

Specifies the index in the input buffer of the command in execution

Definition at line 191 of file csec_driver.h.

16.4.2.1.8   const uint8_t∗ inputBuff

Specifies the input of the command in execution

Definition at line 189 of file csec_driver.h.

16.4.2.1.9   const uint8_t∗ iv

Specifies the IV of the command in execution (for encryption/decryption using CBC mode)

Definition at line 196 of file csec_driver.h.

16.4.2.1.10   csec_key_id_t keyId

Specifies the key used for the command in execution

Definition at line 194 of file csec_driver.h.

16.4.2.1.11   const uint8_t∗ mac

Specifies the MAC to be verified for a MAC verification command

Definition at line 201 of file csec_driver.h.

16.4.2.1.12   uint32_t macLen

Specifies the number of bits of the MAC to be verified for a MAC verification command

Definition at line 202 of file csec_driver.h.

16.4.2.1.13   bool macWritten

Specifies if the MAC to be verified was written in CSE_PRAM for a MAC verification command

Definition at line 200 of file csec_driver.h.

16.4.2.1.14   uint32_t msgLen

Specifies the message size (in bits) for the command in execution (for MAC generation/verification)

Definition at line 198 of file csec_driver.h.

16.4.2.1.15   uint8_t∗ outputBuff

Specifies the output of the command in execution

Definition at line 190 of file csec_driver.h.

16.4.2.1.16   uint32_t partSize

Specifies the size of the chunck of the input currently processed

Definition at line 193 of file csec_driver.h.

16.4.2.1.17   csec_call_sequence_t seq

Specifies if the information is the first or a following function call.

Definition at line 197 of file csec_driver.h.

16.4.2.1.18   bool∗ verifStatus

Specifies the result of the last executed MAC verification command

Definition at line 199 of file csec_driver.h.

16.4.3 Macro Definition Documentation

16.4.3.1 #define CSEC_STATUS_BOOT_FINISHED (0x8U)

The bit is set when the secure booting has been finished by calling either CMD_BOOT_FAILURE or CMD_BOO$\leftarrow$
T_OK or if CMD_SECURE_BOOT failed in verifying BOOT_MAC.

Definition at line 73 of file csec_driver.h.

16.4.3.2 #define CSEC_STATUS_BOOT_INIT (0x4U)

The bit is set if the secure booting has been personalized during the boot sequence.

Definition at line 69 of file csec_driver.h.

16.4.3.3 #define CSEC_STATUS_BOOT_OK (0x10U)

The bit is set if the secure booting (CMD_SECURE_BOOT) succeeded. If CMD_BOOT_FAILURE is called the bit
is erased.

Definition at line 76 of file csec_driver.h.

16.4.3.4 #define CSEC_STATUS_BUSY (0x1U)

The bit is set whenever SHE is processing a command.

Definition at line 64 of file csec_driver.h.

16.4.3.5 #define CSEC_STATUS_EXT_DEBUGGER (0x40U)

The bit is set if an external debugger is connected to the chip.

Definition at line 80 of file csec_driver.h.

16.4.3.6 #define CSEC_STATUS_INT_DEBUGGER (0x80U)

The bit is set if the internal debugging mechanisms of SHE are activated.

Definition at line 83 of file csec_driver.h.

16.4.3.7 #define CSEC_STATUS_RND_INIT (0x20U)

The bit is set if the random number generator has been initialized.

Definition at line 78 of file csec_driver.h.

16.4.3.8 #define CSEC_STATUS_SECURE_BOOT (0x2U)

The bit is set if the secure booting is activated.

Definition at line 66 of file csec_driver.h.

16.4.4 Typedef Documentation

16.4.4.1 typedef uint8_t csec_status_t

Represents the status of the CSEc module. Provides one bit for each status code as per SHE specification. CSE$\leftarrow$
C_STATUS_$*$ masks can be used for verifying the status.

Implements : csec_status_t_Class

Definition at line 92 of file csec_driver.h.

16.4.5 Enumeration Type Documentation

16.4.5.1 enum csec_boot_flavor_t

Specifies the boot type for the BOOT_DEFINE command.

Implements : csec_boot_flavor_t_Class

Enumerator

> *CSEC_BOOT_STRICT*
> *CSEC_BOOT_SERIAL*
> *CSEC_BOOT_PARALLEL*
> *CSEC_BOOT_NOT_DEFINED*

Definition at line 170 of file csec_driver.h.

16.4.5.2 enum csec_call_sequence_t

Specifies if the information is the first or a following function call.

Implements : csec_call_sequence_t_Class

Enumerator

> *CSEC_CALL_SEQ_FIRST*
> *CSEC_CALL_SEQ_SUBSEQUENT*

Definition at line 160 of file csec_driver.h.

16.4.5.3 enum csec_cmd_t

CSEc commands which follow the same values as the SHE command definition.

Implements : csec_cmd_t_Class

Enumerator

> *CSEC_CMD_ENC_ECB*
> *CSEC_CMD_ENC_CBC*
> *CSEC_CMD_DEC_ECB*
> *CSEC_CMD_DEC_CBC*
> *CSEC_CMD_GENERATE_MAC*
> *CSEC_CMD_VERIFY_MAC*
> *CSEC_CMD_LOAD_KEY*
> *CSEC_CMD_LOAD_PLAIN_KEY*
> *CSEC_CMD_EXPORT_RAM_KEY*
> *CSEC_CMD_INIT_RNG*
> *CSEC_CMD_EXTEND_SEED*
> *CSEC_CMD_RND*
> *CSEC_CMD_RESERVED_1*
> *CSEC_CMD_BOOT_FAILURE*
> *CSEC_CMD_BOOT_OK*
> *CSEC_CMD_GET_ID*
> *CSEC_CMD_BOOT_DEFINE*
> *CSEC_CMD_DBG_CHAL*

> *CSEC_CMD_DBG_AUTH*
>
> *CSEC_CMD_RESERVED_2*
>
> *CSEC_CMD_RESERVED_3*
>
> *CSEC_CMD_MP_COMPRESS*

Definition at line 130 of file csec_driver.h.

16.4.5.4   enum csec_key_id_t

Specify the KeyID to be used to implement the requested cryptographic operation.

Implements : csec_key_id_t_Class

Enumerator

> *CSEC_SECRET_KEY*
>
> *CSEC_MASTER_ECU*
>
> *CSEC_BOOT_MAC_KEY*
>
> *CSEC_BOOT_MAC*
>
> *CSEC_KEY_1*
>
> *CSEC_KEY_2*
>
> *CSEC_KEY_3*
>
> *CSEC_KEY_4*
>
> *CSEC_KEY_5*
>
> *CSEC_KEY_6*
>
> *CSEC_KEY_7*
>
> *CSEC_KEY_8*
>
> *CSEC_KEY_9*
>
> *CSEC_KEY_10*
>
> *CSEC_RAM_KEY*
>
> *CSEC_KEY_11*
>
> *CSEC_KEY_12*
>
> *CSEC_KEY_13*
>
> *CSEC_KEY_14*
>
> *CSEC_KEY_15*
>
> *CSEC_KEY_16*
>
> *CSEC_KEY_17*

Definition at line 100 of file csec_driver.h.

16.4.6   Function Documentation

16.4.6.1   status_t CSEC_DRV_BootDefine (uint32_t *bootSize,* csec_boot_flavor_t *bootFlavor*)

Implements an extension of the SHE standard to define both the user boot size and boot method.

The function implements an extension of the SHE standard to define both the user boot size and boot method.

Parameters

in	*bootSize*	Number of blocks of 128-bit data to check on boot. Maximum size is 512k↩ Bytes.
in	*bootFlavor*	The boot method.

Returns

> Error Code after command execution.

Definition at line 929 of file csec_driver.c.

16.4.6.2 status_t CSEC_DRV_BootFailure (void)

Signals a failure detected during later stages of the boot process.

The function is called during later stages of the boot process to detect a failure.

Returns

> Error Code after command execution.

Definition at line 861 of file csec_driver.c.

16.4.6.3 status_t CSEC_DRV_BootOK (void)

Marks a successful boot verification during later stages of the boot process.

The function is called during later stages of the boot process to mark successful boot verification.

Returns

> Error Code after command execution.

Definition at line 895 of file csec_driver.c.

16.4.6.4 void CSEC_DRV_CancelCommand (void)

Cancels a previously launched asynchronous command.

Definition at line 1778 of file csec_driver.c.

16.4.6.5 status_t CSEC_DRV_DbgAuth (const uint8_t ∗ authorization)

Erases all keys (actual and outdated) stored in NVM Memory if the authorization is confirmed by CSEc.

This function erases all keys (actual and outdated) stored in NVM Memory if the authorization is confirmed by CSEc.

Parameters

in	*authorization*	Pointer to the 128-bit buffer containing the authorization value.

Returns

> Error Code after command execution.

Definition at line 1060 of file csec_driver.c.

16.4.6.6 status_t CSEC_DRV_DbgChal (uint8_t ∗ challenge)

Obtains a random number which the user shall use along with the MASTER_ECU_KEY and UID to return an authorization request.

This function obtains a random number which the user shall use along with the MASTER_ECU_KEY and UID to return an authorization request.

Parameters

out	*challenge*	Pointer to the 128-bit buffer where the challenge data will be stored.

Returns

Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 1020 of file csec_driver.c.

16.4.6.7 status_t CSEC_DRV_DecryptCBC (csec_key_id_t *keyId,* const uint8_t ∗ *cipherText,* uint32_t *length,* const uint8_t ∗ *iv,* uint8_t ∗ *plainText,* uint32_t *timeout*)

Performs the AES-128 decryption in CBC mode.

This function performs the AES-128 decryption in CBC mode of the input cipher text buffer.

Parameters

in	*keyId*	KeyID used to perform the cryptographic operation.
in	*cipherText*	Pointer to the cipher text buffer.
in	*length*	Number of bytes of cipher text message to be decrypted. It should be multiple of 16 bytes.
in	*iv*	Pointer to the initialization vector buffer.
out	*plainText*	Pointer to the plain text buffer. The buffer shall have the same size as the cipher text buffer.
in	*timeout*	Timeout in milliseconds.

Returns

Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 330 of file csec_driver.c.

16.4.6.8 status_t CSEC_DRV_DecryptCBCAsync (csec_key_id_t *keyId,* const uint8_t ∗ *cipherText,* uint32_t *length,* const uint8_t ∗ *iv,* uint8_t ∗ *plainText*)

Asynchronously performs the AES-128 decryption in CBC mode.

This function performs the AES-128 decryption in CBC mode of the input cipher text buffer, in an asynchronous manner.

Parameters

in	*keyId*	KeyID used to perform the cryptographic operation.
in	*cipherText*	Pointer to the cipher text buffer.
in	*length*	Number of bytes of cipher text message to be decrypted. It should be multiple of 16 bytes.
in	*iv*	Pointer to the initialization vector buffer.
out	*plainText*	Pointer to the plain text buffer. The buffer shall have the same size as the cipher text buffer.

Returns

STATUS_SUCCESS if the command was successfully launched, STATUS_BUSY if another command was already launched. CSEC_DRV_GetAsyncCmdStatus can be used in order to check the execution status.

Definition at line 1280 of file csec_driver.c.

16.4.6.9 status_t CSEC_DRV_DecryptECB (csec_key_id_t *keyId,* const uint8_t ∗ *cipherText,* uint32_t *length,* uint8_t ∗ *plainText,* uint32_t *timeout*)

Performs the AES-128 decryption in ECB mode.

This function performs the AES-128 decryption in ECB mode of the input cipher text buffer.

Parameters

in	*keyId*	KeyID used to perform the cryptographic operation
in	*cipherText*	Pointer to the cipher text buffer.
in	*length*	Number of bytes of cipher text message to be decrypted. It should be multiple of 16 bytes.
out	*plainText*	Pointer to the plain text buffer. The buffer shall have the same size as the cipher text buffer.
in	*timeout*	Timeout in milliseconds.

Returns

Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 222 of file csec_driver.c.

16.4.6.10 status_t CSEC_DRV_DecryptECBAsync (csec_key_id_t *keyId,* const uint8_t ∗ *cipherText,* uint32_t *length,* uint8_t ∗ *plainText*)

Asynchronously performs the AES-128 decryption in ECB mode.

This function performs the AES-128 decryption in ECB mode of the input cipher text buffer, in an asynchronous manner.

Parameters

in	*keyId*	KeyID used to perform the cryptographic operation
in	*cipherText*	Pointer to the cipher text buffer.
in	*length*	Number of bytes of cipher text message to be decrypted. It should be multiple of 16 bytes.
out	*plainText*	Pointer to the plain text buffer. The buffer shall have the same size as the cipher text buffer.

Returns

STATUS_SUCCESS if the command was successfully launched, STATUS_BUSY if another command was already launched. CSEC_DRV_GetAsyncCmdStatus can be used in order to check the execution status.

Definition at line 1213 of file csec_driver.c.

16.4.6.11 void CSEC_DRV_Deinit (void)

Clears the internal state of the driver and disables the FTFC interrupt.

Definition at line 154 of file csec_driver.c.

16.4.6.12 status_t CSEC_DRV_EncryptCBC (csec_key_id_t *keyId,* const uint8_t ∗ *plainText,* uint32_t *length,* const uint8_t ∗ *iv,* uint8_t ∗ *cipherText,* uint32_t *timeout*)

Performs the AES-128 encryption in CBC mode.

This function performs the AES-128 encryption in CBC mode of the input plaintext buffer.

Parameters

in	*keyId*	KeyID used to perform the cryptographic operation.
in	*plainText*	Pointer to the plain text buffer.
in	*length*	Number of bytes of plain text message to be encrypted. It should be multiple of 16 bytes.

in	*iv*	Pointer to the initialization vector buffer.
in	*timeout*	Timeout in milliseconds.
out	*cipherText*	Pointer to the cipher text buffer. The buffer shall have the same size as the plain text buffer.
in	*timeout*	Timeout in milliseconds.

Returns

Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 274 of file csec_driver.c.

16.4.6.13 status_t CSEC_DRV_EncryptCBCAsync (csec_key_id_t *keyId,* const uint8_t ∗ *plainText,* uint32_t *length,* const uint8_t ∗ *iv,* uint8_t ∗ *cipherText*)

Asynchronously performs the AES-128 encryption in CBC mode.

This function performs the AES-128 encryption in CBC mode of the input plaintext buffer, in an asynchronous manner.

Parameters

in	*keyId*	KeyID used to perform the cryptographic operation.
in	*plainText*	Pointer to the plain text buffer.
in	*length*	Number of bytes of plain text message to be encrypted. It should be multiple of 16 bytes.
in	*iv*	Pointer to the initialization vector buffer.
out	*cipherText*	Pointer to the cipher text buffer. The buffer shall have the same size as the plain text buffer.

Returns

STATUS_SUCCESS if the command was successfully launched, STATUS_BUSY if another command was already launched. CSEC_DRV_GetAsyncCmdStatus can be used in order to check the execution status.

Definition at line 1245 of file csec_driver.c.

16.4.6.14 status_t CSEC_DRV_EncryptECB (csec_key_id_t *keyId,* const uint8_t ∗ *plainText,* uint32_t *length,* uint8_t ∗ *cipherText,* uint32_t *timeout*)

Performs the AES-128 encryption in ECB mode.

This function performs the AES-128 encryption in ECB mode of the input plain text buffer

Parameters

in	*keyId*	KeyID used to perform the cryptographic operation.
in	*plainText*	Pointer to the plain text buffer.
in	*length*	Number of bytes of plain text message to be encrypted. It should be multiple of 16 bytes.
out	*cipherText*	Pointer to the cipher text buffer. The buffer shall have the same size as the plain text buffer.
in	*timeout*	Timeout in milliseconds.

Returns

Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 169 of file csec_driver.c.

16.4.6.15   status_t CSEC_DRV_EncryptECBAsync (csec_key_id_t *keyId,* const uint8_t ∗ *plainText,* uint32_t *length,* uint8_t ∗ *cipherText*)

Asynchronously performs the AES-128 encryption in ECB mode.

This function performs the AES-128 encryption in ECB mode of the input plain text buffer, in an asynchronous manner.

Parameters

in	keyId	KeyID used to perform the cryptographic operation.
in	plainText	Pointer to the plain text buffer.
in	length	Number of bytes of plain text message to be encrypted. It should be multiple of 16 bytes.
out	cipherText	Pointer to the cipher text buffer. The buffer shall have the same size as the plain text buffer.

Returns

> STATUS_SUCCESS if the command was successfully launched, STATUS_BUSY if another command was already launched. CSEC_DRV_GetAsyncCmdStatus can be used in order to check the execution status.

Definition at line 1181 of file csec_driver.c.

16.4.6.16   status_t CSEC_DRV_ExportRAMKey (uint8_t ∗ *m1,* uint8_t ∗ *m2,* uint8_t ∗ *m3,* uint8_t ∗ *m4,* uint8_t ∗ *m5*)

Exports the RAM_KEY into a format protected by SECRET_KEY.

This function exports the RAM_KEY into a format protected by SECRET_KEY.

Parameters

out	m1	Pointer to a buffer where the M1 parameter will be exported.
out	m2	Pointer to a buffer where the M2 parameter will be exported.
out	m3	Pointer to a buffer where the M3 parameter will be exported.
out	m4	Pointer to a buffer where the M4 parameter will be exported.
out	m5	Pointer to a buffer where the M5 parameter will be exported.

Returns

> Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 694 of file csec_driver.c.

16.4.6.17   status_t CSEC_DRV_ExtendSeed (const uint8_t ∗ *entropy*)

Extends the seed of the PRNG.

Extends the seed of the PRNG by compressing the former seed value and the supplied entropy into a new seed. This new seed is then to be used to generate a random number by invoking the CMD_RND command. The random number generator must be initialized by CMD_INIT_RNG before the seed may be extended.

Parameters

in	entropy	Pointer to a 128-bit buffer containing the entropy.

Returns

> Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 783 of file csec_driver.c.

16.4.6.18   status_t CSEC_DRV_GenerateMAC (csec_key_id_t *keyId,* **const uint8_t** ∗ *msg,* **uint32_t** *msgLen,* **uint8_t** ∗ *cmac,*
uint32_t *timeout* **)**

Calculates the MAC of a given message using CMAC with AES-128.

This function calculates the MAC of a given message using CMAC with AES-128.

Parameters

in	keyId	KeyID used to perform the cryptographic operation.
in	msg	Pointer to the message buffer.
in	msgLen	Number of bits of message on which CMAC will be computed.
out	cmac	Pointer to the buffer containing the result of the CMAC computation.
in	timeout	Timeout in milliseconds.

Returns

Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 386 of file csec_driver.c.

16.4.6.19   status_t CSEC_DRV_GenerateMACAddrMode (csec_key_id_t *keyId,* **const uint8_t** ∗ *msg,* **uint32_t** *msgLen,*
uint8_t ∗ *cmac* **)**

Calculates the MAC of a given message (located in Flash) using CMAC with AES-128.

This function calculates the MAC of a given message using CMAC with AES-128. It is different from the CSEC_↩
DRV_GenerateMAC function in the sense that it does not involve an extra copy of the data on which the CMAC is
computed and the message pointer should be a pointer to Flash memory.

Parameters

in	keyId	KeyID used to perform the cryptographic operation.
in	msg	Pointer to the message buffer (pointing to Flash memory).
in	msgLen	Number of bits of message on which CMAC will be computed.
out	cmac	Pointer to the buffer containing the result of the CMAC computation.

Returns

Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 442 of file csec_driver.c.

16.4.6.20   status_t CSEC_DRV_GenerateMACAsync (csec_key_id_t *keyId,* **const uint8_t** ∗ *msg,* **uint32_t** *msgLen,* **uint8_t** ∗
cmac **)**

Asynchronously calculates the MAC of a given message using CMAC with AES-128.

This function calculates the MAC of a given message using CMAC with AES-128, in an asynchronous manner.

Parameters

in	keyId	KeyID used to perform the cryptographic operation.
in	msg	Pointer to the message buffer.
in	msgLen	Number of bits of message on which CMAC will be computed.
out	cmac	Pointer to the buffer containing the result of the CMAC computation.

Returns

STATUS_SUCCESS if the command was successfully launched, STATUS_BUSY if another command was
already launched. CSEC_DRV_GetAsyncCmdStatus can be used in order to check the execution status.

Definition at line 1315 of file csec_driver.c.

16.4.6.21   status_t CSEC_DRV_GenerateRND (uint8_t ∗ *rnd*)

Generates a vector of 128 random bits.

The function returns a vector of 128 random bits. The random number generator has to be initialized by calling CSEC_DRV_InitRNG before random numbers can be supplied.

Parameters

out	*rnd*	Pointer to a 128-bit buffer where the generated random number has to be stored.

Returns

> Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 821 of file csec_driver.c.

16.4.6.22   status_t CSEC_DRV_GetAsyncCmdStatus (void)

Checks the status of the execution of an asynchronous command.

This function checks the status of the execution of an asynchronous command. If the command is still in progress, returns STATUS_BUSY.

Returns

> Error Code after command execution.

Definition at line 1388 of file csec_driver.c.

16.4.6.23   status_t CSEC_DRV_GetID (const uint8_t ∗ *challenge,* uint8_t ∗ *uid,* uint8_t ∗ *sreg,* uint8_t ∗ *mac*)

Returns the identity (UID) and the value of the status register protected by a MAC over a challenge and the data.

This function returns the identity (UID) and the value of the status register protected by a MAC over a challenge and the data.

Parameters

in	*challenge*	Pointer to the 128-bit buffer containing Challenge data.
out	*uid*	Pointer to 120 bit buffer where the UID will be stored.
out	*sreg*	Value of the status register.
out	*mac*	Pointer to the 128 bit buffer where the MAC generated over challenge and UID and status will be stored.

Returns

> Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 968 of file csec_driver.c.

16.4.6.24   static csec_status_t CSEC_DRV_GetStatus (void) `[inline]`,`[static]`

Returns the content of the status register.

The function shall return the content of the status register.

Returns

> Value of the status register.

Implements : CSEC_DRV_GetStatus_Activity

Definition at line 529 of file csec_driver.h.

16.4.6.25   void CSEC_DRV_Init (csec_state_t ∗ *state*)

Initializes the internal state of the driver and enables the FTFC interrupt.

Parameters

in		*state*	Pointer to the state structure which will be used for holding the internal state of the driver.

Definition at line 134 of file csec_driver.c.

16.4.6.26   status_t CSEC_DRV_InitRNG (void)

Initializes the seed and derives a key for the PRNG.

The function initializes the seed and derives a key for the PRNG. The function must be called before CMD_RND after every power cycle/reset.

Returns

Error Code after command execution.

Definition at line 746 of file csec_driver.c.

16.4.6.27   void CSEC_DRV_InstallCallback (security_callback_t *callbackFunc,* void ∗ *callbackParam*)

Installs a callback function which will be invoked when an asynchronous command finishes its execution.

Parameters

in	*callbackFunc*	The function to be invoked.
in	*callbackParam*	The parameter to be passed to the callback function.

Definition at line 1763 of file csec_driver.c.

16.4.6.28   status_t CSEC_DRV_LoadKey (csec_key_id_t *keyId,* const uint8_t ∗ *m1,* const uint8_t ∗ *m2,* const uint8_t ∗ *m3,* uint8_t ∗ *m4,* uint8_t ∗ *m5*)

Updates an internal key per the SHE specification.

This function updates an internal key per the SHE specification.

Parameters

in	*keyId*	KeyID of the key to be updated.
in	*m1*	Pointer to the 128-bit M1 message containing the UID, Key ID and Authentication Key ID.
in	*m2*	Pointer to the 256-bit M2 message contains the new security flags, counter and the key value all encrypted using a derived key generated from the Authentication Key.
in	*m3*	Pointer to the 128-bit M3 message is a MAC generated over messages M1 and M2.
out	*m4*	Pointer to a 256 bits buffer where the computed M4 parameter is stored.
out	*m5*	Pointer to a 128 bits buffer where the computed M5 parameters is stored.

Returns

Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 603 of file csec_driver.c.

16.4.6.29   status_t CSEC_DRV_LoadPlainKey (const uint8_t ∗ *plainKey*)

Updates the RAM key memory slot with a 128-bit plaintext.

The function updates the RAM key memory slot with a 128-bit plaintext. The key is loaded without encryption and verification of the key, i.e. the key is handed over in plaintext. A plain key can only be loaded into the RAM_KEY slot.

Parameters

in	plainKey	Pointer to the 128-bit buffer containing the key that needs to be copied in R↩AM_KEY slot.

Returns

>   Error Code after command execution.

Definition at line 657 of file csec_driver.c.

16.4.6.30   status_t CSEC_DRV_MPCompress (const uint8_t ∗ *msg,* uint16_t *msgLen,* uint8_t ∗ *mpCompress,* uint32_t *timeout*)

Compresses the given messages by accessing the Miyaguchi-Prenell compression feature with in the CSEc feature set.

This function accesses a Miyaguchi-Prenell compression feature within the CSEc feature set to compress the given messages.

Parameters

in	msg	Pointer to the messages to be compressed. Messages must be pre-processed per SHE specification if they do not already meet the full 128-bit block size requirement.
in	msgLen	The number of 128 bit messages to be compressed.
out	mpCompress	Pointer to the 128 bit buffer storing the compressed data.
in	timeout	Timeout in milliseconds.

Returns

>   Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 1097 of file csec_driver.c.

16.4.6.31   status_t CSEC_DRV_VerifyMAC (csec_key_id_t *keyId,* const uint8_t ∗ *msg,* uint32_t *msgLen,* const uint8_t ∗ *mac,* uint16_t *macLen,* bool ∗ *verifStatus,* uint32_t *timeout*)

Verifies the MAC of a given message using CMAC with AES-128.

This function verifies the MAC of a given message using CMAC with AES-128.

Parameters

in	keyId	KeyID used to perform the cryptographic operation.
in	msg	Pointer to the message buffer.
in	msgLen	Number of bits of message on which CMAC will be computed.
in	mac	Pointer to the buffer containing the CMAC to be verified.
in	macLen	Number of bits of the CMAC to be compared. A macLength value of zero indicates that all 128-bits are compared.
out	verifStatus	Status of MAC verification command (true: verification operation passed, false: verification operation failed).
in	timeout	Timeout in milliseconds.

Returns

>   Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 487 of file csec_driver.c.

16.4.6.32   status_t CSEC_DRV_VerifyMACAddrMode (csec_key_id_t *keyId,* **const uint8_t** ∗ *msg,* **uint32_t** *msgLen,* **const uint8_t** ∗ *mac,* **uint16_t** *macLen,* **bool** ∗ *verifStatus* **)**

Verifies the MAC of a given message (located in Flash) using CMAC with AES-128.

This function verifies the MAC of a given message using CMAC with AES-128. It is different from the CSEC_DRV↩
_VerifyMAC function in the sense that it does not involve an extra copy of the data on which the CMAC is computed and the message pointer should be a pointer to Flash memory.

Parameters

in	*keyId*	KeyID used to perform the cryptographic operation.
in	*msg*	Pointer to the message buffer (pointing to Flash memory).
in	*msgLen*	Number of bits of message on which CMAC will be computed.
in	*mac*	Pointer to the buffer containing the CMAC to be verified.
in	*macLen*	Number of bits of the CMAC to be compared. A macLength value of zero indicates that all 128-bits are compared.
out	*verifStatus*	Status of MAC verification command (true: verification operation passed, false: verification operation failed).

Returns

Error Code after command execution. Output parameters are valid if the error code is STATUS_SUCCESS.

Definition at line 550 of file csec_driver.c.

16.4.6.33   status_t CSEC_DRV_VerifyMACAsync (csec_key_id_t *keyId,* **const uint8_t** ∗ *msg,* **uint32_t** *msgLen,* **const uint8_t** ∗ *mac,* **uint16_t** *macLen,* **bool** ∗ *verifStatus* **)**

Asynchronously verifies the MAC of a given message using CMAC with AES-128.

This function verifies the MAC of a given message using CMAC with AES-128, in an asynchronous manner.

Parameters

in	*keyId*	KeyID used to perform the cryptographic operation.
in	*msg*	Pointer to the message buffer.
in	*msgLen*	Number of bits of message on which CMAC will be computed.
in	*mac*	Pointer to the buffer containing the CMAC to be verified.
in	*macLen*	Number of bits of the CMAC to be compared. A macLength value of zero indicates that all 128-bits are compared.
out	*verifStatus*	Status of MAC verification command (true: verification operation passed, false: verification operation failed).

Returns

STATUS_SUCCESS if the command was successfully launched, STATUS_BUSY if another command was already launched. CSEC_DRV_GetAsyncCmdStatus can be used in order to check the execution status.

Definition at line 1348 of file csec_driver.c.

16.5 Clock Manager

16.5.1 Detailed Description

This module covers the clock management API and clock related functionality.

This section describes the programming interface of the clock_manager driver. Clock_manager achieves its functionality by configuring the hardware modules involved in clock distribution and management.

Notes

Current implementation assumes that the clock configurations are valid and are applied in a valid sequence. Mainly this means that the configuration doesn't reinitialize the clock used as the system clock.

Code Example

This is an example for switching between two configurations:

```
CLOCK_SYS_Init(g_clockManConfigsArr,
            CLOCK_MANAGER_CONFIG_CNT,
            g_clockManCallbacksArr,
            CLOCK_MANAGER_CALLBACK_CNT);

CLOCK_SYS_UpdateConfiguration(0,
      CLOCK_MANAGER_POLICY_FORCIBLE);
CLOCK_SYS_UpdateConfiguration(1,
      CLOCK_MANAGER_POLICY_FORCIBLE);
```

Modules

- Clock_manager_s32k1xx

Data Structures

- struct clock_notify_struct_t

    Clock notification structure passed to clock callback function. Implements clock_notify_struct_t_Class. More...
- struct clock_manager_callback_user_config_t

    Structure for callback function and its parameter. Implements clock_manager_callback_user_config_t_Class. More...
- struct clock_manager_state_t

    Clock manager state structure. Implements clock_manager_state_t_Class. More...

Typedefs

- typedef status_t(∗ clock_manager_callback_t) (clock_notify_struct_t ∗notify, void ∗callbackData)

    Type of clock callback functions.

Enumerations

- enum clock_manager_notify_t { CLOCK_MANAGER_NOTIFY_RECOVER = 0x00U, CLOCK_MANAGER↩_NOTIFY_BEFORE = 0x01U, CLOCK_MANAGER_NOTIFY_AFTER = 0x02U }

    The clock notification type. Implements clock_manager_notify_t_Class.
- enum clock_manager_callback_type_t { CLOCK_MANAGER_CALLBACK_BEFORE = 0x01U, CLOCK_M↩ANAGER_CALLBACK_AFTER = 0x02U, CLOCK_MANAGER_CALLBACK_BEFORE_AFTER = 0x03U }

    The callback type, indicates what kinds of notification this callback handles. Implements clock_manager_callback↩_type_t_Class.

- enum clock_manager_policy_t { CLOCK_MANAGER_POLICY_AGREEMENT, CLOCK_MANAGER_POL↩
  ICY_FORCIBLE }

  Clock transition policy. Implements clock_manager_policy_t_Class.

Dynamic clock setting

- status_t CLOCK_SYS_Init (clock_manager_user_config_t const ∗∗clockConfigsPtr, uint8_t configsNumber,
  clock_manager_callback_user_config_t ∗∗callbacksPtr, uint8_t callbacksNumber)

  Install pre-defined clock configurations.
- status_t CLOCK_SYS_UpdateConfiguration (uint8_t targetConfigIndex, clock_manager_policy_t policy)

  Set system clock configuration according to pre-defined structure.
- status_t CLOCK_SYS_SetConfiguration (clock_manager_user_config_t const ∗config)

  Set system clock configuration.
- uint8_t CLOCK_SYS_GetCurrentConfiguration (void)

  Get current system clock configuration.
- clock_manager_callback_user_config_t ∗ CLOCK_SYS_GetErrorCallback (void)

  Get the callback which returns error in last clock switch.
- status_t CLOCK_SYS_GetFreq (clock_names_t clockName, uint32_t ∗frequency)

  Gets the clock frequency for a specific clock name.

16.5.2 Data Structure Documentation

16.5.2.1 struct clock_notify_struct_t

Clock notification structure passed to clock callback function. Implements clock_notify_struct_t_Class.

Definition at line 114 of file clock_manager.h.

Data Fields

- uint8_t targetClockConfigIndex
- clock_manager_policy_t policy
- clock_manager_notify_t notifyType

Field Documentation

16.5.2.1.1 clock_manager_notify_t notifyType

Clock notification type.

Definition at line 118 of file clock_manager.h.

16.5.2.1.2 clock_manager_policy_t policy

Clock transition policy.

Definition at line 117 of file clock_manager.h.

16.5.2.1.3 uint8_t targetClockConfigIndex

Target clock configuration index.

Definition at line 116 of file clock_manager.h.

16.5.2.2 struct clock_manager_callback_user_config_t

Structure for callback function and its parameter. Implements clock_manager_callback_user_config_t_Class.

Definition at line 131 of file clock_manager.h.

Data Fields

- clock_manager_callback_t callback
- clock_manager_callback_type_t callbackType
- void ∗ callbackData

Field Documentation

16.5.2.2.1 clock_manager_callback_t callback

Entry of callback function.

Definition at line 133 of file clock_manager.h.

16.5.2.2.2 void∗ callbackData

Parameter of callback function.

Definition at line 135 of file clock_manager.h.

16.5.2.2.3 clock_manager_callback_type_t callbackType

Callback type.

Definition at line 134 of file clock_manager.h.

16.5.2.3 struct clock_manager_state_t

Clock manager state structure. Implements clock_manager_state_t_Class.

Definition at line 142 of file clock_manager.h.

Data Fields

- clock_manager_user_config_t const ∗∗ configTable
- uint8_t clockConfigNum
- uint8_t curConfigIndex
- clock_manager_callback_user_config_t ∗∗ callbackConfig
- uint8_t callbackNum
- uint8_t errorCallbackIndex

Field Documentation

16.5.2.3.1 clock_manager_callback_user_config_t∗∗ callbackConfig

Pointer to callback table.

Definition at line 147 of file clock_manager.h.

16.5.2.3.2 uint8_t callbackNum

Number of clock callbacks.

Definition at line 148 of file clock_manager.h.

16.5.2.3.3 uint8_t clockConfigNum

Number of clock configurations.

Definition at line 145 of file clock_manager.h.

16.5.2.3.4 clock_manager_user_config_t const∗∗ configTable

Pointer to clock configure table.

Definition at line 144 of file clock_manager.h.

16.5.2.3.5 uint8_t curConfigIndex

Index of current configuration.

Definition at line 146 of file clock_manager.h.

16.5.2.3.6 uint8_t errorCallbackIndex

Index of callback returns error.

Definition at line 149 of file clock_manager.h.

16.5.3 Typedef Documentation

16.5.3.1 typedef status_t(∗ clock_manager_callback_t) (clock_notify_struct_t ∗notify, void ∗callbackData)

Type of clock callback functions.

Definition at line 124 of file clock_manager.h.

16.5.4 Enumeration Type Documentation

16.5.4.1 enum clock_manager_callback_type_t

The callback type, indicates what kinds of notification this callback handles. Implements clock_manager_callback↩
_type_t_Class.

Enumerator

> ***CLOCK_MANAGER_CALLBACK_BEFORE*** Callback handles BEFORE notification.
>
> ***CLOCK_MANAGER_CALLBACK_AFTER*** Callback handles AFTER notification.
>
> ***CLOCK_MANAGER_CALLBACK_BEFORE_AFTER*** Callback handles BEFORE and AFTER notification

Definition at line 93 of file clock_manager.h.

16.5.4.2 enum clock_manager_notify_t

The clock notification type. Implements clock_manager_notify_t_Class.

Enumerator

> ***CLOCK_MANAGER_NOTIFY_RECOVER*** Notify IP to recover to previous work state.
>
> ***CLOCK_MANAGER_NOTIFY_BEFORE*** Notify IP that system will change clock setting.
>
> ***CLOCK_MANAGER_NOTIFY_AFTER*** Notify IP that have changed to new clock setting.

Definition at line 82 of file clock_manager.h.

16.5.4.3 enum clock_manager_policy_t

Clock transition policy. Implements clock_manager_policy_t_Class.

Enumerator

> ***CLOCK_MANAGER_POLICY_AGREEMENT*** Clock transfers gracefully.
>
> ***CLOCK_MANAGER_POLICY_FORCIBLE*** Clock transfers forcefully.

Definition at line 104 of file clock_manager.h.

16.5.5 Function Documentation

16.5.5.1 uint8_t CLOCK_SYS_GetCurrentConfiguration (void)

Get current system clock configuration.

Returns

> Current clock configuration index.

Definition at line 208 of file clock_manager.c.

16.5.5.2 clock_manager_callback_user_config_t ∗ CLOCK_SYS_GetErrorCallback (void)

Get the callback which returns error in last clock switch.

When graceful policy is used, if some IP is not ready to change clock setting, the callback will return error and system stay in current configuration. Applications can use this function to check which IP callback returns error.

Returns

> Pointer to the callback which returns error.

Definition at line 220 of file clock_manager.c.

16.5.5.3 status_t CLOCK_SYS_GetFreq (clock_names_t *clockName,* uint32_t ∗ *frequency*)

Gets the clock frequency for a specific clock name.

This function checks the current clock configurations and then calculates the clock frequency for a specific clock name defined in clock_names_t. The SCG must be properly configured before using this function. See the reference manual for supported clock names for different chip families. The returned value is in Hertz. If it cannot find the clock name or the name is not supported for a specific chip family, it returns an STATUS_UNSUPPORTED. If frequency is required for a peripheral and the module is not clocked, then STATUS_MCU_GATED_OFF status is returned. Frequency is returned if a valid address is provided. If frequency is required for a peripheral that doesn't support functional clock, the zero value is provided.

Parameters

in	*clockName*	Clock names defined in clock_names_t
out	*frequency*	Returned clock frequency value in Hertz

Returns

> status Error code defined in status_t

Definition at line 1551 of file clock_S32K1xx.c.

16.5.5.4 status_t CLOCK_SYS_Init (clock_manager_user_config_t const ∗∗ *clockConfigsPtr,* uint8_t *configsNumber,* clock_manager_callback_user_config_t ∗∗ *callbacksPtr,* uint8_t *callbacksNumber*)

Install pre-defined clock configurations.

This function installs the pre-defined clock configuration table to clock manager.

Parameters

in	*clockConfigsPtr*	Pointer to the clock configuration table.

in	*configsNumber*	Number of clock configurations in table.
in	*callbacksPtr*	Pointer to the callback configuration table.
in	*callbacks↩ Number*	Number of callback configurations in table.

Returns

>   Error code.

Definition at line 57 of file clock_manager.c.

16.5.5.5   status_t CLOCK_SYS_SetConfiguration (clock_manager_user_config_t const ∗ *config*)

Set system clock configuration.

This function sets the system to target configuration, it only sets the clock modules registers for clock mode change, but not send notifications to drivers. This function is different by different SoCs.

Parameters

in	*config*	Target configuration.

Returns

>   Error code.

Note

>   If external clock is used in the target mode, please make sure it is enabled, for example, if the external oscillator is used, please setup EREFS/HGO correctly and make sure OSCINIT is set.

Definition at line 474 of file clock_S32K1xx.c.

16.5.5.6   status_t CLOCK_SYS_UpdateConfiguration (uint8_t *targetConfigIndex,* clock_manager_policy_t *policy*)

Set system clock configuration according to pre-defined structure.

This function sets system to target clock configuration; before transition, clock manager will send notifications to all drivers registered to the callback table. When graceful policy is used, if some drivers are not ready to change, clock transition will not occur, all drivers still work in previous configuration and error is returned. When forceful policy is used, all drivers should stop work and system changes to new clock configuration.

Parameters

in	*targetConfig↩ Index*	Index of the clock configuration.
in	*policy*	Transaction policy, graceful or forceful.

Returns

>   Error code.

Note

>   If external clock is used in the target mode, please make sure it is enabled, for example, if the external oscillator is used, please setup EREFS/HGO correctly and make sure OSCINIT is set.

Definition at line 90 of file clock_manager.c.

16.6 Clock_manager_s32k1xx

16.6.1 Detailed Description

Data Structures

- struct sim_clock_out_config_t

  SIM ClockOut configuration. Implements sim_clock_out_config_t_Class. More...
- struct sim_lpo_clock_config_t

  SIM LPO Clocks configuration. Implements sim_lpo_clock_config_t_Class. More...
- struct sim_tclk_config_t

  SIM Platform Gate Clock configuration. Implements sim_tclk_config_t_Class. More...
- struct sim_plat_gate_config_t

  SIM Platform Gate Clock configuration. Implements sim_plat_gate_config_t_Class. More...
- struct sim_qspi_ref_clk_gating_t

  SIM QSPI reference clock gating. Implements sim_qspi_ref_clk_gating_t_Class. More...
- struct sim_trace_clock_config_t

  SIM Debug Trace clock configuration. Implements sim_trace_clock_config_t_Class. More...
- struct sim_clock_config_t

  SIM configure structure. Implements sim_clock_config_t_Class. More...
- struct scg_system_clock_config_t

  SCG system clock configuration. Implements scg_system_clock_config_t_Class. More...

Macros

- #define NUMBER_OF_TCLK_INPUTS 3U
- #define MAX_NO_SYS_CLOCK_DIVS 7

Enumerations

- enum sim_rtc_clk_sel_src_t { SIM_RTCCLK_SEL_SOSCDIV1_CLK = 0x0U, SIM_RTCCLK_SEL_LPO_32K = 0x1U, SIM_RTCCLK_SEL_RTC_CLKIN = 0x2U, SIM_RTCCLK_SEL_FIRCDIV1_CLK = 0x3U }

  SIM CLK32KSEL clock source select Implements sim_rtc_clk_sel_src_t_Class.
- enum sim_lpoclk_sel_src_t { SIM_LPO_CLK_SEL_LPO_128K = 0x0, SIM_LPO_CLK_SEL_NO_CLOCK = 0x1, SIM_LPO_CLK_SEL_LPO_32K = 0x2, SIM_LPO_CLK_SEL_LPO_1K = 0x3 }

  SIM LPOCLKSEL clock source select Implements sim_lpoclk_sel_src_t_Class.
- enum sim_clkout_src_t {
  SIM_CLKOUT_SEL_SYSTEM_SCG_CLKOUT = 0U, SIM_CLKOUT_SEL_SYSTEM_SOSC_DIV2_CLK = 2U, SIM_CLKOUT_SEL_SYSTEM_SIRC_DIV2_CLK = 4U, SIM_CLKOUT_SEL_SYSTEM_FIRC_DIV2_↩
  CLK = 6U,
  SIM_CLKOUT_SEL_SYSTEM_HCLK = 7U, SIM_CLKOUT_SEL_SYSTEM_SPLL_DIV2_CLK = 8U, SIM_↩
  CLKOUT_SEL_SYSTEM_BUS_CLK = 9U, SIM_CLKOUT_SEL_SYSTEM_LPO_128K_CLK = 10U,
  SIM_CLKOUT_SEL_SYSTEM_LPO_CLK = 12U, SIM_CLKOUT_SEL_SYSTEM_RTC_CLK = 14U }

  SIM CLKOUT select.
- enum sim_clkout_div_t {
  SIM_CLKOUT_DIV_BY_1 = 0x0U, SIM_CLKOUT_DIV_BY_2 = 0x1U, SIM_CLKOUT_DIV_BY_3 = 0x2U, SIM_CLKOUT_DIV_BY_4 = 0x3U,
  SIM_CLKOUT_DIV_BY_5 = 0x4U, SIM_CLKOUT_DIV_BY_6 = 0x5U, SIM_CLKOUT_DIV_BY_7 = 0x6U, SIM_CLKOUT_DIV_BY_8 = 0x7U }

  SIM CLKOUT divider.
- enum clock_trace_src_t { CLOCK_TRACE_SRC_CORE_CLK = 0x0 }

  Debug trace clock source select Implements clock_trace_src_t_Class.

- enum scg_system_clock_src_t { SCG_SYSTEM_CLOCK_SRC_SYS_OSC = 1U, SCG_SYSTEM_CLO↩
  CK_SRC_SIRC = 2U, SCG_SYSTEM_CLOCK_SRC_FIRC = 3U, SCG_SYSTEM_CLOCK_SRC_NONE =
  255U }

    SCG system clock source. Implements scg_system_clock_src_t_Class.

- enum scg_system_clock_div_t {
  SCG_SYSTEM_CLOCK_DIV_BY_1 = 0U, SCG_SYSTEM_CLOCK_DIV_BY_2 = 1U, SCG_SYSTEM_CL↩
  OCK_DIV_BY_3 = 2U, SCG_SYSTEM_CLOCK_DIV_BY_4 = 3U,
  SCG_SYSTEM_CLOCK_DIV_BY_5 = 4U, SCG_SYSTEM_CLOCK_DIV_BY_6 = 5U, SCG_SYSTEM_CL↩
  OCK_DIV_BY_7 = 6U, SCG_SYSTEM_CLOCK_DIV_BY_8 = 7U,
  SCG_SYSTEM_CLOCK_DIV_BY_9 = 8U, SCG_SYSTEM_CLOCK_DIV_BY_10 = 9U, SCG_SYSTEM_C↩
  LOCK_DIV_BY_11 = 10U, SCG_SYSTEM_CLOCK_DIV_BY_12 = 11U,
  SCG_SYSTEM_CLOCK_DIV_BY_13 = 12U, SCG_SYSTEM_CLOCK_DIV_BY_14 = 13U, SCG_SYSTE↩
  M_CLOCK_DIV_BY_15 = 14U, SCG_SYSTEM_CLOCK_DIV_BY_16 = 15U }

    SCG system clock divider value. Implements scg_system_clock_div_t_Class.

Variables

- const uint8_t peripheralFeaturesList [CLOCK_NAME_COUNT]

    Peripheral features list Constant array storing the mappings between clock names of the peripherals and feature lists.

- uint32_t g_TClkFreq [NUMBER_OF_TCLK_INPUTS]
- uint32_t g_xtal0ClkFreq
- uint32_t g_RtcClkInFreq

16.6.2   Data Structure Documentation

16.6.2.1   struct sim_clock_out_config_t

SIM ClockOut configuration. Implements sim_clock_out_config_t_Class.

Definition at line 140 of file clock_S32K1xx.h.

Data Fields

- bool initialize
- bool enable
- sim_clkout_src_t source
- sim_clkout_div_t divider

Field Documentation

16.6.2.1.1   sim_clkout_div_t divider

SIM ClockOut divide ratio.

Definition at line 145 of file clock_S32K1xx.h.

16.6.2.1.2   bool enable

SIM ClockOut enable.

Definition at line 143 of file clock_S32K1xx.h.

16.6.2.1.3   bool initialize

Initialize or not the ClockOut clock.

Definition at line 142 of file clock_S32K1xx.h.

16.6.2.1.4   sim_clkout_src_t source

SIM ClockOut source select.

Definition at line 144 of file clock_S32K1xx.h.

16.6.2.2   struct sim_lpo_clock_config_t

SIM LPO Clocks configuration. Implements sim_lpo_clock_config_t_Class.

Definition at line 153 of file clock_S32K1xx.h.

Data Fields

- bool initialize
- sim_rtc_clk_sel_src_t sourceRtcClk
- sim_lpoclk_sel_src_t sourceLpoClk
- bool enableLpo32k
- bool enableLpo1k

Field Documentation

16.6.2.2.1   bool enableLpo1k

MSCM Clock Gating Control enable.

Definition at line 159 of file clock_S32K1xx.h.

16.6.2.2.2   bool enableLpo32k

MSCM Clock Gating Control enable.

Definition at line 158 of file clock_S32K1xx.h.

16.6.2.2.3   bool initialize

Initialize or not the LPO clock.

Definition at line 155 of file clock_S32K1xx.h.

16.6.2.2.4   sim_lpoclk_sel_src_t sourceLpoClk

LPO clock source select.

Definition at line 157 of file clock_S32K1xx.h.

16.6.2.2.5   sim_rtc_clk_sel_src_t sourceRtcClk

RTC_CLK source select.

Definition at line 156 of file clock_S32K1xx.h.

16.6.2.3   struct sim_tclk_config_t

SIM Platform Gate Clock configuration. Implements sim_tclk_config_t_Class.

Definition at line 166 of file clock_S32K1xx.h.

Data Fields

- bool initialize
- uint32_t tclkFreq [NUMBER_OF_TCLK_INPUTS]

Field Documentation

16.6.2.3.1 bool initialize

Initialize or not the Trace clock.

Definition at line 168 of file clock_S32K1xx.h.

16.6.2.3.2 uint32_t tclkFreq[NUMBER_OF_TCLK_INPUTS]

TCLKx frequency.

Definition at line 169 of file clock_S32K1xx.h.

16.6.2.4 struct sim_plat_gate_config_t

SIM Platform Gate Clock configuration. Implements sim_plat_gate_config_t_Class.

Definition at line 176 of file clock_S32K1xx.h.

Data Fields

- bool initialize
- bool enableMscm
- bool enableMpu
- bool enableDma
- bool enableErm
- bool enableEim

Field Documentation

16.6.2.4.1 bool enableDma

DMA Clock Gating Control enable.

Definition at line 181 of file clock_S32K1xx.h.

16.6.2.4.2 bool enableEim

EIM Clock Gating Control enable.

Definition at line 183 of file clock_S32K1xx.h.

16.6.2.4.3 bool enableErm

ERM Clock Gating Control enable.

Definition at line 182 of file clock_S32K1xx.h.

16.6.2.4.4 bool enableMpu

MPU Clock Gating Control enable.

Definition at line 180 of file clock_S32K1xx.h.

16.6.2.4.5 bool enableMscm

MSCM Clock Gating Control enable.

Definition at line 179 of file clock_S32K1xx.h.

16.6.2.4.6 bool initialize

Initialize or not the Trace clock.

Definition at line 178 of file clock_S32K1xx.h.

16.6.2.5   struct sim_qspi_ref_clk_gating_t

SIM QSPI reference clock gating. Implements sim_qspi_ref_clk_gating_t_Class.

Definition at line 190 of file clock_S32K1xx.h.

Data Fields

- bool enableQspiRefClk

Field Documentation

16.6.2.5.1   bool enableQspiRefClk

qspi internal reference clock gating control enable.

Definition at line 192 of file clock_S32K1xx.h.

16.6.2.6   struct sim_trace_clock_config_t

SIM Debug Trace clock configuration. Implements sim_trace_clock_config_t_Class.

Definition at line 210 of file clock_S32K1xx.h.

Data Fields

- bool initialize
- bool divEnable
- clock_trace_src_t source
- uint8_t divider
- bool divFraction

Field Documentation

16.6.2.6.1   bool divEnable

Trace clock divider enable.

Definition at line 213 of file clock_S32K1xx.h.

16.6.2.6.2   bool divFraction

Trace clock divider fraction.

Definition at line 216 of file clock_S32K1xx.h.

16.6.2.6.3   uint8_t divider

Trace clock divider divisor.

Definition at line 215 of file clock_S32K1xx.h.

16.6.2.6.4   bool initialize

Initialize or not the Trace clock.

Definition at line 212 of file clock_S32K1xx.h.

16.6.2.6.5   clock_trace_src_t source

Trace clock select.

Definition at line 214 of file clock_S32K1xx.h.

16.6.2.7 struct sim_clock_config_t

SIM configure structure. Implements sim_clock_config_t_Class.

Definition at line 223 of file clock_S32K1xx.h.

Data Fields

- sim_clock_out_config_t clockOutConfig
- sim_lpo_clock_config_t lpoClockConfig
- sim_tclk_config_t tclkConfig
- sim_plat_gate_config_t platGateConfig
- sim_trace_clock_config_t traceClockConfig
- sim_qspi_ref_clk_gating_t qspiRefClkGating

Field Documentation

16.6.2.7.1 sim_clock_out_config_t clockOutConfig

Clock Out configuration.

Definition at line 225 of file clock_S32K1xx.h.

16.6.2.7.2 sim_lpo_clock_config_t lpoClockConfig

Low Power Clock configuration.

Definition at line 226 of file clock_S32K1xx.h.

16.6.2.7.3 sim_plat_gate_config_t platGateConfig

Platform Gate Clock configuration.

Definition at line 228 of file clock_S32K1xx.h.

16.6.2.7.4 sim_qspi_ref_clk_gating_t qspiRefClkGating

Qspi Reference Clock Gating.

Definition at line 230 of file clock_S32K1xx.h.

16.6.2.7.5 sim_tclk_config_t tclkConfig

Platform Gate Clock configuration.

Definition at line 227 of file clock_S32K1xx.h.

16.6.2.7.6 sim_trace_clock_config_t traceClockConfig

Trace clock configuration.

Definition at line 229 of file clock_S32K1xx.h.

16.6.2.8 struct scg_system_clock_config_t

SCG system clock configuration. Implements scg_system_clock_config_t_Class.

Definition at line 277 of file clock_S32K1xx.h.

Data Fields

- scg_system_clock_div_t divSlow
- scg_system_clock_div_t divBus
- scg_system_clock_div_t divCore
- scg_system_clock_src_t src

Field Documentation

16.6.2.8.1   scg_system_clock_div_t divBus

BUS clock divider.

Definition at line 280 of file clock_S32K1xx.h.

16.6.2.8.2   scg_system_clock_div_t divCore

Core clock divider.

Definition at line 281 of file clock_S32K1xx.h.

16.6.2.8.3   scg_system_clock_div_t divSlow

Slow clock divider.

Definition at line 279 of file clock_S32K1xx.h.

16.6.2.8.4   scg_system_clock_src_t src

System clock source.

Definition at line 282 of file clock_S32K1xx.h.

16.6.3   Macro Definition Documentation

16.6.3.1   #define MAX_NO_SYS_CLOCK_DIVS 7

Definition at line 70 of file clock_S32K1xx.h.

16.6.3.2   #define NUMBER_OF_TCLK_INPUTS 3U

Definition at line 60 of file clock_S32K1xx.h.

16.6.4   Enumeration Type Documentation

16.6.4.1   enum clock_trace_src_t

Debug trace clock source select Implements clock_trace_src_t_Class.

Enumerator

> **CLOCK_TRACE_SRC_CORE_CLK**   core clock

Definition at line 200 of file clock_S32K1xx.h.

16.6.4.2   enum scg_system_clock_div_t

SCG system clock divider value. Implements scg_system_clock_div_t_Class.

Enumerator

> **SCG_SYSTEM_CLOCK_DIV_BY_1**   Divided by 1.
> **SCG_SYSTEM_CLOCK_DIV_BY_2**   Divided by 2.
> **SCG_SYSTEM_CLOCK_DIV_BY_3**   Divided by 3.
> **SCG_SYSTEM_CLOCK_DIV_BY_4**   Divided by 4.
> **SCG_SYSTEM_CLOCK_DIV_BY_5**   Divided by 5.
> **SCG_SYSTEM_CLOCK_DIV_BY_6**   Divided by 6.

> **SCG_SYSTEM_CLOCK_DIV_BY_7** Divided by 7.
>
> **SCG_SYSTEM_CLOCK_DIV_BY_8** Divided by 8.
>
> **SCG_SYSTEM_CLOCK_DIV_BY_9** Divided by 9.
>
> **SCG_SYSTEM_CLOCK_DIV_BY_10** Divided by 10.
>
> **SCG_SYSTEM_CLOCK_DIV_BY_11** Divided by 11.
>
> **SCG_SYSTEM_CLOCK_DIV_BY_12** Divided by 12.
>
> **SCG_SYSTEM_CLOCK_DIV_BY_13** Divided by 13.
>
> **SCG_SYSTEM_CLOCK_DIV_BY_14** Divided by 14.
>
> **SCG_SYSTEM_CLOCK_DIV_BY_15** Divided by 15.
>
> **SCG_SYSTEM_CLOCK_DIV_BY_16** Divided by 16.

Definition at line 253 of file clock_S32K1xx.h.

16.6.4.3   enum scg_system_clock_src_t

SCG system clock source. Implements scg_system_clock_src_t_Class.

Enumerator

> **SCG_SYSTEM_CLOCK_SRC_SYS_OSC** System OSC.
>
> **SCG_SYSTEM_CLOCK_SRC_SIRC** Slow IRC.
>
> **SCG_SYSTEM_CLOCK_SRC_FIRC** Fast IRC.
>
> **SCG_SYSTEM_CLOCK_SRC_NONE** MAX value.

Definition at line 238 of file clock_S32K1xx.h.

16.6.4.4   enum sim_clkout_div_t

SIM CLKOUT divider.

Enumerator

> **SIM_CLKOUT_DIV_BY_1** Divided by 1
>
> **SIM_CLKOUT_DIV_BY_2** Divided by 2
>
> **SIM_CLKOUT_DIV_BY_3** Divided by 3
>
> **SIM_CLKOUT_DIV_BY_4** Divided by 4
>
> **SIM_CLKOUT_DIV_BY_5** Divided by 5
>
> **SIM_CLKOUT_DIV_BY_6** Divided by 6
>
> **SIM_CLKOUT_DIV_BY_7** Divided by 7
>
> **SIM_CLKOUT_DIV_BY_8** Divided by 8

Definition at line 123 of file clock_S32K1xx.h.

16.6.4.5   enum sim_clkout_src_t

SIM CLKOUT select.

Enumerator

> **SIM_CLKOUT_SEL_SYSTEM_SCG_CLKOUT** SCG CLKOUT
>
> **SIM_CLKOUT_SEL_SYSTEM_SOSC_DIV2_CLK** SOSC DIV2 CLK
>
> **SIM_CLKOUT_SEL_SYSTEM_SIRC_DIV2_CLK** SIRC DIV2 CLK
>
> **SIM_CLKOUT_SEL_SYSTEM_FIRC_DIV2_CLK** FIRC DIV2 CLK

> ***SIM_CLKOUT_SEL_SYSTEM_HCLK*** HCLK
>
> ***SIM_CLKOUT_SEL_SYSTEM_SPLL_DIV2_CLK*** SPLL DIV2 CLK
>
> ***SIM_CLKOUT_SEL_SYSTEM_BUS_CLK*** BUS_CLK
>
> ***SIM_CLKOUT_SEL_SYSTEM_LPO_128K_CLK*** LPO_CLK 128 Khz
>
> ***SIM_CLKOUT_SEL_SYSTEM_LPO_CLK*** LPO_CLK as selected by SIM LPO CLK Select
>
> ***SIM_CLKOUT_SEL_SYSTEM_RTC_CLK*** RTC CLK as selected by SIM CLK 32 KHz Select

Definition at line 99 of file clock_S32K1xx.h.

16.6.4.6 enum sim_lpoclk_sel_src_t

SIM LPOCLKSEL clock source select Implements sim_lpoclk_sel_src_t_Class.

Enumerator

> ***SIM_LPO_CLK_SEL_LPO_128K***
>
> ***SIM_LPO_CLK_SEL_NO_CLOCK***
>
> ***SIM_LPO_CLK_SEL_LPO_32K***
>
> ***SIM_LPO_CLK_SEL_LPO_1K***

Definition at line 88 of file clock_S32K1xx.h.

16.6.4.7 enum sim_rtc_clk_sel_src_t

SIM CLK32KSEL clock source select Implements sim_rtc_clk_sel_src_t_Class.

Enumerator

> ***SIM_RTCCLK_SEL_SOSCDIV1_CLK***
>
> ***SIM_RTCCLK_SEL_LPO_32K***
>
> ***SIM_RTCCLK_SEL_RTC_CLKIN***
>
> ***SIM_RTCCLK_SEL_FIRCDIV1_CLK***

Definition at line 76 of file clock_S32K1xx.h.

16.6.5 Variable Documentation

16.6.5.1 uint32_t g_RtcClkInFreq

Definition at line 76 of file clock_S32K1xx.c.

16.6.5.2 uint32_t g_TClkFreq[NUMBER_OF_TCLK_INPUTS]

Definition at line 73 of file clock_S32K1xx.c.

16.6.5.3 uint32_t g_xtal0ClkFreq

Definition at line 79 of file clock_S32K1xx.c.

16.6.5.4 const uint8_t peripheralFeaturesList[CLOCK_NAME_COUNT]

Peripheral features list Constant array storing the mappings between clock names of the peripherals and feature lists.

Definition at line 349 of file clock_S32K1xx.c.

16.7 Common Core API.

16.7.1 Detailed Description

This group contains general core APIs that used for both protocol LIN 2.1 and J2602.

Modules

- **Driver and cluster management**

    API perform the initialization of the LIN core.
- **Interface management**

    This group contains APIs that help users manage interface(s) in LIN node.
- **Notification**

    This group contains APIs that let users know when a signal's value changed.
- **Schedule management**

    This group contains APIs that help users manage schedule tables in master node only.
- **Signal interaction**

    This group contains APIs that help users interract with signals of LIN node.
- **User provided call-outs**

    This group contains APIs which may be called from within the LIN module in order to enable/disable LIN communication interrupts.

Macros

- #define SAVE_CONFIG_SET 0x0040U
- #define EVENT_TRIGGER_COLLISION_SET 0x0020U
- #define BUS_ACTIVITY_SET 0x0010U
- #define GO_TO_SLEEP_SET 0x0008U
- #define OVERRUN 0x0004U
- #define SUCCESSFULL_TRANSFER 0x0002U
- #define ERROR_IN_RESPONSE 0x0001U

16.7.2 Macro Definition Documentation

16.7.2.1 #define BUS_ACTIVITY_SET 0x0010U

Bus activity

Definition at line 35 of file lin_common_api.h.

16.7.2.2 #define ERROR_IN_RESPONSE 0x0001U

Error in response

Definition at line 39 of file lin_common_api.h.

16.7.2.3 #define EVENT_TRIGGER_COLLISION_SET 0x0020U

Event triggered frame collision

Definition at line 34 of file lin_common_api.h.

16.7.2.4 #define GO_TO_SLEEP_SET 0x0008U

Go to sleep

Definition at line 36 of file lin_common_api.h.

16.7.2.5   #define OVERRUN 0x0004U

Overrun

Definition at line 37 of file lin_common_api.h.

16.7.2.6   #define SAVE_CONFIG_SET 0x0040U

Save configuration

Definition at line 33 of file lin_common_api.h.

16.7.2.7   #define SUCCESSFULL_TRANSFER 0x0002U

Successful transfer

Definition at line 38 of file lin_common_api.h.

16.8 Common Transport Layer API

16.8.1 Detailed Description

Contains Transport Layer APIs that used for both protocols LIN 2.1 and J2602.

Modules

- Cooked API

    Cooked processing of diagnostic messages manages one complete message at a time.
- Initialization

    Initialize transport layer (queues, status, ...).
- Raw API

    The raw API is operating on PDU level and it is typically used to gateway PDUs between CAN and LIN.

Macros

- #define LD_READ_OK 0x33U
- #define LD_LENGTH_TOO_SHORT 0x34U
- #define LD_DATA_ERROR 0x43U
- #define LD_LENGTH_NOT_CORRECT 0x44U
- #define LD_SET_OK 0x45U
- #define SERVICE_TARGET_RESET 0xB5U
- #define RES_POSITIVE 0x40U
- #define LIN_PRODUCT_ID 0x00U
- #define LIN_SERIAL_NUMBER 0x01U
- #define LD_BROADCAST 0x7FU
- #define LD_FUNCTIONAL_NAD 0x7EU
- #define LD_ANY_SUPPLIER 0x7FFFU
- #define LD_ANY_FUNCTION 0xFFFFU
- #define LD_ANY_MESSAGE 0xFFFFU
- #define RES_NEGATIVE 0x7FU
- #define GENERAL_REJECT 0x10U
- #define SERVICE_NOT_SUPPORTED 0x11U
- #define SUBFUNCTION_NOT_SUPPORTED 0x12U
- #define NEGATIVE 0U
- #define POSITIVE 1U
- #define TRANSMITTING 0U
- #define RECEIVING 1U
- #define DIAG_SERVICE_CALLBACK_HANDLER(iii, sid) lin_diag_service_callback((iii), (sid))

Functions

- void lin_diag_service_callback (l_ifc_handle iii, l_u8 sid)

16.8.2 Macro Definition Documentation

16.8.2.1 #define DIAG_SERVICE_CALLBACK_HANDLER(*iii, sid*) lin_diag_service_callback((iii), (sid))

Definition at line 89 of file lin_commontl_api.h.

16.8.2.2   #define GENERAL_REJECT 0x10U

Error code raised when request for service not supported comes

Definition at line 74 of file lin_commontl_api.h.

16.8.2.3   #define LD_ANY_FUNCTION 0xFFFFU

Function

Definition at line 69 of file lin_commontl_api.h.

16.8.2.4   #define LD_ANY_MESSAGE 0xFFFFU

Message

Definition at line 70 of file lin_commontl_api.h.

16.8.2.5   #define LD_ANY_SUPPLIER 0x7FFFU

Supplier

Definition at line 68 of file lin_commontl_api.h.

16.8.2.6   #define LD_BROADCAST 0x7FU

Broadcast NAD

Definition at line 66 of file lin_commontl_api.h.

16.8.2.7   #define LD_DATA_ERROR 0x43U

Data error

Definition at line 53 of file lin_commontl_api.h.

16.8.2.8   #define LD_FUNCTIONAL_NAD 0x7EU

Functional NAD

Definition at line 67 of file lin_commontl_api.h.

16.8.2.9   #define LD_LENGTH_NOT_CORRECT 0x44U

Length not correct

Definition at line 54 of file lin_commontl_api.h.

16.8.2.10   #define LD_LENGTH_TOO_SHORT 0x34U

Length too short

Definition at line 51 of file lin_commontl_api.h.

16.8.2.11   #define LD_READ_OK 0x33U

Read OK

Definition at line 50 of file lin_commontl_api.h.

16.8.2.12   #define LD_SET_OK 0x45U

Set OK

Definition at line 55 of file lin_commontl_api.h.

16.8.2.13   #define LIN_PRODUCT_ID 0x00U

Node product identifier

Definition at line 62 of file lin_commontl_api.h.

16.8.2.14   #define LIN_SERIAL_NUMBER 0x01U

Serial number

Definition at line 63 of file lin_commontl_api.h.

16.8.2.15   #define NEGATIVE 0U

Negative response

Definition at line 79 of file lin_commontl_api.h.

16.8.2.16   #define POSITIVE 1U

Positive response

Definition at line 80 of file lin_commontl_api.h.

16.8.2.17   #define RECEIVING 1U

Receiving

Definition at line 83 of file lin_commontl_api.h.

16.8.2.18   #define RES_NEGATIVE 0x7FU

Negative response

Definition at line 73 of file lin_commontl_api.h.

16.8.2.19   #define RES_POSITIVE 0x40U

Positive response

Definition at line 59 of file lin_commontl_api.h.

16.8.2.20   #define SERVICE_NOT_SUPPORTED 0x11U

Error code in negative response for not supported service

Definition at line 75 of file lin_commontl_api.h.

16.8.2.21   #define SERVICE_TARGET_RESET 0xB5U

Target reset service

Definition at line 58 of file lin_commontl_api.h.

16.8.2.22   #define SUBFUNCTION_NOT_SUPPORTED 0x12U

Error code in negative response for not supported sub function

Definition at line 76 of file lin_commontl_api.h.

16.8.2.23   #define TRANSMITTING 0U

Transmitting

Definition at line 82 of file lin_commontl_api.h.

16.8.3   Function Documentation

16.8.3.1   void lin_diag_service_callback (l_ifc_handle *iii,* l_u8 *sid*)

Definition at line 1059 of file lin_diagnostic_service.c.

16.9 Comparator (CMP)

16.9.1 Detailed Description

Hardware background

The comparator (CMP) module is an analog comparator integrated in MCU.

Features of the CMP module include:

- 8 bit DAC with 2 voltage reference source

- 8 analog inputs from external pins

- Round robin check. In summary, this allow the CMP to operate independently in STOP and VLPS mode, whilst being triggered periodically to sample up to 8 inputs. Only if an input changes state is a full wakeup generated.

- Operational over the entire supply range

- Inputs may range from rail to rail

- Programmable hysteresis control

- Selectable interrupt on rising-edge, falling-edge, or both rising or falling edges of the comparator output

- Selectable inversion on comparator output

- Capability to produce a wide range of outputs such as: sampled, windowed, which is ideal for certain PWM zero-crossing-detection applications and digitally filtered

- A comparison event can be selected to trigger a DMA transfer

- The window and filter functions are not available in STOP modes.

How to use the CMP driver in your application

The user can configure the CMP in many ways: -CMP_DRV_Init - configures all CMP features -CMP_DRV_←
ConfigDAC - configures only DAC features -CMP_DRV_ConfigTriggerMode - configures only trigger mode features -CMP_DRV_ConfigComparator - configures only analog comparator features -CMP_DRV_ConfigMUX - configures only MUX features

Also the current configuration can be read using: -CMP_DRV_GetConfigAll - gets all CMP configuration -CM←
P_DRV_GetDACConfig - gets only DAC configuration -CMP_DRV_GetMUXConfig - gets only MUX configuration -CMP_DRV_GetInitTriggerMode - gets only trigger mode configuration -CMP_DRV_GetComparatorConfig - gets only analog comparator features

When the MCU exits from STOP mode CMP_DRV_GetInputFlags can be used to get the channel which triggered the wakeup. Please use this function only in this use case. CMP_DRV_ClearInputFlags will be used to clear this input change flags.

CMP_DRV_GetOutputFlags can be used to get output flag state and CMP_DRV_GetOutputFlags to clear them.

The main structure used to configure your application is **cmp_module_t**. This structure includes configuration structures for trigger mode, MUX, DAC and comparator: **cmp_comparator_t**, **cmp_anmux_t**, **cmp_dac_t** and **cmp_trigger_mode_t**

Example for S32K14x:

The next example will compare 2 external signals (CMP input 0 an CMP input 1). The output can be measured on port E, pin 4.

```
const cmp_module_t cmp_general_config =
 {
     {
         .dmaTriggerState       = false,
```

```
            .outputInterruptTrigger = CMP_NO_EVENT,
            .mode                   = CMP_CONTINUOUS,
            .filterSamplePeriod     = 0,
            .filterSampleCount      = 0,
            .powerMode              = CMP_LOW_SPEED,
            .inverterState          = CMP_NORMAL,
            .outputSelect           = CMP_COUT,
            .pinState               = CMP_AVAILABLE,
            .offsetLevel            = CMP_LEVEL_OFFSET_0,
            .hysteresisLevel        = CMP_LEVEL_HYS_0
        },

        {
            .positivePortMux        = CMP_MUX,
            .negativePortMux        = CMP_MUX,
            .positiveInputMux       = 0,
            .negativeInputMux       = 1
        },

        {
            .voltageReferenceSource = CMP_VIN1,
            .voltage                = 120,
            .state                  = false
        },

        {
            .roundRobinState            = false,
            .roundRobinInterruptState   = false,
            .fixedPort                  = CMP_PLUS_FIXED,
            .fixedChannel               = 0,
            .samples                    = 0,
            .initializationDelay        = 0,
            /* Channel 0 is enabled for round robin check */
            /* Channel 1 is enabled for round robin check */
            /* Channel 2 is enabled for round robin check */
            /* Channel 3 is enabled for round robin check */
            /* Channel 4 is enabled for round robin check */
            /* Channel 5 is enabled for round robin check */
            /* Channel 6 is enabled for round robin check */
            /* Channel 7 is enabled for round robin check */
            .roundRobinChannelsState    = 255,
            /* Initial comparison result for channel 0 is 1 */
            /* Initial comparison result for channel 1 is 1 */
            /* Initial comparison result for channel 2 is 1 */
            /* Initial comparison result for channel 3 is 1 */
            /* Initial comparison result for channel 4 is 1 */
            /* Initial comparison result for channel 5 is 1 */
            /* Initial comparison result for channel 6 is 1 */
            /* Initial comparison result for channel 7 is 1 */
            .programedState             = 255
        }
};

#define COMPARATOR_PORT         PORTA
#define COMPARATOR_INPUT1_PIN   0UL
#define COMPARATOR_INPUT2_PIN   1UL
#define COMPARATOR_OUTPUT       4UL
#define COMPARATOR_INSTANCE     0UL

int main(void)
{
    /* Initialize and configure clocks
     *  -   Setup system clocks
     *  -   Enable clock feed for Ports and Comparator
     *  -   See Clock Manager component for more info
     */
    CLOCK_SYS_Init(g_clockManConfigsArr, CLOCK_MANAGER_CONFIG_CNT,
                   g_clockManCallbacksArr, CLOCK_MANAGER_CALLBACK_CNT);
    CLOCK_SYS_UpdateConfiguration(0U,
     CLOCK_MANAGER_POLICY_AGREEMENT);

    /* Set pins used by CMP */
    /* The negative port is connected to PTA0 and positive port is connected to PTA1. The
     comparator output can be visualized on PTA4 */
    /* Initialize pins
     *  -   Setup input pins for Comparator
     *  -   Setup output pins for LEDs
     *  -   See PinSettings component for more info
     */
    PINS_DRV_Init(NUM_OF_CONFIGURED_PINS, g_pin_mux_InitConfigArr);

    /* Init CMP module */
    CMP_DRV_Init(COMPARATOR_INSTANCE, &cmp_general_config);
    for (;;)
        {}
    return(0);
}
```

Example for MPC574XG:

The next example will compare 2 external signals (CMP input 0 an CMP input 1). The output can be measured on port E, pin 4.

```c
const cmp_module_t cmp_general_config =
 {
    {
        .dmaTriggerState        = false,
        .outputInterruptTrigger = CMP_NO_EVENT,
        .mode                   = CMP_CONTINUOUS,
        .filterSamplePeriod     = 0,
        .filterSampleCount      = 0,
        .powerMode              = CMP_LOW_SPEED,
        .inverterState          = CMP_NORMAL,
        .outputSelect           = CMP_COUT,
        .pinState               = CMP_AVAILABLE,
        .hysteresisLevel        = CMP_LEVEL_HYS_0
    },

    {
        .positivePortMux        = CMP_MUX,
        .negativePortMux        = CMP_MUX,
        .positiveInputMux       = 0,
        .negativeInputMux       = 1
    },

    {
        .voltageReferenceSource = CMP_VIN1,
        .voltage                = 120,
        .state                  = false,
        .fixRefInputMux         = false
    },

    {
        .roundRobinState           = false,
        .roundRobinInterruptState  = false,
        .fixedPort                 = CMP_PLUS_FIXED,
        .fixedChannel              = 0,
        .samples                   = 0,
        /* Channel 0 is enabled for round robin check */
        /* Channel 1 is enabled for round robin check */
        /* Channel 2 is enabled for round robin check */
        /* Channel 3 is enabled for round robin check */
        /* Channel 4 is enabled for round robin check */
        /* Channel 5 is enabled for round robin check */
        /* Channel 6 is enabled for round robin check */
        /* Channel 7 is enabled for round robin check */
        .roundRobinChannelsState   = 255,
        /* Initial comparison result for channel 0 is 1 */
        /* Initial comparison result for channel 1 is 1 */
        /* Initial comparison result for channel 2 is 1 */
        /* Initial comparison result for channel 3 is 1 */
        /* Initial comparison result for channel 4 is 1 */
        /* Initial comparison result for channel 5 is 1 */
        /* Initial comparison result for channel 6 is 1 */
        /* Initial comparison result for channel 7 is 1 */
        .programedState            = 255
    }
};

#define COMPARATOR_PORT         PORTA
#define COMPARATOR_INPUT1_PIN   0UL
#define COMPARATOR_INPUT2_PIN   1UL
#define COMPARATOR_OUTPUT       4UL
#define COMPARATOR_INSTANCE     0UL

int main(void)
{
    /* Write your local variable definition here */
    /* Initialize and configure clocks
     *  -   Setup system clocks
     *  -   Enable clock feed for Ports and Comparator
     *  -   See Clock Manager component for more info
     */
    CLOCK_SYS_Init(g_clockManConfigsArr, CLOCK_MANAGER_CONFIG_CNT,
                    g_clockManCallbacksArr, CLOCK_MANAGER_CALLBACK_CNT);
    CLOCK_SYS_UpdateConfiguration(0U,
     CLOCK_MANAGER_POLICY_AGREEMENT);

    /* Set pins used by CMP */
    /* The negative port is connected to PTA0 and positive port is connected to PTA1. The
     comparator output can be visualized on PTA4 */
    /* Initialize pins
     *  -   Setup input pins for Comparator
```

```
 *  -   Setup output pins for LEDs
 *  -   See PinSettings component for more info
 */
PINS_DRV_Init(NUM_OF_CONFIGURED_PINS, g_pin_mux_InitConfigArr);

/* Init CMP module */
CMP_DRV_Init(COMPARATOR_INSTANCE, &cmp_general_config);
for (;;)
    {}
return(0);
}
```

Modules

- Comparator Driver

    Comparator Peripheral Driver.

16.10 Comparator Driver

16.10.1 Detailed Description

Comparator Peripheral Driver.

Definitions

Data Structures

- struct cmp_comparator_t

  Defines the block configuration. More...

- struct cmp_anmux_t

  Defines the analog mux. More...

- struct cmp_dac_t

  Defines the DAC block. More...

- struct cmp_trigger_mode_t

  Defines the trigger mode. More...

- struct cmp_module_t

  Defines the comparator module configuration. More...

Macros

- #define CMP_INPUT_FLAGS_MASK 0xFF0000
- #define CMP_INPUT_FLAGS_SHIFT 16U
- #define CMP_ROUND_ROBIN_CHANNELS_MASK 0xFF0000
- #define CMP_ROUND_ROBIN_CHANNELS_SHIFT 16U

Typedefs

- typedef uint8_t cmp_ch_list_t

  Comparator channels list (1bit/channel) |--------|--------|--—|--------|--------| |CH7_state|CH6_state|.....|CH1_↩ state|CH0_state| |--------|--------|--—|--------|--------| Implements : cmp_ch_list_t_Class.

- typedef uint8_t cmp_ch_number_t

  Number of channel Implements : cmp_ch_number_t_Class.

Enumerations

- enum cmp_power_mode_t { CMP_LOW_SPEED = 0U, CMP_HIGH_SPEED = 1U }

  Power Modes selection Implements : cmp_power_mode_t_Class.

- enum cmp_voltage_reference_t { CMP_VIN1 = 0U, CMP_VIN2 = 1U }

  Voltage Reference selection Implements : cmp_voltage_reference_t_Class.

- enum cmp_port_mux_t { CMP_DAC = CMP_DAC_SOURCE, CMP_MUX = CMP_MUX_SOURCE }

  Port Mux Source selection Implements : cmp_port_mux_t_Class.

- enum cmp_inverter_t { CMP_NORMAL = 0U, CMP_INVERT = 1U }

  Comparator output invert selection Implements : cmp_inverter_t_Class.

- enum cmp_output_select_t { CMP_COUT = 0U, CMP_COUTA = 1U }

  Comparator output select selection Implements : cmp_output_select_t_Class.

- enum cmp_output_enable_t { CMP_UNAVAILABLE = 0U, CMP_AVAILABLE = 1U }

  Comparator output pin enable selection Implements : cmp_output_enable_t_Class.

- enum cmp_hysteresis_t { CMP_LEVEL_HYS_0 = 0U, CMP_LEVEL_HYS_1 = 1U, CMP_LEVEL_HYS_2 = 2U, CMP_LEVEL_HYS_3 = 3U }

Comparator hysteresis control Implements : cmp_hysteresis_t_Class.

- enum cmp_fixed_port_t { CMP_PLUS_FIXED = 0U, CMP_MINUS_FIXED = 1U }

    Comparator Round-Robin fixed port Implements : cmp_fixed_port_t_Class.

- enum cmp_output_trigger_t { CMP_NO_EVENT = 0U, CMP_FALLING_EDGE = 1U, CMP_RISING_EDGE = 2U, CMP_BOTH_EDGES = 3U }

    Comparator output interrupt configuration Implements : cmp_output_trigger_t_Class.

- enum cmp_mode_t {
  CMP_DISABLED = 0U, CMP_CONTINUOUS = 1U, CMP_SAMPLED_NONFILTRED_INT_CLK = 2U, CM↩
  P_SAMPLED_NONFILTRED_EXT_CLK = 3U,
  CMP_SAMPLED_FILTRED_INT_CLK = 4U, CMP_SAMPLED_FILTRED_EXT_CLK = 5U, CMP_WINDO↩
  WED = 6U, CMP_WINDOWED_RESAMPLED = 7U,
  CMP_WINDOWED_FILTRED = 8U }

    Comparator functional modes Implements : cmp_mode_t_Class.

cMP DRV.

- status_t CMP_DRV_Reset (const uint32_t instance)

    Reset all registers.

- status_t CMP_DRV_GetInitConfigAll (cmp_module_t ∗config)

    Get reset configuration for all registers.

- status_t CMP_DRV_Init (const uint32_t instance, const cmp_module_t ∗const config)

    Configure all comparator features with the given configuration structure.

- status_t CMP_DRV_GetConfigAll (const uint32_t instance, cmp_module_t ∗const config)

    Gets the current comparator configuration.

- status_t CMP_DRV_GetInitConfigDAC (cmp_dac_t ∗config)

    Get reset configuration for registers related with DAC.

- status_t CMP_DRV_ConfigDAC (const uint32_t instance, const cmp_dac_t ∗config)

    Configure only the DAC component.

- status_t CMP_DRV_GetDACConfig (const uint32_t instance, cmp_dac_t ∗const config)

    Return current configuration for DAC.

- status_t CMP_DRV_GetInitConfigMUX (cmp_anmux_t ∗config)

    Get reset configuration for registers related with MUX.

- status_t CMP_DRV_ConfigMUX (const uint32_t instance, const cmp_anmux_t ∗config)

    Configure only the MUX component.

- status_t CMP_DRV_GetMUXConfig (const uint32_t instance, cmp_anmux_t ∗const config)

    Return configuration only for the MUX component.

- status_t CMP_DRV_GetInitTriggerMode (cmp_trigger_mode_t ∗config)

    Get reset configuration for registers related with Trigger Mode.

- status_t CMP_DRV_ConfigTriggerMode (const uint32_t instance, const cmp_trigger_mode_t ∗config)

    Configure trigger mode.

- status_t CMP_DRV_GetTriggerModeConfig (const uint32_t instance, cmp_trigger_mode_t ∗const config)

    Get current trigger mode configuration.

- status_t CMP_DRV_GetOutputFlags (const uint32_t instance, cmp_output_trigger_t ∗flags)

    Get comparator output flags.

- status_t CMP_DRV_ClearOutputFlags (const uint32_t instance)

    Clear comparator output flags.

- status_t CMP_DRV_GetInputFlags (const uint32_t instance, cmp_ch_list_t ∗flags)

    Gets input channels change flags.

- status_t CMP_DRV_ClearInputFlags (const uint32_t instance)

    Clear comparator input channels flags.

- status_t CMP_DRV_GetInitConfigComparator (cmp_comparator_t ∗config)

Get reset configuration for registers related with comparator features.

- status_t CMP_DRV_ConfigComparator (const uint32_t instance, const cmp_comparator_t ∗config)

    Configure only comparator features.

- status_t CMP_DRV_GetComparatorConfig (const uint32_t instance, cmp_comparator_t ∗config)

    Return configuration for comparator from CMP module.

16.10.2 Data Structure Documentation

16.10.2.1 struct cmp_comparator_t

Defines the block configuration.

This structure is used to configure only comparator block module(filtering, sampling, power_mode etc.) Implements : cmp_comparator_t_Class

Definition at line 172 of file cmp_driver.h.

Data Fields

- bool dmaTriggerState
- cmp_output_trigger_t outputInterruptTrigger
- cmp_mode_t mode
- uint8_t filterSamplePeriod
- uint8_t filterSampleCount
- cmp_power_mode_t powerMode
- cmp_inverter_t inverterState
- cmp_output_enable_t pinState
- cmp_output_select_t outputSelect
- cmp_hysteresis_t hysteresisLevel

Field Documentation

16.10.2.1.1 bool dmaTriggerState

True if DMA transfer trigger from comparator is enable.

Definition at line 174 of file cmp_driver.h.

16.10.2.1.2 uint8_t filterSampleCount

Number of sample count for filtering.

Definition at line 181 of file cmp_driver.h.

16.10.2.1.3 uint8_t filterSamplePeriod

Filter sample period.

Definition at line 180 of file cmp_driver.h.

16.10.2.1.4 cmp_hysteresis_t hysteresisLevel

CMP_LEVEL_HYS_0 if hard block output has level 0 hysteresis. CMP_LEVEL_HYS_1 if hard block output has level 1 hysteresis. CMP_LEVEL_HYS_2 if hard block output has level 2 hysteresis. CMP_LEVEL_HYS_3 if hard block output has level 3 hysteresis.

Definition at line 194 of file cmp_driver.h.

16.10.2.1.5 cmp_inverter_t inverterState

CMP_NORMAL if does not invert the comparator output. CMP_INVERT if inverts the comparator output.

Definition at line 184 of file cmp_driver.h.

16.10.2.1.6 cmp_mode_t mode

Configuration structure which define: the comparator functional mode, sample period and sample count.

Definition at line 179 of file cmp_driver.h.

16.10.2.1.7 cmp_output_trigger_t outputInterruptTrigger

CMP_NO_INTERRUPT comparator output would not trigger any interrupt. CMP_FALLING_EDGE comparator output would trigger an interrupt on falling edge. CMP_RISING_EDGE comparator output would trigger an interrupt on rising edge. CMP_BOTH_EDGES comparator output would trigger an interrupt on rising and falling edges.

Definition at line 175 of file cmp_driver.h.

16.10.2.1.8 cmp_output_select_t outputSelect

CMP_COUT if output signal is equal to COUT(filtered). CMP_COUTA if output signal is equal to COUTA(unfiltered).

Definition at line 188 of file cmp_driver.h.

16.10.2.1.9 cmp_output_enable_t pinState

CMP_UNAVAILABLE if comparator output is not available to package pin. CMP_AVAILABLE if comparator output is available to package pin.

Definition at line 186 of file cmp_driver.h.

16.10.2.1.10 cmp_power_mode_t powerMode

CMP_LOW_SPEED if low speed mode is selected. CMP_HIGH_SPEED if high speed mode is selected

Definition at line 182 of file cmp_driver.h.

16.10.2.2 struct cmp_anmux_t

Defines the analog mux.

This structure is used to configure the analog multiplexor to select compared signals Implements : cmp_anmux_↩
t_Class

Definition at line 206 of file cmp_driver.h.

Data Fields

- cmp_port_mux_t positivePortMux
- cmp_port_mux_t negativePortMux
- cmp_ch_number_t positiveInputMux
- cmp_ch_number_t negativeInputMux

Field Documentation

16.10.2.2.1 cmp_ch_number_t negativeInputMux

Select which channel is selected for the minus mux.

Definition at line 216 of file cmp_driver.h.

16.10.2.2.2 cmp_port_mux_t negativePortMux

Select negative port signal. CMP_DAC if source is digital to analog converter. CMP_MUX if source is 8 ch MUX

Definition at line 211 of file cmp_driver.h.

16.10.2.2.3 cmp_ch_number_t positiveInputMux

Select which channel is selected for the plus mux.

Definition at line 215 of file cmp_driver.h.

16.10.2.2.4   cmp_port_mux_t positivePortMux

Select positive port signal. CMP_DAC if source is digital to analog converter. CMP_MUX if source is 8 ch MUX

Definition at line 208 of file cmp_driver.h.

16.10.2.3   struct cmp_dac_t

Defines the DAC block.

This structure is used to configure the DAC block integrated in comparator module Implements : cmp_dac_t_Class

Definition at line 225 of file cmp_driver.h.

Data Fields

- cmp_voltage_reference_t voltageReferenceSource
- uint8_t voltage
- bool state

Field Documentation

16.10.2.3.1   bool state

True if DAC is enabled.

Definition at line 230 of file cmp_driver.h.

16.10.2.3.2   uint8_t voltage

The digital value which is converted to analog signal.

Definition at line 229 of file cmp_driver.h.

16.10.2.3.3   cmp_voltage_reference_t voltageReferenceSource

CMP_VIN1 if selected voltage reference is VIN1. CMP_VIN2 if selected voltage reference is VIN2.

Definition at line 227 of file cmp_driver.h.

16.10.2.4   struct cmp_trigger_mode_t

Defines the trigger mode.

This structure is used to configure the trigger mode operation when MCU enters STOP modes Implements : cmp↩
_trigger_mode_t_Class

Definition at line 245 of file cmp_driver.h.

Data Fields

- bool roundRobinState
- bool roundRobinInterruptState
- cmp_fixed_port_t fixedPort
- cmp_ch_number_t fixedChannel
- uint8_t samples
- cmp_ch_list_t roundRobinChannelsState
- cmp_ch_list_t programedState

Field Documentation

16.10.2.4.1 cmp_ch_number_t fixedChannel

Select which channel would be assigned to the fixed port.

Definition at line 251 of file cmp_driver.h.

16.10.2.4.2 cmp_fixed_port_t fixedPort

CMP_PLUS_FIXED if plus port is fixed. CMP_MINUS_FIXED if minus port is fixed.

Definition at line 249 of file cmp_driver.h.

16.10.2.4.3 cmp_ch_list_t programedState

Pre-programmed state for comparison result.

Definition at line 260 of file cmp_driver.h.

16.10.2.4.4 cmp_ch_list_t roundRobinChannelsState

One bite for each channel state. |———|———|—|———|———| |CH7_state|CH6_state|.....|CH1_state|CH0↩
_state| |-------—|-------—|-—|-------—|-------—|

Definition at line 256 of file cmp_driver.h.

16.10.2.4.5 bool roundRobinInterruptState

True if Round-Robin interrupt is enabled.

Definition at line 248 of file cmp_driver.h.

16.10.2.4.6 bool roundRobinState

True if Round-Robin is enabled.

Definition at line 247 of file cmp_driver.h.

16.10.2.4.7 uint8_t samples

Select number of round-robin clock cycles for a given channel.

Definition at line 252 of file cmp_driver.h.

16.10.2.5 struct cmp_module_t

Defines the comparator module configuration.

This structure is used to configure all components of comparator module Implements : cmp_module_t_Class

Definition at line 269 of file cmp_driver.h.

Data Fields

- cmp_comparator_t comparator
- cmp_anmux_t mux
- cmp_dac_t dac
- cmp_trigger_mode_t triggerMode

Field Documentation

16.10.2.5.1 cmp_comparator_t comparator

Definition at line 271 of file cmp_driver.h.

16.10.2.5.2 cmp_dac_t dac

Definition at line 273 of file cmp_driver.h.

16.10.2.5.3 cmp_anmux_t mux

Definition at line 272 of file cmp_driver.h.

16.10.2.5.4 cmp_trigger_mode_t triggerMode

Definition at line 274 of file cmp_driver.h.

16.10.3 Macro Definition Documentation

16.10.3.1 #define CMP_INPUT_FLAGS_MASK 0xFF0000

Definition at line 33 of file cmp_driver.h.

16.10.3.2 #define CMP_INPUT_FLAGS_SHIFT 16U

Definition at line 34 of file cmp_driver.h.

16.10.3.3 #define CMP_ROUND_ROBIN_CHANNELS_MASK 0xFF0000

Definition at line 35 of file cmp_driver.h.

16.10.3.4 #define CMP_ROUND_ROBIN_CHANNELS_SHIFT 16U

Definition at line 36 of file cmp_driver.h.

16.10.4 Typedef Documentation

16.10.4.1 typedef uint8_t cmp_ch_list_t

Comparator channels list (1bit/channel) |--------|--------|--—|--------|--------| |CH7_state|CH6_state|.....|CH1_↩
state|CH0_state| |--------|--------|--—|--------|--------| Implements : cmp_ch_list_t_Class.

Definition at line 159 of file cmp_driver.h.

16.10.4.2 typedef uint8_t cmp_ch_number_t

Number of channel Implements : cmp_ch_number_t_Class.

Definition at line 164 of file cmp_driver.h.

16.10.5 Enumeration Type Documentation

16.10.5.1 enum cmp_fixed_port_t

Comparator Round-Robin fixed port Implements : cmp_fixed_port_t_Class.

Enumerator

> **CMP_PLUS_FIXED** The Plus port is fixed. Only the inputs to the Minus port are swept in each round.
>
> **CMP_MINUS_FIXED** The Minus port is fixed. Only the inputs to the Plus port are swept in each round.

Definition at line 120 of file cmp_driver.h.

16.10.5.2 enum cmp_hysteresis_t

Comparator hysteresis control Implements : cmp_hysteresis_t_Class.

Enumerator

> ***CMP_LEVEL_HYS_0***
>
> ***CMP_LEVEL_HYS_1***
>
> ***CMP_LEVEL_HYS_2***
>
> ***CMP_LEVEL_HYS_3***

Definition at line 109 of file cmp_driver.h.

16.10.5.3 enum cmp_inverter_t

Comparator output invert selection Implements : cmp_inverter_t_Class.

Enumerator

> ***CMP_NORMAL*** Output signal isn't inverted.
>
> ***CMP_INVERT*** Output signal is inverted.

Definition at line 71 of file cmp_driver.h.

16.10.5.4 enum cmp_mode_t

Comparator functional modes Implements : cmp_mode_t_Class.

Enumerator

> ***CMP_DISABLED***
>
> ***CMP_CONTINUOUS***
>
> ***CMP_SAMPLED_NONFILTRED_INT_CLK***
>
> ***CMP_SAMPLED_NONFILTRED_EXT_CLK***
>
> ***CMP_SAMPLED_FILTRED_INT_CLK***
>
> ***CMP_SAMPLED_FILTRED_EXT_CLK***
>
> ***CMP_WINDOWED***
>
> ***CMP_WINDOWED_RESAMPLED***
>
> ***CMP_WINDOWED_FILTRED***

Definition at line 140 of file cmp_driver.h.

16.10.5.5 enum cmp_output_enable_t

Comparator output pin enable selection Implements : cmp_output_enable_t_Class.

Enumerator

> ***CMP_UNAVAILABLE*** Comparator output isn't available to a specific pin
>
> ***CMP_AVAILABLE*** Comparator output is available to a specific pin

Definition at line 89 of file cmp_driver.h.

16.10.5.6 enum **cmp_output_select_t**

Comparator output select selection Implements : cmp_output_select_t_Class.

Enumerator

       CMP_COUT   Select COUT as comparator output signal.

       CMP_COUTA   Select COUTA as comparator output signal.

Definition at line 80 of file cmp_driver.h.

16.10.5.7 enum **cmp_output_trigger_t**

Comparator output interrupt configuration Implements : cmp_output_trigger_t_Class.

Enumerator

       CMP_NO_EVENT   Comparator output interrupts are disabled OR no event occurred.

       CMP_FALLING_EDGE   Comparator output interrupts will be generated only on falling edge OR only falling edge event occurred.

       CMP_RISING_EDGE   Comparator output interrupts will be generated only on rising edge OR only rising edge event occurred.

       CMP_BOTH_EDGES   Comparator output interrupts will be generated on both edges OR both edges event occurred.

Definition at line 129 of file cmp_driver.h.

16.10.5.8 enum **cmp_port_mux_t**

Port Mux Source selection Implements : cmp_port_mux_t_Class.

Enumerator

       CMP_DAC   Select DAC as source for the comparator port.

       CMP_MUX   Select MUX8 as source for the comparator port.

Definition at line 62 of file cmp_driver.h.

16.10.5.9 enum **cmp_power_mode_t**

Power Modes selection Implements : cmp_power_mode_t_Class.

Enumerator

       CMP_LOW_SPEED   Module in low speed mode.

       CMP_HIGH_SPEED   Module in high speed mode.

Definition at line 44 of file cmp_driver.h.

16.10.5.10 enum **cmp_voltage_reference_t**

Voltage Reference selection Implements : cmp_voltage_reference_t_Class.

Enumerator

       CMP_VIN1   Use Vin1 as supply reference source for DAC.

       CMP_VIN2   Use Vin2 as supply reference source for DAC.

Definition at line 53 of file cmp_driver.h.

16.10.6   Function Documentation

16.10.6.1   status_t CMP_DRV_ClearInputFlags (const uint32_t *instance*)

Clear comparator input channels flags.

This function clear comparator input channels flags.

Parameters

instance	- instance number

Returns

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 512 of file cmp_driver.c.

16.10.6.2   status_t CMP_DRV_ClearOutputFlags (const uint32_t *instance*)

Clear comparator output flags.

This function clear comparator output flags(rising and falling edge).

Parameters

instance	- instance number

Returns

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 467 of file cmp_driver.c.

16.10.6.3   status_t CMP_DRV_ConfigComparator (const uint32_t *instance,* const cmp_comparator_t ∗ *config*)

Configure only comparator features.

This function configure only features related with comparator: DMA request, power mode, output select, interrupts enable, invert, offset, hysteresis.

Parameters

instance	- instance number
config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 561 of file cmp_driver.c.

16.10.6.4   status_t CMP_DRV_ConfigDAC (const uint32_t *instance,* const cmp_dac_t ∗ *config*)

Configure only the DAC component.

This function configures the DAC with the options provided in the config structure.

Parameters

instance	- instance number
config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 266 of file cmp_driver.c.

16.10.6.5 status_t CMP_DRV_ConfigMUX (const uint32_t *instance,* const cmp_anmux_t ∗ *config*)

Configure only the MUX component.

This function configures the MUX with the options provided in the config structure.

Parameters

instance	- instance number
config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 331 of file cmp_driver.c.

16.10.6.6 status_t CMP_DRV_ConfigTriggerMode (const uint32_t *instance,* const cmp_trigger_mode_t ∗ *config*)

Configure trigger mode.

This function configures the trigger mode with the options provided in the config structure.

Parameters

instance	- instance number
config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 395 of file cmp_driver.c.

16.10.6.7 status_t CMP_DRV_GetComparatorConfig (const uint32_t *instance,* cmp_comparator_t ∗ *config*)

Return configuration for comparator from CMP module.

This function return configuration for features related with comparator: DMA request, power mode, output select, interrupts enable, invert, offset, hysteresis.

Parameters

instance	- instance number

config	- the configuration structure returned

Returns

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 591 of file cmp_driver.c.

16.10.6.8   status_t CMP_DRV_GetConfigAll (const uint32_t *instance,* cmp_module_t ∗const *config*)

Gets the current comparator configuration.

This function returns the current configuration for comparator as a configuration structure.

Parameters

instance	- instance number
config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 192 of file cmp_driver.c.

16.10.6.9   status_t CMP_DRV_GetDACConfig (const uint32_t *instance,* cmp_dac_t ∗const *config*)

Return current configuration for DAC.

This function returns current configuration only for DAC.

Parameters

instance	- instance number
config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 290 of file cmp_driver.c.

16.10.6.10   status_t CMP_DRV_GetInitConfigAll (cmp_module_t ∗ *config*)

Get reset configuration for all registers.

This function returns a configuration structure with reset values for all registers from comparator module.

Parameters

config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 88 of file cmp_driver.c.

16.10.6.11 status_t CMP_DRV_GetInitConfigComparator (cmp_comparator_t ∗ *config*)

Get reset configuration for registers related with comparator features.

This function return a configuration structure with reset values for features associated with comparator (DMA request, power mode, output select, interrupts enable, invert, offset, hysteresis).

Parameters

config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 534 of file cmp_driver.c.

16.10.6.12 status_t CMP_DRV_GetInitConfigDAC (cmp_dac_t ∗ *config*)

Get reset configuration for registers related with DAC.

This function returns a configuration structure with reset values for features associated with DAC.

Parameters

config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 245 of file cmp_driver.c.

16.10.6.13 status_t CMP_DRV_GetInitConfigMUX (cmp_anmux_t ∗ *config*)

Get reset configuration for registers related with MUX.

This function returns a configuration structure with reset values for features associated with MUX.

Parameters

config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 312 of file cmp_driver.c.

16.10.6.14 status_t CMP_DRV_GetInitTriggerMode (cmp_trigger_mode_t ∗ *config*)

Get reset configuration for registers related with Trigger Mode.

This function returns a configuration structure with reset values for features associated with Trigger Mode.

Parameters

config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 372 of file cmp_driver.c.

16.10.6.15   status_t CMP_DRV_GetInputFlags (const uint32_t *instance,* cmp_ch_list_t ∗ *flags*)

Gets input channels change flags.

This function return in <flags> all input channels flags as uint8_t(1 bite for each channel flag).

Parameters

instance	- instance number
flags	- pointer to input flags

Returns

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 494 of file cmp_driver.c.

16.10.6.16   status_t CMP_DRV_GetMUXConfig (const uint32_t *instance,* cmp_anmux_t ∗const *config*)

Return configuration only for the MUX component.

This function returns current configuration to determine which signals go to comparator ports.

Parameters

instance	- instance number
config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 352 of file cmp_driver.c.

16.10.6.17   status_t CMP_DRV_GetOutputFlags (const uint32_t *instance,* cmp_output_trigger_t ∗ *flags*)

Get comparator output flags.

This function returns in <flags> comparator output flags(rising and falling edge).

Parameters

instance	- instance number
-	flags - pointer to output flags NO_EVENT RISING_EDGE FALLING_EDGE BOTH_EDGE

Returns

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 449 of file cmp_driver.c.

16.10.6.18   status_t CMP_DRV_GetTriggerModeConfig (const uint32_t *instance,* **cmp_trigger_mode_t** ∗**const** *config* **)**

Get current trigger mode configuration.

This function returns the current trigger mode configuration for trigger mode.

Parameters

instance	- instance number
config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 422 of file cmp_driver.c.

16.10.6.19   status_t CMP_DRV_Init (const uint32_t *instance,* **const cmp_module_t** ∗**const** *config* **)**

Configure all comparator features with the given configuration structure.

This function configures the comparator module with the options provided in the config structure.

Parameters

instance	- instance number
config	- the configuration structure

Returns

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 138 of file cmp_driver.c.

16.10.6.20   status_t CMP_DRV_Reset (const uint32_t *instance* **)**

Reset all registers.

This function set all CMP registers to reset values.

Parameters

instance	- instance number

Returns

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 69 of file cmp_driver.c.

16.11 Controller Area Network with Flexible Data Rate (FlexCAN)

16.11.1 Detailed Description

The S32 SDK provides a Peripheral Driver for the FlexCAN module of S32 SDK devices.

Hardware background

The FlexCAN module is a communication controller implementing the CAN protocol according to the ISO 11898-1 standard and CAN 2.0 B protocol specifications. The FlexCAN module is a full implementation of the CAN protocol specification, the CAN with Flexible Data rate (CAN FD) protocol and the CAN 2.0 version B protocol, which supports both standard and extended message frames and long payloads up to 64 bytes transfered at faster rates up to 8 Mbps. The message buffers are stored in an embedded RAM dedicated to the FlexCAN module.

The FlexCAN module includes these distinctive features:

- Full implementation of the CAN with Flexible Data Rate (CAN FD) protocol specification and CAN protocol specification, Version 2.0 B (see the FEATURE_CAN_HAS_FD define for the availability of this feature on each platform)

    - Standard data frames
    - Extended data frames
    - Zero to sixty four bytes data length
    - Programmable bit rate (see the chip-specific FlexCAN information for the specific maximum bit rate configuration)
    - Content-related addressing

- Compliant with the ISO 11898-1 standard

- Flexible mailboxes configurable to store 0 to 8, 16, 32 or 64 bytes data length (payloads longer than 8 bytes are available only for some platforms, see the FEATURE_CAN_HAS_FD define)

- Each mailbox configurable as receive or transmit, all supporting standard and extended messages

- Individual Rx Mask registers per mailbox

- Full-featured Rx FIFO with storage capacity for up to six frames and automatic internal pointer handling with DMA support (DMA support is available only for some platforms, see the FEATURE_CAN_HAS_DMA_EN↩ABLE define)

- Transmission abort capability

- Flexible message buffers (MBs) configurable as Rx or Tx (see the FEATURE_CAN_MAX_MB_NUM define for the specific maximum number of message buffers configurable on each platform)

- Programmable clock source to the CAN Protocol Interface, either peripheral clock or oscillator clock (this feature might differ depending on the platform, see FEATURE_CAN_HAS_PE_CLKSRC_SELECT define for the availability of this feature on each platform)

- RAM not used by reception or transmission structures can be used as general purpose RAM space

- Listen-Only mode capability

- Programmable Loop-Back mode supporting self-test operation

- Maskable interrupts

- Short latency time due to an arbitration scheme for high-priority messages

- Low power modes or matching with received frames - Pretended Networking (see FEATURE_CAN_HAS_↩PRETENDED_NETWORKING define for the availability of this feature on each platform)

- Transceiver Delay Compensation feature when transmitting CAN FD messages at faster data rates (see the FEATURE_CAN_HAS_FD define for the availability of this feature on each platform)

- Remote request frames may be handled automatically or by software

- CAN bit time settings and configuration bits can only be written in Freeze mode

- SYNCH bit available in Error in Status 1 register to inform that the module is synchronous with CAN bus

- CRC status for transmitted message

- Rx FIFO Global Mask register

- Selectable priority between mailboxes and Rx FIFO during matching process

- Powerful Rx FIFO ID filtering, capable of matching incoming IDs against either 128 extended, 256 standard, or 512 partial (8 bit) IDs, with up to 32 individual masking capability

- 100% backward compatibility with previous FlexCAN version

- Supports Pretended Networking functionality in low power: Stop mode (see FEATURE_CAN_HAS_PRET↩ ENDED_NETWORKING define for the availability of this feature on each platform)

- Supports detection and correction of errors in memory read accesses. Errors in one bit can be corrected and errors in 2 bits can be detected but not corrected (this feature might not be available on some platforms, see chip-specific FlexCAN information for details)

- Supports Self Wake Up feature when FlexCAN is in a low power mode: Stop mode (see FEATURE_CAN_↩ HAS_SELF_WAKE_UP define for the availability of this feature on each platform)

Modules

- FlexCAN Driver

16.12   Cooked API

16.12.1   Detailed Description

Cooked processing of diagnostic messages manages one complete message at a time.

Functions

- void ld_send_message (l_ifc_handle iii, l_u16 length, l_u8 NAD, const l_u8 ∗const data)

  Pack the information specified by data and length into one or multiple diagnostic frames.
- void ld_receive_message (l_ifc_handle iii, l_u16 ∗const length, l_u8 ∗const NAD, l_u8 ∗const data)

  Prepare the LIN diagnostic module to receive one message and store it in the buffer pointed to by data.
- l_u8 ld_tx_status (l_ifc_handle iii)

  Get the status of the last made call to ld_send_message.
- l_u8 ld_rx_status (l_ifc_handle iii)

  Get the status of the last made call to ld_send_message.

16.12.2   Function Documentation

16.12.2.1   void ld_receive_message (l_ifc_handle *iii,* l_u16 ∗const *length,* l_u8 ∗const *NAD,* l_u8 ∗const *data*)

Prepare the LIN diagnostic module to receive one message and store it in the buffer pointed to by data.

Parameters

in	*iii*	Lin interface handle
in	*length*	Length of data to receive
in	*NAD*	Node address of slave node
in	*data*	Data to be sent

Returns

> void

Prepare the LIN diagnostic module to receive one message and store it in the buffer pointed to by data. At the call, length shall specify the maximum length allowed. When the reception has completed, length is changed to the actual length and NAD to the NAD in the message.

Definition at line 385 of file lin_commontl_api.c.

16.12.2.2   l_u8 ld_rx_status (l_ifc_handle *iii*)

Get the status of the last made call to ld_send_message.

Parameters

in	*iii*	Lin interface handle

Returns

> l_u8

The call returns the status of the last made call to ld_receive_message. < br / > The following values can be returned: < br / > LD_IN_PROGRESS: The reception is not yet completed. < br / > LD_COMPLETED: The reception has completed successfully and all < br / > information (length, NAD, data) is available. (You can < br / > also issue a new ld_receive_message call). This < br / > value is also returned after initialization of the < br / > transport layer. < br / > LD_FAILED: The reception ended in an error. The data was only < br / > partially received and should not be trusted. Initialize < br / > before processing further transport layer messages. < br /

$>$ For LIN2.0 and J2602 Users can make a new call to ld_receive_message. For LIN2.1 and above, the transport layer shall be reinitialized before processing further messages. To find out why a transmission has failed, check the status management function l_ifc_read_status. $*$ LD_N_CR_TIMEOUT The reception failed because of a N_Cr timeout (For LIN2.1 and above only)$<$ br $/>$ LD_WRONG_SN The reception failed because of an unexpected sequence number. (For LIN2.1 and above only)

Definition at line 431 of file lin_commontl_api.c.

16.12.2.3   void ld_send_message (l_ifc_handle *iii,* l_u16 *length,* l_u8 *NAD,* const l_u8 $*$const *data*)

Pack the information specified by data and length into one or multiple diagnostic frames.

Parameters

in	*iii*	Lin interface handle
in	*length*	Length of data to send
in	*NAD*	Node address of slave node
in	*data*	Data to be sent

Returns

> void

Pack the information specified by data and length into one or multiple diagnostic frames. If the call is made in a master node application the frames are transmitted to the slave node with the address NAD. If the call is made in a slave node application the frames are transmitted to the master node with the address NAD. The parameter NAD is not used in slave nodes.

Definition at line 214 of file lin_commontl_api.c.

16.12.2.4   l_u8 ld_tx_status (l_ifc_handle *iii*)

Get the status of the last made call to ld_send_message.

Parameters

in	*iii*	Lin interface handle

Returns

> l_u8

Get the status of the last made call to ld_send_message. The following values can be returned: LD_IN_PRO$\leftarrow$ GRESS: The transmission is not yet completed. LD_COMPLETED: The transmission has completed successfully (and you can issue a new ld_send_message call). This value is also returned after initialization of the transport layer. LD_FAILED: The transmission ended in an error. The data was only partially sent. The transport layer shall be reinitialized before processing further messages. To find out why a transmission has failed, check the status management function l_read_status. For LIN2.0 and J2602 Users can make a new call to ld_send_message. For LIN2.1 and above, the transport layer shall be reinitialized before processing further messages. LD_N_AS_TIME$\leftarrow$ OUT: The transmission failed because of a N_As timeout. This applies for LIN2.1 and above only.

Definition at line 415 of file lin_commontl_api.c.

16.13 Cryptographic Services Engine (CSEc)

16.13.1 Detailed Description

The S32 SDK provides a Peripheral Driver for the Cryptographic Services Engine (CSEc) module of S32 SDK devices.

The FTFC module has added features to comply with the SHE specification. By using an embedded processor, firmware and hardware assisted AES-128 sub-block, the FTFC macro enables encryption, decryption and message generation and authentication algorithms for secure messaging applications. Additionally a TRNG and Miyaguchi-Prenell compression sub-blocks enables true random number generation (entropy generator for PRNG in AES sub-block).

Hardware background

Features of the CSEc module include:

- Secure cryptographic key storage (ranging from 3 to 21 user keys)

- AES-128 encryption and decryption

- AES-128 CMAC (Cipher-based Message Authentication Code) calculation and authentication

- ECB (Electronic Cypher Book) Mode - encryption and decryption

- CBC (Cipher Block Chaining) Mode - encryption and decryption

- True and Pseudo random number generation

- Miyaguchi-Prenell compression function

- Secure Boot Mode (user configurable)

    – Sequential Boot Mode

    – Parallel Boot Mode

    – Strict Sequential Boot Mode (unchangeable once set)

Modules

- CSEc Driver

    Cryptographic Services Engine Peripheral Driver.

16.14 Cyclic Redundancy Check (CRC)

16.14.1 Detailed Description

The S32 SDK provides Peripheral Drivers for the Cyclic Redundancy Check (CRC) module.
.

The cyclic redundancy check (CRC) module generates 16/32-bit CRC code with S32K1xx and S32M or 8/16/32-bit CRC code with MPC574x for error detection.
The CRC module provides a programmable polynomial, seed, and other parameters required to implement 16/32-bit CRC standard with S32K1xx and S32M or 8/16/32-bit CRC standard with MPC574x .
The 16/32-bit code with S32K1xx and S32M or 8/16/32-bit code with MPC574x is calculated for 32 bits of data at a time.

Important note when use CRC module with MPC574x devices:

1. When generating CRC-32 for the ITU-T V.42 standard the user needs to set SWAP_BYTEWISE together with INV and SWAP.

2. When generating CRC-16 the user needs to set SWAP_BITWISE bit.

Modules

- CRC Driver

16.15 Diagnostic services

16.15.1 Detailed Description

Diagnostic services defines methods to implement diagnostic data transfer between a master node connected with a diagnostic tester and the slave nodes.

Three different classes of diagnostic nodes are supported.

The master node and the diagnostic tester are connected via a back-bone bus (e.g. CAN). The master node shall receive all diagnostic requests addressed to the slave nodes from the back-bone bus, and gateway them to the correct LIN cluster(s). Responses from the slave nodes shall be gatewayed back to the back-bone bus through the master node.

All diagnostic requests and responses (services) addressed to the slave nodes can be routed in the network layer (i.e. no application layer routing). In this case, the master node must implement the LIN transport protocol, see Transport Layer Specification, as well as the transport protocols used on the back- bone busses (e.g. IS↩O15765-2 on CAN).

Currently, LinStack support some service. With other service which LinStack doesn't support or user want to add action when any service is received, user can choose or create service in supported services of PEX GUI and use API of transport layer to implement it. in application.

Example in slave node:

```
for(;;)
{
    /* length shall specify the maximum length allowed */
    length = 106;
    ld_receive_message(LI0,&length, &nad, req_data);
    /* if receive READ_DATA_BY_IDENTIFIER master request successfully */
    if(diag_get_flag(LI0, LI0_DIAGSRV_READ_DATA_BY_IDENTIFIER_ORDER))
    {
        diag_clear_flag(LI0, LI0_DIAGSRV_READ_DATA_BY_IDENTIFIER_ORDER);
        /* implement what you want to do when receive this message
           length will return real length of this message
           req_data will contain SID and data of this message */
        /* send back response data */
        ld_send_message(LI0,17,nad, res_data);
    }
}
```

Modules

- Node configuration

    This group contains APIs that used for node configuration purpose.

- Node identification

    This group contains API that used for node identification purpose.

Functions

- void diag_read_data_by_identifier (l_ifc_handle iii, const l_u8 NAD, const l_u8 ∗const data)

    This function reads data by identifier, Diagnostic Class II service (0x22).

- void diag_write_data_by_identifier (l_ifc_handle iii, const l_u8 NAD, l_u16 data_length, const l_u8 ∗const data)

    Write Data by Identifier for a specified node - Diagnostic Class II service (0x2E)

- void diag_session_control (l_ifc_handle iii, const l_u8 NAD, l_u16 data_length, const l_u8 ∗const data)

    This function is used for master node only. It will pack data and send request to slave node with service ID = 0x10: Session control.

- void diag_fault_memory_read (l_ifc_handle iii, const l_u8 NAD, l_u16 data_length, const l_u8 ∗const data)

This function is used for master node only. It will pack data and send request to slave node with service ID = 0x19: Fault memory read.

- void [diag_fault_memory_clear](l_ifc_handle iii, const l_u8 NAD, l_u16 data_length, const l_u8 ∗const data)

  This function is used for master node only. It will pack data and send request to slave node with service ID = 0x14: Fault memory clear.

- void [diag_IO_control](l_ifc_handle iii, const l_u8 NAD, l_u16 data_length, const l_u8 ∗const data)

  This function is used for master node only. It will pack data and send request to slave node with service ID = 0x2F: Input/Output control service.

- l_u8 [diag_get_flag](l_ifc_handle iii, l_u8 flag_order)

  This function will return flag of diagnostic service, if LIN slave node receive master request of the diagnostic service.

- void [diag_clear_flag](l_ifc_handle iii, l_u8 flag_order)

  This function will clear flag of diagnostic service,.

16.15.2 Function Documentation

16.15.2.1 void diag_clear_flag (l_ifc_handle *iii,* l_u8 *flag_order*)

This function will clear flag of diagnostic service,.

Parameters

in	*iii*	LIN interface handle
in	*flag_order*	Order of service flag

Returns

> void

Definition at line 1031 of file lin_diagnostic_service.c.

16.15.2.2 void diag_fault_memory_clear (l_ifc_handle *iii,* const l_u8 *NAD,* l_u16 *data_length,* const l_u8 ∗const *data*)

This function is used for master node only. It will pack data and send request to slave node with service ID = 0x14: Fault memory clear.

Parameters

in	*iii*	LIN interface handle
in	*NAD*	Node address value of the destination node for the transmission
in	*data_length*	Data length of frame
in	*data*	Buffer for the data to be transmitted

Returns

> void

Definition at line 765 of file lin_diagnostic_service.c.

16.15.2.3 void diag_fault_memory_read (l_ifc_handle *iii,* const l_u8 *NAD,* l_u16 *data_length,* const l_u8 ∗const *data*)

This function is used for master node only. It will pack data and send request to slave node with service ID = 0x19: Fault memory read.

Parameters

in	*iii*	LIN interface handle
in	*NAD*	Node address value of the destination node for the transmission
in	*data_length*	Data length of frame
in	*data*	Buffer for the data to be transmitted

Returns

> void

Definition at line 718 of file lin_diagnostic_service.c.

16.15.2.4   l_u8 diag_get_flag (l_ifc_handle *iii,* l_u8 *flag_order*)

This function will return flag of diagnostic service, if LIN slave node receive master request of the diagnostic service.

Parameters

in	*iii*	LIN interface handle
in	*flag_order*	Order of service flag

Returns

> 1 if LIN Slave node receives master request of the diagnostic service, and the flag has not been cleared by
> diag_clear_flag
> 0 default value
> 0xFF if service is not supported

Definition at line 1000 of file lin_diagnostic_service.c.

16.15.2.5   void diag_IO_control (l_ifc_handle *iii,* const l_u8 *NAD,* l_u16 *data_length,* const l_u8 ∗const *data*)

This function is used for master node only. It will pack data and send request to slave node with service ID = 0x2F:
Input/Output control service.

Parameters

in	*iii*	LIN interface handle
in	*NAD*	Node address value of the destination node for the transmission
in	*data_length*	Data length of frame
in	*data*	Buffer for the data to be transmitted

Returns

> void

Definition at line 811 of file lin_diagnostic_service.c.

16.15.2.6   void diag_read_data_by_identifier (l_ifc_handle *iii,* const l_u8 *NAD,* const l_u8 ∗const *data*)

This function reads data by identifier, Diagnostic Class II service (0x22).

Parameters

in	*iii*	LIN interface handle
in	*NAD*	Node address value of the destination node for the transmission
in	*data*	Buffer for the data to be transmitted

Returns

> void

This function is for Master node only.

Definition at line 571 of file lin_diagnostic_service.c.

16.15.2.7   void diag_session_control (l_ifc_handle *iii,* const l_u8 *NAD,* l_u16 *data_length,* const l_u8 ∗const *data*)

This function is used for master node only. It will pack data and send request to slave node with service ID = 0x10: Session control.

Parameters

in	*iii*	LIN interface handle
in	*NAD*	Node address value of the destination node for the transmission
in	*data_length*	Data length of frame
in	*data*	Buffer for the data to be transmitted

Returns

> void

Definition at line 671 of file lin_diagnostic_service.c.

16.15.2.8   void diag_write_data_by_identifier (l_ifc_handle *iii,* const l_u8 *NAD,* l_u16 *data_length,* const l_u8 ∗const *data*)

Write Data by Identifier for a specified node - Diagnostic Class II service (0x2E)

Parameters

in	*iii*	Lin interface handle
in	*NAD*	Node address value of the destination node for the transmission
in	*data_length*	Data length of frame
in	*data*	Buffer for the data to be transmitted

Returns

> void

This function is for Master node only.

Definition at line 611 of file lin_diagnostic_service.c.

16.16   Driver and cluster management

16.16.1   Detailed Description

API perform the initialization of the LIN core.

Functions

- l_bool l_sys_init (void)

  This function performs the initialization of the LIN core; is the first call a user must use in the LIN core before using any other API functions. The implementation of this function can be replaced by user if needed.

16.16.2   Function Documentation

16.16.2.1   l_bool l_sys_init (void)

This function performs the initialization of the LIN core; is the first call a user must use in the LIN core before using any other API functions. The implementation of this function can be replaced by user if needed.

Returns

Operation status = Zero, which is equivalent to 'Initialization was successful'.

Definition at line 60 of file lin_common_api.c.

16.17 EDMA Driver

16.17.1 Detailed Description

This module covers the functionality of the Enhanced Direct Memory Access (eDMA) peripheral driver.

The eDMA driver implements direct memory access functionality with multiple features: (single block/multi block/loop/scatter-gather transfers); the main usage of this module is to offload the bus read/write accesses from the core to the eDMA engine.

Features

- Memory-to-memory, peripheral-to-memory, memory-to-peripheral transfers

- Simple single-block transfers with minimum configuration

- Multi-block transfers with minimum configuration (based on subsequent requests)

- Loop transfers for complex use-cases (e.g. double buffering)

- Scatter/gather

- Dynamic channel allocation

Functionality

Initialization

In order to use the eDMA driver, the module must be first initialized, using EDMA_DRV_Init() function. Once initialized, it cannot be initialized again until it is de-initialized, using EDMA_DRV_Deinit(). The initialization function does the following operations:

- resets eDMA and DMAMUX modules

- clears the eDMA driver state structure

- sets the arbitration mode and halt settings

- enables error and channel interrupts

Upon module initialization, the application must initialize the channel(s) to be used, using EDMA_DRV_Channel↩ Init() function. This operation means enabling an eDMA channel number (or dynamically allocating one), selecting a source trigger (eDMA request multiplexed via DMAMUX) and setting the channel priority. Additionally, a user callback can be installed for each channel, which will be called when the corresponding interrupt is triggered.

Transfer Configuration

After initialization, the transfer control descriptor for the selected channel must be configured before use. Depending on the application use-case, on of the three transfer configuration methods should be called.

Single-block transfer

For the simplest use-case where a contiguous chunk of data must be transferred, the most suitable function is EDMA_DRV_ConfigSingleBlockTransfer(). This takes the source/destination addresses as parameters, as well as transfer type/size and data buffer size, and configures the channel TCD to read/write the data in a single request. The looping and scatter/gather features are not used in this scenario. The driver computes the appropriate offsets for source/destination addresses and set the other TCD fields.

Multi-block transfer

This type of transfer can be seen as a sequence of single-block transfers, as described above, which are triggered by subsequent requests. This configuration is suitable for contiguous chunks of data which need to be transferred in multiple steps (e.g. writing one/several bytes from a memory buffer to a peripheral data register each time the

module is free - eDMA-based communication). In order to configure this kind of transfer, EDMA_DRV_Config↩
MultiBlockTransfer function should be used; aside from the EDMA_DRV_ConfigSingleBlockTransfer parameters, this function also takes two additional parameters: the number of transfer loops (expected number of requests to finish the data) and a boolean variable configuring whether requests should be disabled for the current channel upon transfer completion.

Loop transfer

The eDMA IP supports complex addressing modes. One of the methods to configure complex transfers in multiple requests is using the minor/major loop support. The EDMA_DRV_ConfigLoopTransfer() function sets up the transfer control descriptor for subsequent requests to trigger multiple transfers. The addresses are adjusted after each minor/major loop, according to user setup. This method takes a transfer configuration structure as parameter, with settings for all the fields that control addressing mode (source/destination offsets, minor loop offset, channel linking, minor/major loop count, address last adjustments). It is the responsibility of the application to correctly initialize the configuration structure passed to this function, according to the addressed use-case.

Scatter/gather

The eDMA driver also supports scatter/gather feature, which allows various transfer scenarios. When scatter/gather is enabled, a new TCD structure is automatically loaded in the current channel's TCD registers when a transfer is complete, allowing the application to define multiple different subsequent transfers. The EDMA_DRV_Config↩
ScatterGatherTransfer() function sets up a list of TCD structures based on the parameters received and configures the eDMA channel for the first transfer; upon completion, the second TCD from the list will be loaded and the channel will be ready to start the new transfer when a new request is received.
The application must allocate memory for the TCD list passed to this function (with an extra 32-bytes buffer, as the TCD structures need to be 32 bytes aligned); nevertheless, the driver will take care of initializing the array of descriptors, based on the other parameters passed. The function also received two lists of scatter/gather configuration structures (for source and destination, respectively), which define the address, length and type for each transfer. Besides these, the other parameters received are the transfer size, the number of bytes to be transferred on each request and the number of TCD structures to be used. This method will initialize all the descriptors according to user input and link them together; the linkage is done by writing the address of the next descriptor in the appropriate field of each one, similar to a linked-list data structure. The first descriptor is also copied to the TCD registers of the selected channel; if no errors are returned, after calling this function the channel is configured for the transfer defined by the first descriptor.

Virtual Channel Definition

The virtual channel is used to map multiple hardware channels across multiple eDMA instances. If only one eDMA instance is available, then the virtual channels will map one-on-one with the hardware channels. If more than one eDMA instance is available, then the virtual channels will map continuously and linearly over all of the harware channels. Example: If the SOC has 4 eDMAs modules each with 32 channels, then the user will be able to address a total of 128 virtual channels, that seamlessly map onto the hardware channels.

Virtual Channel Control

The eDMA driver provides functions that allow the user to start, stop, allocate and release an eDMA virtual channel. The EDMA_DRV_StartChannel() enables the eDMA requests for a virtual channel; this function should be called when the virtual channel is already initialized, as the first request received after the function call will trigger the transfer based on the current values of the virtual channel's TCD registers.
The EDMA_DRV_StopChannel() function disables requests for the selected virtual channel; this function should be called whenever the application needs to ignore eDMA requests for a virtual channel. It is automatically called when the virtual channel is released.
The EDMA_DRV_RequestChannel() function selects a virtual channel to be used by application and updates the driver state structure accordingly. Two types of virtual channel allocation are available:

- static: the user passes the virtual channel number as parameter; if the virtual channel is already allocated, the function returns an error;

- dynamic: the driver allocates the first available virtual channel and returns its number (or an error if no channel is availabe).

The EDMA_DRV_ReleaseChannel() function frees the hardware and software resources allocated for that virtual channel; it clears the virtual channel state structure, updates the driver state and disables requests for that virtual channel.

Important Notes

- Before using the eDMA driver the clock for eDMA and DMAMUX modules must be configured

- The driver enables the interrupts for the eDMA module, but any interrupt priority must be done by the application

- When using the modulo feature, application is responsible with ensuring that the source/destination address is properly aligned on a modulo-size boudary.

Data Structures

- struct edma_user_config_t

  The user configuration structure for the eDMA driver. More...
- struct edma_chn_state_t

  Data structure for the eDMA channel state. Implements : edma_chn_state_t_Class. More...
- struct edma_channel_config_t

  The user configuration structure for the an eDMA driver channel. More...
- struct edma_scatter_gather_list_t

  Data structure for configuring a discrete memory transfer. Implements : edma_scatter_gather_list_t_Class. More...
- struct edma_state_t

  Runtime state structure for the eDMA driver. More...
- struct edma_loop_transfer_config_t

  eDMA loop transfer configuration. More...
- struct edma_transfer_config_t

  eDMA transfer size configuration. More...

Macros

- #define STCD_SIZE(number) (((number) ∗ 32U) - 1U)

  Macro for the memory size needed for the software TCD.
- #define STCD_ADDR(address) (((uint32_t)address + 31UL) & ∼0x1FUL)
- #define EDMA_ERR_LSB_MASK 1U

  Macro for accessing the least significant bit of the ERR register.

Typedefs

- typedef void(∗ edma_callback_t) (void ∗parameter, edma_chn_status_t status)

  Definition for the eDMA channel callback function.

Enumerations

- enum edma_channel_interrupt_t { EDMA_CHN_ERR_INT = 0U, EDMA_CHN_HALF_MAJOR_LOOP_INT, EDMA_CHN_MAJOR_LOOP_INT }

  eDMA channel interrupts. Implements : edma_channel_interrupt_t_Class
- enum edma_arbitration_algorithm_t { EDMA_ARBITRATION_FIXED_PRIORITY = 0U, EDMA_ARBITRAT↩ION_ROUND_ROBIN }

  eDMA channel arbitration algorithm used for selection among channels. Implements : edma_arbitration_algorithm↩_t_Class

- enum edma_channel_priority_t {
  EDMA_CHN_PRIORITY_0 = 0U, EDMA_CHN_PRIORITY_1 = 1U, EDMA_CHN_PRIORITY_2 = 2U, EDM←
  A_CHN_PRIORITY_3 = 3U,
  EDMA_CHN_PRIORITY_4 = 4U, EDMA_CHN_PRIORITY_5 = 5U, EDMA_CHN_PRIORITY_6 = 6U, EDM←
  A_CHN_PRIORITY_7 = 7U,
  EDMA_CHN_PRIORITY_8 = 8U, EDMA_CHN_PRIORITY_9 = 9U, EDMA_CHN_PRIORITY_10 = 10U, E←
  DMA_CHN_PRIORITY_11 = 11U,
  EDMA_CHN_PRIORITY_12 = 12U, EDMA_CHN_PRIORITY_13 = 13U, EDMA_CHN_PRIORITY_14 = 14U,
  EDMA_CHN_PRIORITY_15 = 15U,
  EDMA_CHN_DEFAULT_PRIORITY = 255U }

  eDMA channel priority setting Implements : edma_channel_priority_t_Class

- enum edma_modulo_t {
  EDMA_MODULO_OFF = 0U, EDMA_MODULO_2B, EDMA_MODULO_4B, EDMA_MODULO_8B,
  EDMA_MODULO_16B, EDMA_MODULO_32B, EDMA_MODULO_64B, EDMA_MODULO_128B,
  EDMA_MODULO_256B, EDMA_MODULO_512B, EDMA_MODULO_1KB, EDMA_MODULO_2KB,
  EDMA_MODULO_4KB, EDMA_MODULO_8KB, EDMA_MODULO_16KB, EDMA_MODULO_32KB,
  EDMA_MODULO_64KB, EDMA_MODULO_128KB, EDMA_MODULO_256KB, EDMA_MODULO_512KB,
  EDMA_MODULO_1MB, EDMA_MODULO_2MB, EDMA_MODULO_4MB, EDMA_MODULO_8MB,
  EDMA_MODULO_16MB, EDMA_MODULO_32MB, EDMA_MODULO_64MB, EDMA_MODULO_128MB,
  EDMA_MODULO_256MB, EDMA_MODULO_512MB, EDMA_MODULO_1GB, EDMA_MODULO_2GB }

  eDMA modulo configuration Implements : edma_modulo_t_Class

- enum edma_transfer_size_t { EDMA_TRANSFER_SIZE_1B = 0x0U, EDMA_TRANSFER_SIZE_2B = 0x1U,
  EDMA_TRANSFER_SIZE_4B = 0x2U }

  eDMA transfer configuration Implements : edma_transfer_size_t_Class

- enum edma_chn_status_t { EDMA_CHN_NORMAL = 0U, EDMA_CHN_ERROR }

  Channel status for eDMA channel.

- enum edma_transfer_type_t { EDMA_TRANSFER_PERIPH2MEM = 0U, EDMA_TRANSFER_MEM2PERI←
  PH, EDMA_TRANSFER_MEM2MEM, EDMA_TRANSFER_PERIPH2PERIPH }

  A type for the DMA transfer. Implements : edma_transfer_type_t_Class.

eDMA peripheral driver module level functions

- status_t EDMA_DRV_Init (edma_state_t ∗edmaState, const edma_user_config_t ∗userConfig, edma_chn←
  _state_t ∗const chnStateArray[ ], const edma_channel_config_t ∗const chnConfigArray[ ], uint32_t chnCount)

  Initializes the eDMA module.

- status_t EDMA_DRV_Deinit (void)

  De-initializes the eDMA module.

eDMA peripheral driver channel management functions

- status_t EDMA_DRV_ChannelInit (edma_chn_state_t ∗edmaChannelState, const edma_channel_config_t
  ∗edmaChannelConfig)

  Initializes an eDMA channel.

- status_t EDMA_DRV_ReleaseChannel (uint8_t virtualChannel)

  Releases an eDMA channel.

eDMA peripheral driver transfer setup functions

- void EDMA_DRV_PushConfigToReg (uint8_t virtualChannel, const edma_transfer_config_t ∗tcd)

  Copies the channel configuration to the TCD registers.

- void EDMA_DRV_PushConfigToSTCD (const edma_transfer_config_t ∗config, edma_software_tcd_t ∗stcd)

  Copies the channel configuration to the software TCD structure.

- void EDMA_DRV_DisableRequestsOnTransferComplete (uint8_t virtualChannel, bool disable)

    Disables/Enables the DMA request after the major loop completes for the TCD.

- void EDMA_DRV_ConfigureInterrupt (uint8_t virtualChannel, edma_channel_interrupt_t intSrc, bool enable)

    Disables/Enables the channel interrupt requests.

- void EDMA_DRV_TriggerSwRequest (uint8_t virtualChannel)

    Triggers a sw request for the current channel.

eDMA Peripheral callback and interrupt functions

- status_t EDMA_DRV_InstallCallback (uint8_t virtualChannel, edma_callback_t callback, void ∗parameter)

    Registers the callback function and the parameter for eDMA channel.

eDMA Peripheral driver miscellaneous functions

- edma_chn_status_t EDMA_DRV_GetChannelStatus (uint8_t virtualChannel)

    Gets the eDMA channel status.

16.17.2   Data Structure Documentation

16.17.2.1   struct edma_user_config_t

The user configuration structure for the eDMA driver.

Use an instance of this structure with the EDMA_DRV_Init() function. This allows the user to configure settings of the EDMA peripheral with a single function call. Implements : edma_user_config_t_Class

Definition at line 235 of file edma_driver.h.

Data Fields

- edma_arbitration_algorithm_t chnArbitration
- bool haltOnError

Field Documentation

16.17.2.1.1   edma_arbitration_algorithm_t chnArbitration

eDMA channel arbitration.

Definition at line 236 of file edma_driver.h.

16.17.2.1.2   bool haltOnError

Any error causes the HALT bit to set. Subsequently, all service requests are ignored until the HALT bit is cleared.

Definition at line 244 of file edma_driver.h.

16.17.2.2   struct edma_chn_state_t

Data structure for the eDMA channel state. Implements : edma_chn_state_t_Class.

Definition at line 271 of file edma_driver.h.

Data Fields

- uint8_t virtChn
- edma_callback_t callback
- void ∗ parameter
- volatile edma_chn_status_t status

Field Documentation

16.17.2.2.1   edma_callback_t callback

Callback function pointer for the eDMA channel. It will be called at the eDMA channel complete and eDMA channel error.

Definition at line 273 of file edma_driver.h.

16.17.2.2.2   void∗ parameter

Parameter for the callback function pointer.

Definition at line 276 of file edma_driver.h.

16.17.2.2.3   volatile edma_chn_status_t status

eDMA channel status.

Definition at line 277 of file edma_driver.h.

16.17.2.2.4   uint8_t virtChn

Virtual channel number.

Definition at line 272 of file edma_driver.h.

16.17.2.3   struct edma_channel_config_t

The user configuration structure for the an eDMA driver channel.

Use an instance of this structure with the EDMA_DRV_ChannelInit() function. This allows the user to configure settings of the EDMA channel with a single function call. Implements : edma_channel_config_t_Class

Definition at line 287 of file edma_driver.h.

Data Fields

- edma_channel_priority_t channelPriority
- uint8_t virtChnConfig
- dma_request_source_t source
- edma_callback_t callback
- void ∗ callbackParam

Field Documentation

16.17.2.3.1   edma_callback_t callback

Callback that will be registered for this channel

Definition at line 298 of file edma_driver.h.

16.17.2.3.2   void∗ callbackParam

Parameter passed to the channel callback

Definition at line 299 of file edma_driver.h.

16.17.2.3.3   edma_channel_priority_t channelPriority

eDMA channel priority - only used when channel arbitration mode is 'Fixed priority'.

Definition at line 294 of file edma_driver.h.

16.17.2.3.4   dma_request_source_t source

Selects the source of the DMA request for this channel

Definition at line 297 of file edma_driver.h.

16.17.2.3.5   uint8_t virtChnConfig

eDMA virtual channel number

Definition at line 296 of file edma_driver.h.

16.17.2.4   struct edma_scatter_gather_list_t

Data structure for configuring a discrete memory transfer. Implements : edma_scatter_gather_list_t_Class.

Definition at line 315 of file edma_driver.h.

Data Fields

- uint32_t address
- uint32_t length
- edma_transfer_type_t type

Field Documentation

16.17.2.4.1   uint32_t address

Address of buffer.

Definition at line 316 of file edma_driver.h.

16.17.2.4.2   uint32_t length

Length of buffer.

Definition at line 317 of file edma_driver.h.

16.17.2.4.3   edma_transfer_type_t type

Type of the DMA transfer

Definition at line 318 of file edma_driver.h.

16.17.2.5   struct edma_state_t

Runtime state structure for the eDMA driver.

This structure holds data that is used by the eDMA peripheral driver to manage multi eDMA channels. The user passes the memory for this run-time state structure and the eDMA driver populates the members. Implements : edma_state_t_Class

Definition at line 330 of file edma_driver.h.

Data Fields

- edma_chn_state_t *volatile virtChnState [(uint32_t) FEATURE_DMA_VIRTUAL_CHANNELS]

Field Documentation

16.17.2.5.1   edma_chn_state_t∗ volatile virtChnState[(uint32_t) FEATURE_DMA_VIRTUAL_CHANNELS]

Pointer array storing channel state.

Definition at line 331 of file edma_driver.h.

16.17.2.6   struct edma_loop_transfer_config_t

eDMA loop transfer configuration.

This structure configures the basic minor/major loop attributes. Implements : edma_loop_transfer_config_t_Class

Definition at line 340 of file edma_driver.h.

Data Fields

- uint32_t majorLoopIterationCount
- bool srcOffsetEnable
- bool dstOffsetEnable
- int32_t minorLoopOffset
- bool minorLoopChnLinkEnable
- uint8_t minorLoopChnLinkNumber
- bool majorLoopChnLinkEnable
- uint8_t majorLoopChnLinkNumber

Field Documentation

16.17.2.6.1   bool dstOffsetEnable

Selects whether the minor loop offset is applied to the destination address upon minor loop completion.

Definition at line 344 of file edma_driver.h.

16.17.2.6.2   bool majorLoopChnLinkEnable

Enables channel-to-channel linking on major loop complete.

Definition at line 351 of file edma_driver.h.

16.17.2.6.3   uint8_t majorLoopChnLinkNumber

The number of the next channel to be started by DMA engine when major loop completes.

Definition at line 352 of file edma_driver.h.

16.17.2.6.4   uint32_t majorLoopIterationCount

Number of major loop iterations.

Definition at line 341 of file edma_driver.h.

16.17.2.6.5   bool minorLoopChnLinkEnable

Enables channel-to-channel linking on minor loop complete.

Definition at line 348 of file edma_driver.h.

16.17.2.6.6   uint8_t minorLoopChnLinkNumber

The number of the next channel to be started by DMA engine when minor loop completes.

Definition at line 349 of file edma_driver.h.

16.17.2.6.7   int32_t minorLoopOffset

Sign-extended offset applied to the source or destination address to form the next-state value after the minor loop completes.

Definition at line 346 of file edma_driver.h.

16.17.2.6.8 bool srcOffsetEnable

Selects whether the minor loop offset is applied to the source address upon minor loop completion.

Definition at line 342 of file edma_driver.h.

16.17.2.7 struct edma_transfer_config_t

eDMA transfer size configuration.

This structure configures the basic source/destination transfer attribute. Implements : edma_transfer_config_t_↩
Class

Definition at line 362 of file edma_driver.h.

Data Fields

- uint32_t srcAddr
- uint32_t destAddr
- edma_transfer_size_t srcTransferSize
- edma_transfer_size_t destTransferSize
- int16_t srcOffset
- int16_t destOffset
- int32_t srcLastAddrAdjust
- int32_t destLastAddrAdjust
- edma_modulo_t srcModulo
- edma_modulo_t destModulo
- uint32_t minorByteTransferCount
- bool scatterGatherEnable
- uint32_t scatterGatherNextDescAddr
- bool interruptEnable
- edma_loop_transfer_config_t * loopTransferConfig

Field Documentation

16.17.2.7.1 uint32_t destAddr

Memory address pointing to the destination data.

Definition at line 364 of file edma_driver.h.

16.17.2.7.2 int32_t destLastAddrAdjust

Last destination address adjustment. Note here it is only valid when scatter/gather feature is not enabled.

Definition at line 374 of file edma_driver.h.

16.17.2.7.3 edma_modulo_t destModulo

Destination address modulo.

Definition at line 377 of file edma_driver.h.

16.17.2.7.4 int16_t destOffset

Sign-extended offset applied to the current destination address to form the next-state value as each source read/write is completed.

Definition at line 370 of file edma_driver.h.

16.17.2.7.5 edma_transfer_size_t destTransferSize

Destination data transfer size.

Definition at line 366 of file edma_driver.h.

16.17.2.7.6   bool interruptEnable

Enable the interrupt request when the major loop count completes

Definition at line 385 of file edma_driver.h.

16.17.2.7.7   edma_loop_transfer_config_t∗ loopTransferConfig

Pointer to loop transfer configuration structure (defines minor/major loop attributes) Note: this field is only used when minor loop mapping is enabled from DMA configuration.

Definition at line 387 of file edma_driver.h.

16.17.2.7.8   uint32_t minorByteTransferCount

Number of bytes to be transferred in each service request of the channel.

Definition at line 378 of file edma_driver.h.

16.17.2.7.9   bool scatterGatherEnable

Enable scatter gather feature.

Definition at line 380 of file edma_driver.h.

16.17.2.7.10   uint32_t scatterGatherNextDescAddr

The address of the next descriptor to be used, when scatter/gather feature is enabled. Note: this value is not used when scatter/gather feature is disabled.

Definition at line 381 of file edma_driver.h.

16.17.2.7.11   uint32_t srcAddr

Memory address pointing to the source data.

Definition at line 363 of file edma_driver.h.

16.17.2.7.12   int32_t srcLastAddrAdjust

Last source address adjustment.

Definition at line 373 of file edma_driver.h.

16.17.2.7.13   edma_modulo_t srcModulo

Source address modulo.

Definition at line 376 of file edma_driver.h.

16.17.2.7.14   int16_t srcOffset

Sign-extended offset applied to the current source address to form the next-state value as each source read/write is completed.

Definition at line 367 of file edma_driver.h.

16.17.2.7.15   edma_transfer_size_t srcTransferSize

Source data transfer size.

Definition at line 365 of file edma_driver.h.

16.17.3   Macro Definition Documentation

16.17.3.1   #define EDMA_ERR_LSB_MASK 1U

Macro for accessing the least significant bit of the ERR register.

The erroneous channels are retrieved from ERR register by subsequently right shifting all the ERR bits + "AND"-ing the result with this mask.

Definition at line 68 of file edma_driver.h.

16.17.3.2   #define STCD_ADDR(*address* **) (((uint32_t)address + 31UL) &** ∼**0x1FUL)**

Definition at line 60 of file edma_driver.h.

16.17.3.3   #define STCD_SIZE(*number* **) (((number)** ∗ **32U) - 1U)**

Macro for the memory size needed for the software TCD.

Software TCD is aligned to 32 bytes. We don't need a software TCD structure for the first descriptor, since the configuration is pushed directly to registers. To make sure the software TCD can meet the eDMA module requirement regarding alignment, allocate memory for the remaining descriptors with extra 31 bytes.

Definition at line 59 of file edma_driver.h.

16.17.4   Typedef Documentation

16.17.4.1   typedef void(∗ **edma_callback_t) (void** ∗**parameter,** **edma_chn_status_t status)**

Definition for the eDMA channel callback function.

Prototype for the callback function registered in the eDMA driver. Implements : edma_callback_t_Class

Definition at line 266 of file edma_driver.h.

16.17.5   Enumeration Type Documentation

16.17.5.1   enum edma_arbitration_algorithm_t

eDMA channel arbitration algorithm used for selection among channels. Implements : edma_arbitration_algorithm←
_t_Class

Enumerator

> ***EDMA_ARBITRATION_FIXED_PRIORITY***   Fixed Priority
>
> ***EDMA_ARBITRATION_ROUND_ROBIN***   Round-Robin arbitration

Definition at line 82 of file edma_driver.h.

16.17.5.2   enum edma_channel_interrupt_t

eDMA channel interrupts. Implements : edma_channel_interrupt_t_Class

Enumerator

> ***EDMA_CHN_ERR_INT***   Error interrupt
>
> ***EDMA_CHN_HALF_MAJOR_LOOP_INT***   Half major loop interrupt.
>
> ***EDMA_CHN_MAJOR_LOOP_INT***   Complete major loop interrupt.

Definition at line 73 of file edma_driver.h.

16.17.5.3 enum **edma_channel_priority_t**

eDMA channel priority setting Implements : edma_channel_priority_t_Class

Enumerator

> ***EDMA_CHN_PRIORITY_0***
> ***EDMA_CHN_PRIORITY_1***
> ***EDMA_CHN_PRIORITY_2***
> ***EDMA_CHN_PRIORITY_3***
> ***EDMA_CHN_PRIORITY_4***
> ***EDMA_CHN_PRIORITY_5***
> ***EDMA_CHN_PRIORITY_6***
> ***EDMA_CHN_PRIORITY_7***
> ***EDMA_CHN_PRIORITY_8***
> ***EDMA_CHN_PRIORITY_9***
> ***EDMA_CHN_PRIORITY_10***
> ***EDMA_CHN_PRIORITY_11***
> ***EDMA_CHN_PRIORITY_12***
> ***EDMA_CHN_PRIORITY_13***
> ***EDMA_CHN_PRIORITY_14***
> ***EDMA_CHN_PRIORITY_15***
> ***EDMA_CHN_DEFAULT_PRIORITY***

Definition at line 90 of file edma_driver.h.

16.17.5.4 enum **edma_chn_status_t**

Channel status for eDMA channel.

A structure describing the eDMA channel status. The user can get the status by callback parameter or by calling EDMA_DRV_getStatus() function. Implements : edma_chn_status_t_Class

Enumerator

> ***EDMA_CHN_NORMAL*** eDMA channel normal state.
> ***EDMA_CHN_ERROR*** An error occurred in the eDMA channel.

Definition at line 255 of file edma_driver.h.

16.17.5.5 enum **edma_modulo_t**

eDMA modulo configuration Implements : edma_modulo_t_Class

Enumerator

> ***EDMA_MODULO_OFF***
> ***EDMA_MODULO_2B***
> ***EDMA_MODULO_4B***
> ***EDMA_MODULO_8B***
> ***EDMA_MODULO_16B***
> ***EDMA_MODULO_32B***
> ***EDMA_MODULO_64B***

EDMA_MODULO_128B

EDMA_MODULO_256B

EDMA_MODULO_512B

EDMA_MODULO_1KB

EDMA_MODULO_2KB

EDMA_MODULO_4KB

EDMA_MODULO_8KB

EDMA_MODULO_16KB

EDMA_MODULO_32KB

EDMA_MODULO_64KB

EDMA_MODULO_128KB

EDMA_MODULO_256KB

EDMA_MODULO_512KB

EDMA_MODULO_1MB

EDMA_MODULO_2MB

EDMA_MODULO_4MB

EDMA_MODULO_8MB

EDMA_MODULO_16MB

EDMA_MODULO_32MB

EDMA_MODULO_64MB

EDMA_MODULO_128MB

EDMA_MODULO_256MB

EDMA_MODULO_512MB

EDMA_MODULO_1GB

EDMA_MODULO_2GB

Definition at line 162 of file edma_driver.h.

16.17.5.6   enum edma_transfer_size_t

eDMA transfer configuration Implements : edma_transfer_size_t_Class

Enumerator

EDMA_TRANSFER_SIZE_1B

EDMA_TRANSFER_SIZE_2B

EDMA_TRANSFER_SIZE_4B

Definition at line 200 of file edma_driver.h.

16.17.5.7   enum edma_transfer_type_t

A type for the DMA transfer. Implements : edma_transfer_type_t_Class.

Enumerator

EDMA_TRANSFER_PERIPH2MEM   Transfer from peripheral to memory

EDMA_TRANSFER_MEM2PERIPH   Transfer from memory to peripheral

EDMA_TRANSFER_MEM2MEM   Transfer from memory to memory

EDMA_TRANSFER_PERIPH2PERIPH   Transfer from peripheral to peripheral

Definition at line 305 of file edma_driver.h.

16.17.6 Function Documentation

16.17.6.1 void EDMA_DRV_CancelTransfer (bool *error*)

Cancel the running transfer.

This function cancels the current transfer, optionally signalling an error.

Parameters

bool	error If true, an error will be logged for the current transfer.

Definition at line 1481 of file edma_driver.c.

16.17.6.2 status_t EDMA_DRV_ChannelInit (edma_chn_state_t ∗ *edmaChannelState,* const edma_channel_config_t ∗ *edmaChannelConfig*)

Initializes an eDMA channel.

This function initializes the run-time state structure for a eDMA channel, based on user configuration. It will request the channel, set up the channel priority and install the callback.

Parameters

edmaChannel↩ State	Pointer to the eDMA channel state structure. The user passes the memory for this run-time state structure and the eDMA peripheral driver populates the members. This run-time state structure keeps track of the eDMA channel status. The memory must be kept valid before calling the EDMA_DRV_ReleaseChannel.
edmaChannel↩ Config	User configuration structure for eDMA channel. The user populates the members of this structure and passes the pointer of this structure into the function.

Returns

     STATUS_ERROR or STATUS_SUCCESS.

Definition at line 287 of file edma_driver.c.

16.17.6.3 void EDMA_DRV_ClearTCD (uint8_t *virtualChannel*)

Clears all registers to 0 for the channel's TCD.

Parameters

virtualChannel	eDMA virtual channel number.

Definition at line 1013 of file edma_driver.c.

16.17.6.4 status_t EDMA_DRV_ConfigLoopTransfer (uint8_t *virtualChannel,* const edma_transfer_config_t ∗ *transferConfig*)

Configures the DMA transfer in loop mode.

This function configures the DMA transfer in a loop chain. The user passes a block of memory into this function that configures the loop transfer properties (minor/major loop count, address offsets, channel linking). The DMA driver copies the configuration to TCD registers, only when the loop properties are set up correctly and minor loop mapping is enabled for the eDMA module.

Parameters

virtualChannel	eDMA virtual channel number.

transferConfig	Pointer to the transfer configuration strucutre; this structure defines fields for setting up the basic transfer and also a pointer to a memory strucure that defines the loop chain properties (minor/major).

Returns

> STATUS_ERROR or STATUS_SUCCESS

Definition at line 703 of file edma_driver.c.

16.17.6.5   status_t EDMA_DRV_ConfigMultiBlockTransfer (uint8_t *virtualChannel,* edma_transfer_type_t *type,* uint32_t *srcAddr,* uint32_t *destAddr,* edma_transfer_size_t *transferSize,* uint32_t *blockSize,* uint32_t *blockCount,* bool *disableReqOnCompletion*)

Configures a multiple block data transfer with DMA.

This function configures the descriptor for a multi-block transfer. The function considers contiguous memory blocks, thus it configures the TCD source/destination offset fields to cover the data buffer without gaps, according to "transferSize" parameter (the offset is equal to the number of bytes transferred in a source read/destination write). The buffer is divided in multiple block, each block being transferred upon a single DMA request.

NOTE: For transfers to/from peripherals, make sure the transfer size is equal to the data buffer size of the peripheral used, otherwise only truncated chunks of data may be transferred (e.g. for a communication IP with an 8-bit data register the transfer size should be 1B, whereas for a 32-bit data register, the transfer size should be 4B). The rationale of this constraint is that, on the peripheral side, the address offset is set to zero, allowing to read/write data from/to the peripheral in a single source read/destination write operation.

Parameters

virtualChannel	eDMA virtual channel number.
type	Transfer type (M->M, P->M, M->P, P->P).
srcAddr	A source register address or a source memory address.
destAddr	A destination register address or a destination memory address.
transferSize	The number of bytes to be transferred on every DMA write/read. Source/Dest share the same write/read size.
blockSize	The total number of bytes inside a block.
blockCount	The total number of data blocks (one block is transferred upon a DMA request).
disableReqOn↩ Completion	This parameter specifies whether the DMA channel should be disabled when the transfer is complete (further requests will remain untreated).

Returns

> STATUS_ERROR or STATUS_SUCCESS

Definition at line 657 of file edma_driver.c.

16.17.6.6   status_t EDMA_DRV_ConfigScatterGatherTransfer (uint8_t *virtualChannel,* edma_software_tcd_t ∗ *stcd,* edma_transfer_size_t *transferSize,* uint32_t *bytesOnEachRequest,* const edma_scatter_gather_list_t ∗ *srcList,* const edma_scatter_gather_list_t ∗ *destList,* uint8_t *tcdCount*)

Configures the DMA transfer in a scatter-gather mode.

This function configures the descriptors into a single-ended chain. The user passes blocks of memory into this function. The interrupt is triggered only when the last memory block is completed. The memory block information is passed with the edma_scatter_gather_list_t data structure, which can tell the memory address and length. The DMA driver configures the descriptor for each memory block, transfers the descriptor from the first one to the last one, and stops.

Parameters

virtualChannel	eDMA virtual channel number.
stcd	Array of empty software TCD structures. The user must prepare this memory block. We don't need a software TCD structure for the first descriptor, since the configuration is pushed directly to registers.The "stcd" buffer must align with 32 bytes; if not, an error occurs in the e↩ DMA driver. Thus, the required memory size for "stcd" is equal to tcdCount ∗ size_of(edma↩ _software_tcd_t) - 1; the driver will take care of the memory alignment if the provided memory buffer is big enough. For proper allocation of the "stcd" buffer it is recommended to use ST↩ CD_SIZE macro.
transferSize	The number of bytes to be transferred on every DMA write/read.
bytesOnEach↩ Request	Bytes to be transferred in each DMA request.
srcList	Data structure storing the address, length and type of transfer (M->M, M->P, P->M, P->P) for the bytes to be transferred for source memory blocks. If the source memory is peripheral, the length is not used.
destList	Data structure storing the address, length and type of transfer (M->M, M->P, P->M, P->P) for the bytes to be transferred for destination memory blocks. In the memory-to-memory transfer mode, the user must ensure that the length of the destination scatter gather list is equal to the source scatter gather list. If the destination memory is a peripheral register, the length is not used.
tcdCount	The number of TCD memory blocks contained in the scatter gather list.

Returns

> STATUS_ERROR or STATUS_SUCCESS

Definition at line 757 of file edma_driver.c.

16.17.6.7   status_t EDMA_DRV_ConfigSingleBlockTransfer (uint8_t *virtualChannel,* edma_transfer_type_t *type,* uint32_t *srcAddr,* uint32_t *destAddr,* edma_transfer_size_t *transferSize,* uint32_t *dataBufferSize*)

Configures a simple single block data transfer with DMA.

This function configures the descriptor for a single block transfer. The function considers contiguous memory blocks, thus it configures the TCD source/destination offset fields to cover the data buffer without gaps, according to "transferSize" parameter (the offset is equal to the number of bytes transferred in a source read/destination write).

NOTE: For memory-to-peripheral or peripheral-to-memory transfers, make sure the transfer size is equal to the data buffer size of the peripheral used, otherwise only truncated chunks of data may be transferred (e.g. for a communication IP with an 8-bit data register the transfer size should be 1B, whereas for a 32-bit data register, the transfer size should be 4B). The rationale of this constraint is that, on the peripheral side, the address offset is set to zero, allowing to read/write data from/to the peripheral in a single source read/destination write operation.

Parameters

virtualChannel	eDMA virtual channel number.
type	Transfer type (M->M, P->M, M->P, P->P).
srcAddr	A source register address or a source memory address.
destAddr	A destination register address or a destination memory address.
transferSize	The number of bytes to be transferred on every DMA write/read. Source/Dest share the same write/read size.
dataBufferSize	The total number of bytes to be transferred.

Returns

> STATUS_ERROR or STATUS_SUCCESS

Definition at line 554 of file edma_driver.c.

16.17.6.8 void EDMA_DRV_ConfigureInterrupt (uint8_t *virtualChannel,* edma_channel_interrupt_t *intSrc,* bool *enable*)

Disables/Enables the channel interrupt requests.

This function enables/disables error, half major loop and complete major loop interrupts for the current channel.

Parameters

virtualChannel	eDMA virtual channel number.
interrupt	Interrupt event (error/half major loop/complete major loop).
enable	Enable (true)/Disable (false) interrupts for the current channel.

Definition at line 1433 of file edma_driver.c.

16.17.6.9 status_t EDMA_DRV_Deinit (void)

De-initializes the eDMA module.

This function resets the eDMA module to reset state and disables the interrupt to the core.

Returns

STATUS_ERROR or STATUS_SUCCESS.

Definition at line 239 of file edma_driver.c.

16.17.6.10 void EDMA_DRV_DisableRequestsOnTransferComplete (uint8_t *virtualChannel,* bool *disable*)

Disables/Enables the DMA request after the major loop completes for the TCD.

If disabled, the eDMA hardware automatically clears the corresponding DMA request when the current major iteration count reaches zero.

Parameters

virtualChannel	eDMA virtual channel number.
disable	Disable (true)/Enable (false) DMA request after TCD complete.

Definition at line 1403 of file edma_driver.c.

16.17.6.11 edma_chn_status_t EDMA_DRV_GetChannelStatus (uint8_t *virtualChannel*)

Gets the eDMA channel status.

Parameters

virtualChannel	eDMA virtual channel number.

Returns

Channel status.

Definition at line 1701 of file edma_driver.c.

16.17.6.12 uint32_t EDMA_DRV_GetRemainingMajorIterationsCount (uint8_t *virtualChannel*)

Returns the remaining major loop iteration count.

Gets the number minor loops yet to be triggered (major loop iterations).

Parameters

virtualChannel	eDMA virtual channel number.

Returns

number of major loop iterations yet to be triggered

Definition at line 1342 of file edma_driver.c.

16.17.6.13   status_t EDMA_DRV_Init (edma_state_t ∗ *edmaState,* const edma_user_config_t ∗ *userConfig,* edma_chn_state_t ∗const *chnStateArray[ ],* const edma_channel_config_t ∗const *chnConfigArray[ ],* uint32_t *chnCount*)

Initializes the eDMA module.

This function initializes the run-time state structure to provide the eDMA channel allocation release, protect, and track the state for channels. This function also resets the eDMA modules, initializes the module to user-defined settings and default settings.

Parameters

edmaState	The pointer to the eDMA peripheral driver state structure. The user passes the memory for this run-time state structure and the eDMA peripheral driver populates the members. This run-time state structure keeps track of the eDMA channels status. The memory must be kept valid before calling the EDMA_DRV_DeInit.
userConfig	User configuration structure for eDMA peripheral drivers. The user populates the members of this structure and passes the pointer of this structure into the function.
chnStateArray	Array of pointers to run-time state structures for eDMA channels; will populate the state structures inside the eDMA driver state structure.
chnConfigArray	Array of pointers to channel initialization structures.
chnCount	The number of eDMA channels to be initialized.

Returns

STATUS_ERROR or STATUS_SUCCESS.

Definition at line 119 of file edma_driver.c.

16.17.6.14   status_t EDMA_DRV_InstallCallback (uint8_t *virtualChannel,* edma_callback_t *callback,* void ∗ *parameter*)

Registers the callback function and the parameter for eDMA channel.

This function registers the callback function and the parameter into the eDMA channel state structure. The callback function is called when the channel is complete or a channel error occurs. The eDMA driver passes the channel status to this callback function to indicate whether it is caused by the channel complete event or the channel error event.

To un-register the callback function, set the callback function to "NULL" and call this function.

Parameters

virtualChannel	eDMA virtual channel number.
callback	The pointer to the callback function.
parameter	The pointer to the callback function's parameter.

Returns

STATUS_ERROR or STATUS_SUCCESS.

Definition at line 336 of file edma_driver.c.

16.17.6.15   void EDMA_DRV_PushConfigToReg (uint8_t *virtualChannel,* const edma_transfer_config_t ∗ *tcd*)

Copies the channel configuration to the TCD registers.

Parameters

virtualChannel	eDMA virtual channel number.
tcd	Pointer to the channel configuration structure.

Definition at line 1583 of file edma_driver.c.

16.17.6.16   void EDMA_DRV_PushConfigToSTCD (const edma_transfer_config_t ∗ config, edma_software_tcd_t ∗ stcd)

Copies the channel configuration to the software TCD structure.

This function copies the properties from the channel configuration to the software TCD structure; the address of the software TCD can be used to enable scatter/gather operation (pointer to the next TCD).

Parameters

config	Pointer to the channel configuration structure.
stcd	Pointer to the software TCD structure.

Definition at line 1539 of file edma_driver.c.

16.17.6.17   status_t EDMA_DRV_ReleaseChannel (uint8_t *virtualChannel*)

Releases an eDMA channel.

This function stops the eDMA channel and disables the interrupt of this channel. The channel state structure can be released after this function is called.

Parameters

virtualChannel	eDMA virtual channel number.

Returns

STATUS_ERROR or STATUS_SUCCESS.

Definition at line 421 of file edma_driver.c.

16.17.6.18   status_t EDMA_DRV_SetChannelRequest (uint8_t *virtualChannel,* uint8_t *req*)

Configures the DMA request for the eDMA channel.

Selects which DMA source is routed to a DMA channel. The DMA sources are defined in the file <MCU>_↵ Features.h

Parameters

virtualChannel	eDMA virtual channel number.
req	DMA request source.

Returns

STATUS_SUCCESS.

Definition at line 979 of file edma_driver.c.

16.17.6.19   void EDMA_DRV_SetDestAddr (uint8_t *virtualChannel,* uint32_t *address*)

Configures the destination address for the eDMA channel.

Parameters

virtualChannel	eDMA virtual channel number.
address	The pointer to the destination memory address.

Definition at line 1192 of file edma_driver.c.

16.17.6.20 void EDMA_DRV_SetDestLastAddrAdjustment (uint8_t *virtualChannel,* int32_t *adjust*)

Configures the destination address last adjustment.

Adjustment value added to the destination address at the completion of the major iteration count. This value can be applied to restore the destination address to the initial value, or adjust the address to reference the next data structure.

Parameters

virtualChannel	eDMA virtual channel number.
adjust	Adjustment value.

Definition at line 1162 of file edma_driver.c.

16.17.6.21 void EDMA_DRV_SetDestOffset (uint8_t *virtualChannel,* int16_t *offset*)

Configures the destination address signed offset for the eDMA channel.

Sign-extended offset applied to the current destination address to form the next-state value as each destination write is complete.

Parameters

virtualChannel	eDMA virtual channel number.
offset	signed-offset

Definition at line 1222 of file edma_driver.c.

16.17.6.22 void EDMA_DRV_SetDestWriteChunkSize (uint8_t *virtualChannel,* edma_transfer_size_t *size*)

Configures the destination data chunk size (transferred in a write sequence).

Destination data write transfer size (1/2/4/16/32 bytes).

Parameters

virtualChannel	eDMA virtual channel number.
size	Destination transfer size.

Definition at line 1252 of file edma_driver.c.

16.17.6.23 void EDMA_DRV_SetMajorLoopIterationCount (uint8_t *virtualChannel,* uint32_t *majorLoopCount*)

Configures the number of major loop iterations.

Sets the number of major loop iterations; each major loop iteration will be served upon a request for the current channel, transferring the data block configured for the minor loop (NBYTES).

Parameters

virtualChannel	eDMA virtual channel number.
majorLoopCount	Number of major loop iterations.

Definition at line 1312 of file edma_driver.c.

16.17.6.24 void EDMA_DRV_SetMinorLoopBlockSize (uint8_t *virtualChannel,* uint32_t *nbytes*)

Configures the number of bytes to be transferred in each service request of the channel.

Sets the number of bytes to be transferred each time a request is received (one major loop iteration). This number

needs to be a multiple of the source/destination transfer size, as the data block will be transferred within multiple read/write sequences (minor loops).

Parameters

virtualChannel	eDMA virtual channel number.
nbytes	Number of bytes to be transferred in each service request of the channel

Definition at line 1282 of file edma_driver.c.

16.17.6.25   void EDMA_DRV_SetScatterGatherLink (uint8_t *virtualChannel,* uint32_t *nextTCDAddr*)

Configures the memory address of the next TCD, in scatter/gather mode.

This function configures the address of the next TCD to be loaded form memory, when scatter/gather feature is enabled. This address points to the beginning of a 0-modulo-32 byte region containing the next transfer TCD to be loaded into this channel. The channel reload is performed as the major iteration count completes. The scatter/gather address must be 0-modulo-32-byte. Otherwise, a configuration error is reported.

Parameters

virtualChannel	eDMA virtual channel number.
nextTCDAddr	The address of the next TCD to be linked to this TCD.

Definition at line 1373 of file edma_driver.c.

16.17.6.26   void EDMA_DRV_SetSrcAddr (uint8_t *virtualChannel,* uint32_t *address*)

Configures the source address for the eDMA channel.

Parameters

virtualChannel	eDMA virtual channel number.
address	The pointer to the source memory address.

Definition at line 1042 of file edma_driver.c.

16.17.6.27   void EDMA_DRV_SetSrcLastAddrAdjustment (uint8_t *virtualChannel,* int32_t *adjust*)

Configures the source address last adjustment.

Adjustment value added to the source address at the completion of the major iteration count. This value can be applied to restore the source address to the initial value, or adjust the address to reference the next data structure.

Parameters

virtualChannel	eDMA virtual channel number.
adjust	Adjustment value.

Definition at line 1132 of file edma_driver.c.

16.17.6.28   void EDMA_DRV_SetSrcOffset (uint8_t *virtualChannel,* int16_t *offset*)

Configures the source address signed offset for the eDMA channel.

Sign-extended offset applied to the current source address to form the next-state value as each source read is complete.

Parameters

virtualChannel	eDMA virtual channel number.
offset	Signed-offset for source address.

Definition at line 1072 of file edma_driver.c.

16.17.6.29   void EDMA_DRV_SetSrcReadChunkSize (uint8_t *virtualChannel,* edma_transfer_size_t *size*)

Configures the source data chunk size (transferred in a read sequence).

Source data read transfer size (1/2/4/16/32 bytes).

Parameters

virtualChannel	eDMA virtual channel number.
size	Source transfer size.

Definition at line 1102 of file edma_driver.c.

16.17.6.30   status_t EDMA_DRV_StartChannel (uint8_t *virtualChannel*)

Starts an eDMA channel.

This function enables the eDMA channel DMA request.

Parameters

virtualChannel	eDMA virtual channel number.

Returns

> STATUS_ERROR or STATUS_SUCCESS.

Definition at line 917 of file edma_driver.c.

16.17.6.31   status_t EDMA_DRV_StopChannel (uint8_t *virtualChannel*)

Stops the eDMA channel.

This function disables the eDMA channel DMA request.

Parameters

virtualChannel	eDMA virtual channel number.

Returns

> STATUS_ERROR or STATUS_SUCCESS.

Definition at line 948 of file edma_driver.c.

16.17.6.32   void EDMA_DRV_TriggerSwRequest (uint8_t *virtualChannel*)

Triggers a sw request for the current channel.

This function starts a transfer using the current channel (sw request).

Parameters

virtualChannel	eDMA virtual channel number.

Definition at line 1510 of file edma_driver.c.

16.18   EIM Driver

16.18.1   Detailed Description

Error Injection Module Peripheral Driver.
EIM PD provides a set of high-level APIs/services to configure the Error Injection Module (EIM) module.

Basic Operations of EIM

1. To initialize EIM, call EIM_DRV_Init() with an user channel configuration array. In the following code, EIM is initialized with default settings (after reset) for check-bit mask and data mask and both channels is enabled.

```
#define INST_EIM1 (0U)

#define EIM_CHANNEL_COUNT0 (2U)
/* Configuration structure array */
eim_user_channel_config_t userChannelConfigArr[] =
{
    /* Configuration channel 0 */
    {
        .channel = 0x0U,
        .checkBitMask = 0x00U,
        .dataMask = 0x00U,
        .enable = true
    },
    /* Configuration channel 1 */
    {
        .channel = 0x1U,
        .checkBitMask = 0x00U,
        .dataMask = 0x00U,
        .enable = true
    }
};

/* Initialize the EIM instance 0 with configured channel number of 2 and userChannelConfigArr */
EIM_DRV_Init(INST_EIM1, EIM_CHANNEL_COUNT0 , userChannelConfigArr);
```

2. To get the default configuration (data mask, check-bit mask and enable status) of a channel in EIM, just call EIM_DRV_GetDefaultConfig(). Make sure that the operation is not execute in target RAM where EIM inject the error

```
eim_user_channel_config_t channelConfig;

/* Get default configuration of EIM channel 1*/
EIM_DRV_GetDefaultConfig(1U, &channelConfig);
```

3. To de-initialize EIM, just call the EIM_DRV_Deinit() function. This function sets all registers to reset values and disables EIM.

```
/* De-initializes the EIM module */
EIM_DRV_Deinit(INST_EIM1);
```

Data Structures

- struct eim_user_channel_config_t

  EIM channel configuration structure. More...

Macros

- #define EIM_CHECKBITMASK_DEFAULT (0x01U)

  The value default of EIM check-bit mask.
- #define EIM_DATAMASK_DEFAULT (0x00U)

  The value default of EIM data mask.

EIM Driver API

- void EIM_DRV_Init (uint32_t instance, uint8_t channelCnt, const eim_user_channel_config_t ∗channel↩
ConfigArr)

    Initializes the EIM module.

- void EIM_DRV_Deinit (uint32_t instance)

    De-initializes the EIM module.

- void EIM_DRV_ConfigChannel (uint32_t instance, const eim_user_channel_config_t ∗userChannelConfig)

    Configures the EIM channel.

- void EIM_DRV_GetChannelConfig (uint32_t instance, uint8_t channel, eim_user_channel_config_↩
t ∗channelConfig)

    Gets the EIM channel configuration.

- void EIM_DRV_GetDefaultConfig (uint8_t channel, eim_user_channel_config_t ∗channelConfig)

    Gets the EIM channel configuration default.

16.18.2 Data Structure Documentation

16.18.2.1 struct eim_user_channel_config_t

EIM channel configuration structure.

This structure holds the configuration settings for the EIM channel Implements : eim_user_channel_config_t_Class

Definition at line 58 of file eim_driver.h.

Data Fields

- uint8_t channel
- uint8_t checkBitMask
- uint32_t dataMask
- bool enable

Field Documentation

16.18.2.1.1 uint8_t channel

EIM channel number

Definition at line 60 of file eim_driver.h.

16.18.2.1.2 uint8_t checkBitMask

Specifies whether the corresponding bit of the check-bit bus from the target RAM should be inverted or remain unmodified

Definition at line 61 of file eim_driver.h.

16.18.2.1.3 uint32_t dataMask

Specifies whether the corresponding bit of the read data bus from the target RAM should be inverted or remain unmodified

Definition at line 63 of file eim_driver.h.

16.18.2.1.4 bool enable

true : EIM channel operation is enabled false : EIM channel operation is disabled

Definition at line 65 of file eim_driver.h.

16.18.3 Macro Definition Documentation

16.18.3.1 #define EIM_CHECKBITMASK_DEFAULT (0x01U)

The value default of EIM check-bit mask.

Definition at line 48 of file eim_driver.h.

16.18.3.2 #define EIM_DATAMASK_DEFAULT (0x00U)

The value default of EIM data mask.

Definition at line 50 of file eim_driver.h.

16.18.4 Function Documentation

16.18.4.1 void EIM_DRV_ConfigChannel (uint32_t *instance,* const eim_user_channel_config_t ∗ *userChannelConfig*)

Configures the EIM channel.

This function configures check-bit mask, data mask and operation status(enable/disable) for EIM channel. The EIM channel configuration structure shall be passed as arguments.

This is an example demonstrating how to define a EIM channel configuration structure:

```
1 eim_user_channel_config_t eimTestInit = {
2     .channel = 0x1U,
3     .checkBitMask = 0x25U,
4     .dataMask = 0x11101100U,
5     .enable = true
6 };
```

Parameters

in	*instance*	EIM module instance number
in	*userChannel↩ Config*	Pointer to EIM channel configuration structure

Definition at line 118 of file eim_driver.c.

16.18.4.2 void EIM_DRV_Deinit (uint32_t *instance*)

De-initializes the EIM module.

This function sets all registers to reset value and disables EIM module. In order to use the EIM module again, EIM_DRV_Init must be called.

Parameters

in	*instance*	EIM module instance number

Definition at line 95 of file eim_driver.c.

16.18.4.3 void EIM_DRV_GetChannelConfig (uint32_t *instance,* uint8_t *channel,* eim_user_channel_config_t ∗ *channelConfig*)

Gets the EIM channel configuration.

This function gets check bit mask, data mask and operation status of EIM channel.

Parameters

in	*instance*	EIM module instance number
in	*channel*	EIM channel number
out	*channelConfig*	Pointer to EIM channel configuration structure

Definition at line 144 of file eim_driver.c.

16.18.4.4 void EIM_DRV_GetDefaultConfig (uint8_t *channel,* eim_user_channel_config_t ∗ *channelConfig*)

Gets the EIM channel configuration default.

This function gets check bit mask, data mask and operation status default of EIM channel.

Parameters

in	*channel*	EIM channel number
out	*channelConfig*	Pointer to EIM channel configuration structure default

Definition at line 171 of file eim_driver.c.

16.18.4.5 void EIM_DRV_Init (uint32_t *instance,* uint8_t *channelCnt,* const eim_user_channel_config_t ∗
        ***channelConfigArr*)**

Initializes the EIM module.

This function configures for EIM channels. The EIM channel configuration structure array and number of configured channels shall be passed as arguments. This function should be called before calling any other EIM driver function.

This is an example demonstrating how to define a EIM channel configuration structure array:

```
1 eim_user_channel_config_t channelConfigArr[] =
2 {
3 {
4 .channel = 0x0U,
5 .checkBitMask = 0x12U,
6 .dataMask = 0x01234567U,
7 .enable = true
8 },
9 {
10 .channel = 0x1U,
11 .checkBitMask = 0x22U,
12 .dataMask = 0x01234444U,
13 .enable = false
14 }
15 };
```

Parameters

in	*instance*	EIM module instance number.
in	*channelCnt*	Number of configured channels
in	*channelConfig↩* *Arr*	EIM channel configuration structure array

Definition at line 65 of file eim_driver.c.

16.19   ERM Driver

16.19.1   Detailed Description

Error Reporting Module Peripheral Driver.

This section describes the programming interface of the ERM driver.

Data Structures

- struct erm_interrupt_config_t

    ERM interrupt notification configuration structure Implements : erm_interrupt_config_t_Class. *More...*
- struct erm_user_config_t

    ERM user configuration structure Implements : erm_user_config_t_Class. *More...*

Enumerations

- enum erm_ecc_event_t { ERM_EVENT_NONE = 0U, ERM_EVENT_SINGLE_BIT = 1U, ERM_EVENT_N↩
  ON_CORRECTABLE = 2U }

    ERM types of ECC events Implements : erm_ecc_event_t_Class.

ERM DRIVER API

- void ERM_DRV_Init (uint32_t instance, uint8_t channelCnt, const erm_user_config_t ∗userConfigArr)

    Initializes the ERM module.
- void ERM_DRV_Deinit (uint32_t instance)

    Sets the default configuration.
- void ERM_DRV_SetInterruptConfig (uint32_t instance, uint8_t channel, erm_interrupt_config_t interruptCfg)

    Sets interrupt notification.
- void ERM_DRV_GetInterruptConfig (uint32_t instance, uint8_t channel, erm_interrupt_config_t ∗const
  interruptPtr)

    Gets interrupt notification.
- void ERM_DRV_ClearEvent (uint32_t instance, uint8_t channel, erm_ecc_event_t eccEvent)

    Clears error event and the corresponding interrupt notification.
- erm_ecc_event_t ERM_DRV_GetErrorDetail (uint32_t instance, uint8_t channel, uint32_t ∗addressPtr)

    Gets the address of the last ECC event in Memory n and ECC event.

16.19.2   Data Structure Documentation

16.19.2.1   struct erm_interrupt_config_t

ERM interrupt notification configuration structure Implements : erm_interrupt_config_t_Class.

Definition at line 56 of file erm_driver.h.

Data Fields

- bool enableSingleCorrection
- bool enableNonCorrectable

Field Documentation

16.19.2.1.1   bool enableNonCorrectable

Enable Non-Correctable Interrupt Notification

Definition at line 59 of file erm_driver.h.

16.19.2.1.2   bool enableSingleCorrection

Enable Single Correction Interrupt Notification

Definition at line 58 of file erm_driver.h.

16.19.2.2   struct erm_user_config_t

ERM user configuration structure Implements : erm_user_config_t_Class.

Definition at line 66 of file erm_driver.h.

Data Fields

- uint8_t channel
- const erm_interrupt_config_t ∗ interruptCfg

Field Documentation

16.19.2.2.1   uint8_t channel

The channel assignments

Definition at line 68 of file erm_driver.h.

16.19.2.2.2   const erm_interrupt_config_t∗ interruptCfg

Interrupt configuration

Definition at line 69 of file erm_driver.h.

16.19.3   Enumeration Type Documentation

16.19.3.1   enum erm_ecc_event_t

ERM types of ECC events Implements : erm_ecc_event_t_Class.

Enumerator

    ERM_EVENT_NONE   None events

    ERM_EVENT_SINGLE_BIT   Single-bit correction ECC events

    ERM_EVENT_NON_CORRECTABLE   Non-correctable ECC events

Definition at line 45 of file erm_driver.h.

16.19.4   Function Documentation

16.19.4.1   void ERM_DRV_ClearEvent (uint32_t *instance,* uint8_t *channel,* erm_ecc_event_t *eccEvent*)

Clears error event and the corresponding interrupt notification.

This function clears the record of an event. If the corresponding interrupt is enabled, the interrupt notification will be cleared

Parameters

in	*instance*	The ERM instance number

| in | *channel* | The configured memory channel |
| in | *eccEvent* | The types of ECC events |

Definition at line 145 of file erm_driver.c.

16.19.4.2   void ERM_DRV_Deinit (uint32_t *instance*)

Sets the default configuration.

This function sets the default configuration

Parameters

| in | *instance* | The ERM instance number |

Definition at line 85 of file erm_driver.c.

16.19.4.3   erm_ecc_event_t ERM_DRV_GetErrorDetail (uint32_t *instance,* uint8_t *channel,* uint32_t ∗ *addressPtr*)

Gets the address of the last ECC event in Memory n and ECC event.

This function gets the address of the last ECC event in Memory n and the types of the event

Parameters

in	*instance*	The ERM instance number
in	*channel*	The examined memory channel
out	*addressPtr*	The pointer to address of the last ECC event in Memory n with ECC event

Returns

 The last occurred ECC event

Definition at line 177 of file erm_driver.c.

16.19.4.4   void ERM_DRV_GetInterruptConfig (uint32_t *instance,* uint8_t *channel,* erm_interrupt_config_t ∗const *interruptPtr*)

Gets interrupt notification.

This function gets the current interrupt configuration of the available events (which interrupts are enabled/disabled)

Parameters

in	*instance*	The ERM instance number
in	*channel*	The examined memory channel
out	*interruptPtr*	The pointer to the ERM interrupt configuration structure

Definition at line 123 of file erm_driver.c.

16.19.4.5   void ERM_DRV_Init (uint32_t *instance,* uint8_t *channelCnt,* const erm_user_config_t ∗ *userConfigArr*)

Initializes the ERM module.

This function initializes ERM driver based on user configuration input, channelCnt takes values between 1 and the maximum channel count supported by the hardware

Parameters

| in | *instance* | The ERM instance number |
| in | *channelCnt* | The number of channels |

in	*userConfigArr*	The pointer to the array of ERM user configure structure

Definition at line 57 of file erm_driver.c.

16.19.4.6   void ERM_DRV_SetInterruptConfig (uint32_t *instance,* uint8_t *channel,* erm_interrupt_config_t *interruptCfg*)

Sets interrupt notification.

This function sets interrupt notification based on interrupt notification configuration input

Parameters

in	*instance*	The ERM instance number
in	*channel*	The configured memory channel
in	*interruptCfg*	The ERM interrupt configuration structure

Definition at line 102 of file erm_driver.c.

16.20   Enhanced Direct Memory Access (eDMA)

16.20.1   Detailed Description

The S32 SDK provides Peripheral Driver for the Enhanced Direct Memory Access (eDMA) module.
The direct memory access engine features are used for performing complex data transfers with minimal intervention from the host processor. These sections describe the S32 SDK software modules API that can be used for initializing, configuring and triggering eDMA transfers.

Modules

- EDMA Driver

    This module covers the functionality of the Enhanced Direct Memory Access (eDMA) peripheral driver.

16.21 Error Injection Module (EIM)

16.21.1 Detailed Description

The S32 SDK provides Peripheral Drivers for the Error Injection Module (EIM) of S32 MCU.

The Error Injection Module is mainly used for diagnostic purposes. It provides a method for diagnostic coverage of the peripheral memories.
The Error Injection Module (EIM) provides support for inducing single-bit and multi-bit inversions on read data when accessing peripheral RAMs. Injecting faults on memory accesses can be used to exercise the SEC-DED ECC function of the related system.
Each EIM channel n corresponds to a source of potential memory error events. The following table shows the channel assignments. | Memory n event source EIM channel n |----------------------------------—

| S32K14x | S32K11x

0 | SRAM_L | SRAM_U

1 | SRAM_U | Reserved

Important Note:

1. Make sure that STACK memory is located in RAM different than where EIM will inject a non-correctable error.

2. For single bit error generation, flip only one bit out of DATA_MASK or CHKBIT_MASK bit-fields in EIM control registers.

3. For Double bit error generation, flip only two bits out of DATA_MASK or CHKBIT_MASK bit-fields in EIM control registers.

4. If more than 2 bits are flipped that there is no guarantee in design that what type of error get generated.

Modules

- EIM Driver

  Error Injection Module Peripheral Driver.
  EIM PD provides a set of high-level APIs/services to configure the Error Injection Module (EIM) module.

16.22   Error Reporting Module (ERM)

16.22.1   Detailed Description

The S32 SDK provides Peripheral Drivers for the Error Reporting Module (ERM) module of S32 SDK devices.

The Error Reporting Module (ERM) provides information and optional interrupt notification on memory errors events associated with ECC (Error Correction Code).
The ERM includes these features:
Capture address information on single-bit correction and non-correctable ECC events.
Optional interrupt notification on captured ECC events.
Support for ECC event capturing for memory sources, with individual reporting fields and interrupt configuration per memory channel.
Each ERM channel n corresponds to a source of potential memory error events. The following table shows the channel assignments. | Memory n event source ERM channel n |-----------------------------------—

| S32K14x | S32K11x

0 | SRAM_L | SRAM_U

1 | SRAM_U | Reserved

16.22.2   ERM Driver Initialization

In order to be able to use the error reporting in your application, the first thing to do is initializing it with user configuration input. This is done by calling the **ERM_DRV_Init** function. Note that: channelCnt takes values between 1 and the maximum channel count supported by the hardware.

16.22.3   ERM Driver Operation

After ERM initialization, the ERM_DRV_SetInterruptConfig() shall be used to set interrupt notification based on interrupt notification configuration.
The ERM_DRV_GetInterruptConfig() shall be used to get the current interrupt configuration of the available events (which interrupts are enabled/disabled).

The ERM_DRV_GetErrorDetail() shall be used to get the address of the last ECC event in Memory n and ECC event.

The ERM_DRV_ClearEvent() shall be used to clear both the record of an event and the corresponding interrupt notification.

This is example code to configure the ERM driver:

```
/* Device instance number */
#define INST_ERM1 (0U)

/* The number of configured channel(s) */
#define ERM_NUM_OF_CFG_CHANNEL (2U)

/* Interrupt configuration 0 */
const erm_interrupt_config_t erm1_Interrupt1 =
{
    .enableSingleCorrection = false,
    .enableNonCorrectable   = true,
};

/* Interrupt configuration 1 */
const erm_interrupt_config_t erm1_Interrupt3 =
{
    .enableSingleCorrection = true,
    .enableNonCorrectable   = true,
};


/* User configuration */
const erm_user_config_t erm1_InitConfig[] =
{
```

```
    /* Channel 0U */
    {
        .channel      = 0U,
        .interruptCfg = &erm1_Interrupt1,
    },

    /* Channel 1U */
    {
        .channel      = 1U,
        .interruptCfg = &erm1_Interrupt3,
    }
};

int main()
{
    /* Initializes the ERM module */
    ERM_DRV_Init(INST_ERM1, ERM_NUM_OF_CFG_CHANNEL, erm1_InitConfig);
    ...
    /* De-Initializes the ERM module */
    ERM_DRV_Deinit(INST_ERM1);
    ...
    return 0;
}

/* Interrupt handler */
/* Interrupt handler for single bit */
void ERM_single_fault_IRQHandler()
{
    /* Clears the event for channel 1 */
    ERM_DRV_ClearEvent(INST_ERM1, 1U, ERM_EVENT_SINGLE_BIT);
    ...
}

/* Interrupt handler for non correctable */
void ERM_double_fault_IRQHandler()
{
    /* Clears the event for channel 0 */
    ERM_DRV_ClearEvent(INST_ERM1, 0U,
      ERM_EVENT_NON_CORRECTABLE);
    /* Clears the event for channel 1 */
    ERM_DRV_ClearEvent(INST_ERM1, 1U,
      ERM_EVENT_NON_CORRECTABLE);
    ...
}
```

Modules

- ERM Driver

  Error Reporting Module Peripheral Driver.

16.23   Flash Memory (Flash)

16.23.1   Detailed Description

This section describes the programming interface of the Flash Peripheral Driver.

Data Structures

- struct flash_user_config_t
    Flash User Configuration Structure. More...
- struct flash_ssd_config_t
    Flash SSD Configuration Structure. More...
- struct flash_eeprom_status_t
    EEPROM status structure. More...

Macros

- #define CLEAR_FTFx_FSTAT_ERROR_BITS FTFx_FSTAT = (uint8_t)(FTFx_FSTAT_FPVIOL_MASK | F↩ TFx_FSTAT_ACCERR_MASK | FTFx_FSTAT_RDCOLERR_MASK)
- #define FTFx_WORD_SIZE 0x0002U
- #define FTFx_LONGWORD_SIZE 0x0004U
- #define FTFx_PHRASE_SIZE 0x0008U
- #define FTFx_DPHRASE_SIZE 0x0010U
- #define FTFx_RSRC_CODE_REG FTFx_FCCOB8
- #define FTFx_VERIFY_BLOCK 0x00U
- #define FTFx_VERIFY_SECTION 0x01U
- #define FTFx_PROGRAM_CHECK 0x02U
- #define FTFx_READ_RESOURCE 0x03U
- #define FTFx_PROGRAM_LONGWORD 0x06U
- #define FTFx_PROGRAM_PHRASE 0x07U
- #define FTFx_ERASE_BLOCK 0x08U
- #define FTFx_ERASE_SECTOR 0x09U
- #define FTFx_PROGRAM_SECTION 0x0BU
- #define FTFx_VERIFY_ALL_BLOCK 0x40U
- #define FTFx_READ_ONCE 0x41U
- #define FTFx_PROGRAM_ONCE 0x43U
- #define FTFx_ERASE_ALL_BLOCK 0x44U
- #define FTFx_SECURITY_BY_PASS 0x45U
- #define FTFx_PFLASH_SWAP 0x46U
- #define FTFx_ERASE_ALL_BLOCK_UNSECURE 0x49U
- #define FTFx_PROGRAM_PARTITION 0x80U
- #define FTFx_SET_EERAM 0x81U
- #define RESUME_WAIT_CNT 0x20U
    Resume wait count used in FLASH_DRV_EraseResume function.
- #define SUSPEND_WAIT_CNT 0x40U
    Suspend wait count used in FLASH_DRV_EraseSuspend function.
- #define DFLASH_IFR_READRESOURCE_ADDRESS 0x8000FCU
- #define GET_BIT_0_7(value) ((uint8_t)(((uint32_t)(value)) & 0xFFU))
- #define GET_BIT_8_15(value) ((uint8_t)((((uint32_t)(value)) >> 8) & 0xFFU))
- #define GET_BIT_16_23(value) ((uint8_t)((((uint32_t)(value)) >> 16) & 0xFFU))
- #define GET_BIT_24_31(value) ((uint8_t)(((uint32_t)(value)) >> 24))
- #define FLASH_SECURITY_STATE_KEYEN 0x80U
- #define FLASH_SECURITY_STATE_UNSECURED 0x02U
- #define CSE_KEY_SIZE_CODE_MAX 0x03U
- #define FLASH_CALLBACK_CS 0x0AU
    Callback period count for FlashCheckSum.

Typedefs

- typedef void(∗ flash_callback_t) (void)

    Call back function pointer data type.

Enumerations

- enum flash_flexRam_function_control_code_t {
  EEE_ENABLE = 0x00U, EEE_QUICK_WRITE = 0x55U, EEE_STATUS_QUERY = 0x77U, EEE_COMPL↩
  ETE_INTERRUPT_QUICK_WRITE = 0xAAU,
  EEE_DISABLE = 0xFFU }

    FlexRAM Function control Code.

Variables

- uint32_t PFlashBase
- uint32_t PFlashSize
- uint32_t DFlashBase
- uint32_t EERAMBase
- flash_callback_t CallBack
- uint32_t PFlashBase
- uint32_t PFlashSize
- uint32_t DFlashBase
- uint32_t DFlashSize
- uint32_t EERAMBase
- uint32_t EEESize
- flash_callback_t CallBack
- uint8_t brownOutCode
- uint16_t numOfRecordReqMaintain
- uint16_t sectorEraseCount

PFlash swap control codes

- #define FTFx_SWAP_SET_INDICATOR_ADDR 0x01U

    Initialize Swap System control code.
- #define FTFx_SWAP_SET_IN_PREPARE 0x02U

    Set Swap in Update State.
- #define FTFx_SWAP_SET_IN_COMPLETE 0x04U

    Set Swap in Complete State.
- #define FTFx_SWAP_REPORT_STATUS 0x08U

    Report Swap Status.

PFlash swap states

- #define FTFx_SWAP_UNINIT 0x00U

    Uninitialized swap mode.
- #define FTFx_SWAP_READY 0x01U

    Ready swap mode.
- #define FTFx_SWAP_UPDATE 0x02U

    Update swap mode.
- #define FTFx_SWAP_UPDATE_ERASED 0x03U

    Update-Erased swap mode.
- #define FTFx_SWAP_COMPLETE 0x04U

    Complete swap mode.

Flash security status

- #define FLASH_NOT_SECURE 0x01U

    Flash currently not in secure state.
- #define FLASH_SECURE_BACKDOOR_ENABLED 0x02U

    Flash is secured and backdoor key access enabled.
- #define FLASH_SECURE_BACKDOOR_DISABLED 0x04U

    Flash is secured and backdoor key access disabled.

Null Callback function definition

- #define NULL_CALLBACK ((flash_callback_t)0xFFFFFFFFU)

    Null callback.

Flash driver APIs

- status_t FLASH_DRV_Init (const flash_user_config_t ∗const pUserConf, flash_ssd_config_t ∗const pSSD↩
Config)

    Initializes Flash.
- void FLASH_DRV_GetPFlashProtection (uint32_t ∗protectStatus)

    P-Flash get protection.
- status_t FLASH_DRV_SetPFlashProtection (uint32_t protectStatus)

    P-Flash set protection.
- void FLASH_DRV_GetSecurityState (uint8_t ∗securityState)

    Flash get security state.
- status_t FLASH_DRV_SecurityBypass (const flash_ssd_config_t ∗pSSDConfig, const uint8_t ∗keyBuffer)

    Flash security bypass.
- status_t FLASH_DRV_EraseAllBlock (const flash_ssd_config_t ∗pSSDConfig)

    Flash erase all blocks.
- status_t FLASH_DRV_VerifyAllBlock (const flash_ssd_config_t ∗pSSDConfig, uint8_t marginLevel)

    Flash verify all blocks.
- status_t FLASH_DRV_EraseSector (const flash_ssd_config_t ∗pSSDConfig, uint32_t dest, uint32_t size)

    Flash erase sector.
- status_t FLASH_DRV_VerifySection (const flash_ssd_config_t ∗pSSDConfig, uint32_t dest, uint16_t number,
uint8_t marginLevel)

    Flash verify section.
- void FLASH_DRV_EraseSuspend (void)

    Flash erase suspend.
- void FLASH_DRV_EraseResume (void)

    Flash erase resume.
- status_t FLASH_DRV_ReadOnce (const flash_ssd_config_t ∗pSSDConfig, uint8_t recordIndex, uint8_t ∗p↩
DataArray)

    Flash read once.
- status_t FLASH_DRV_ProgramOnce (const flash_ssd_config_t ∗pSSDConfig, uint8_t recordIndex, const
uint8_t ∗pDataArray)

    Flash program once.
- status_t FLASH_DRV_Program (const flash_ssd_config_t ∗pSSDConfig, uint32_t dest, uint32_t size, const
uint8_t ∗pData)

    Flash program.
- status_t FLASH_DRV_ProgramCheck (const flash_ssd_config_t ∗pSSDConfig, uint32_t dest, uint32_t size,
const uint8_t ∗pExpectedData, uint32_t ∗pFailAddr, uint8_t marginLevel)

Flash program check.

- status_t FLASH_DRV_CheckSum (const flash_ssd_config_t ∗pSSDConfig, uint32_t dest, uint32_t size, uint32_t ∗pSum)

    Calculates check sum.
- status_t FLASH_DRV_EnableCmdCompleteInterupt (void)

    Enable the command complete interrupt.
- void FLASH_DRV_DisableCmdCompleteInterupt (void)

    Disable the command complete interrupt.
- static bool FLASH_DRV_GetCmdCompleteFlag (void)

    Check the command complete flag has completed or not.
- status_t FLASH_DRV_EnableReadColisionInterupt (void)

    Enable the read collision error interrupt.
- void FLASH_DRV_DisableReadColisionInterupt (void)

    Disable the read collision error interrupt.
- static bool FLASH_DRV_GetReadColisionFlag (void)

    Check the read collision error flag is detected or not.
- static void FLASH_DRV_ClearReadColisionFlag (void)

    Clear the read collision error flag.

16.23.2   Data Structure Documentation

16.23.2.1   struct flash_user_config_t

Flash User Configuration Structure.

Implements : flash_user_config_t_Class

Definition at line 557 of file flash_driver.h.

Data Fields

- uint32_t PFlashBase
- uint32_t PFlashSize
- uint32_t DFlashBase
- uint32_t EERAMBase
- flash_callback_t CallBack

16.23.2.2   struct flash_ssd_config_t

Flash SSD Configuration Structure.

The structure includes the static parameters for C90TFS/FTFx which are device-dependent. The fields including PFlashBlockBase, PFlashBlockSize, DFlashBlockBase, EERAMBlockBase, and CallBack are passed via flash_↩ user_config_t. The rest of parameters such as DFlashBlockSize, and EEEBlockSize will be initialized in FLASH_↩ DRV_Init() automatically.

Implements : flash_ssd_config_t_Class

Definition at line 581 of file flash_driver.h.

Data Fields

- uint32_t PFlashBase
- uint32_t PFlashSize
- uint32_t DFlashBase
- uint32_t DFlashSize
- uint32_t EERAMBase
- uint32_t EEESize
- flash_callback_t CallBack

16.23.2.3 struct flash_eeprom_status_t

EEPROM status structure.

Implements : flash_eeprom_status_t_Class

Definition at line 603 of file flash_driver.h.

Data Fields

- uint8_t brownOutCode
- uint16_t numOfRecordReqMaintain
- uint16_t sectorEraseCount

16.23.3 Macro Definition Documentation

**16.23.3.1 #define CLEAR_FTFx_FSTAT_ERROR_BITS FTFx_FSTAT = (uint8_t)(FTFx_FSTAT_FPVIOL_MASK |
FTFx_FSTAT_ACCERR_MASK | FTFx_FSTAT_RDCOLERR_MASK)**

Definition at line 380 of file flash_driver.h.

16.23.3.2 #define CSE_KEY_SIZE_CODE_MAX 0x03U

Definition at line 472 of file flash_driver.h.

16.23.3.3 #define DFLASH_IFR_READRESOURCE_ADDRESS 0x8000FCU

Definition at line 459 of file flash_driver.h.

16.23.3.4 #define FLASH_CALLBACK_CS 0x0AU

Callback period count for FlashCheckSum.

This value is only relevant for FlashCheckSum operation, where a high rate of calling back can impair performance. The rest of the flash operations invoke the callback as often as possible while waiting for the flash controller to finish the requested operation.

Definition at line 515 of file flash_driver.h.

16.23.3.5 #define FLASH_NOT_SECURE 0x01U

Flash currently not in secure state.

Definition at line 498 of file flash_driver.h.

16.23.3.6 #define FLASH_SECURE_BACKDOOR_DISABLED 0x04U

Flash is secured and backdoor key access disabled.

Definition at line 502 of file flash_driver.h.

16.23.3.7 #define FLASH_SECURE_BACKDOOR_ENABLED 0x02U

Flash is secured and backdoor key access enabled.

Definition at line 500 of file flash_driver.h.

16.23.3.8 #define FLASH_SECURITY_STATE_KEYEN 0x80U

Definition at line 468 of file flash_driver.h.

16.23.3.9 #define FLASH_SECURITY_STATE_UNSECURED 0x02U

Definition at line 469 of file flash_driver.h.

16.23.3.10 #define FTFx_DPHRASE_SIZE 0x0010U

Definition at line 389 of file flash_driver.h.

16.23.3.11 #define FTFx_ERASE_ALL_BLOCK 0x44U

Definition at line 413 of file flash_driver.h.

16.23.3.12 #define FTFx_ERASE_ALL_BLOCK_UNSECURE 0x49U

Definition at line 416 of file flash_driver.h.

16.23.3.13 #define FTFx_ERASE_BLOCK 0x08U

Definition at line 407 of file flash_driver.h.

16.23.3.14 #define FTFx_ERASE_SECTOR 0x09U

Definition at line 408 of file flash_driver.h.

16.23.3.15 #define FTFx_LONGWORD_SIZE 0x0004U

Definition at line 385 of file flash_driver.h.

16.23.3.16 #define FTFx_PFLASH_SWAP 0x46U

Definition at line 415 of file flash_driver.h.

16.23.3.17 #define FTFx_PHRASE_SIZE 0x0008U

Definition at line 387 of file flash_driver.h.

16.23.3.18 #define FTFx_PROGRAM_CHECK 0x02U

Definition at line 403 of file flash_driver.h.

16.23.3.19 #define FTFx_PROGRAM_LONGWORD 0x06U

Definition at line 405 of file flash_driver.h.

16.23.3.20 #define FTFx_PROGRAM_ONCE 0x43U

Definition at line 412 of file flash_driver.h.

16.23.3.21 #define FTFx_PROGRAM_PARTITION 0x80U

Definition at line 417 of file flash_driver.h.

16.23.3.22 #define FTFx_PROGRAM_PHRASE 0x07U

Definition at line 406 of file flash_driver.h.

16.23.3.23 #define FTFx_PROGRAM_SECTION 0x0BU

Definition at line 409 of file flash_driver.h.

16.23.3.24 #define FTFx_READ_ONCE 0x41U

Definition at line 411 of file flash_driver.h.

16.23.3.25   #define FTFx_READ_RESOURCE 0x03U

Definition at line 404 of file flash_driver.h.

16.23.3.26   #define FTFx_RSRC_CODE_REG FTFx_FCCOB8

Definition at line 395 of file flash_driver.h.

16.23.3.27   #define FTFx_SECURITY_BY_PASS 0x45U

Definition at line 414 of file flash_driver.h.

16.23.3.28   #define FTFx_SET_EERAM 0x81U

Definition at line 418 of file flash_driver.h.

16.23.3.29   #define FTFx_SWAP_COMPLETE 0x04U

Complete swap mode.

Definition at line 448 of file flash_driver.h.

16.23.3.30   #define FTFx_SWAP_READY 0x01U

Ready swap mode.

Definition at line 442 of file flash_driver.h.

16.23.3.31   #define FTFx_SWAP_REPORT_STATUS 0x08U

Report Swap Status.

Definition at line 432 of file flash_driver.h.

16.23.3.32   #define FTFx_SWAP_SET_IN_COMPLETE 0x04U

Set Swap in Complete State.

Definition at line 430 of file flash_driver.h.

16.23.3.33   #define FTFx_SWAP_SET_IN_PREPARE 0x02U

Set Swap in Update State.

Definition at line 428 of file flash_driver.h.

16.23.3.34   #define FTFx_SWAP_SET_INDICATOR_ADDR 0x01U

Initialize Swap System control code.

Definition at line 426 of file flash_driver.h.

16.23.3.35   #define FTFx_SWAP_UNINIT 0x00U

Uninitialized swap mode.

Definition at line 440 of file flash_driver.h.

16.23.3.36   #define FTFx_SWAP_UPDATE 0x02U

Update swap mode.

Definition at line 444 of file flash_driver.h.

16.23.3.37   #define FTFx_SWAP_UPDATE_ERASED 0x03U

Update-Erased swap mode.

Definition at line 446 of file flash_driver.h.

16.23.3.38   #define FTFx_VERIFY_ALL_BLOCK 0x40U

Definition at line 410 of file flash_driver.h.

16.23.3.39   #define FTFx_VERIFY_BLOCK 0x00U

Definition at line 401 of file flash_driver.h.

16.23.3.40   #define FTFx_VERIFY_SECTION 0x01U

Definition at line 402 of file flash_driver.h.

16.23.3.41   #define FTFx_WORD_SIZE 0x0002U

Definition at line 383 of file flash_driver.h.

16.23.3.42   #define GET_BIT_0_7(*value*) ((uint8_t)(((uint32_t)(value)) & 0xFFU))

Definition at line 462 of file flash_driver.h.

16.23.3.43   #define GET_BIT_16_23(*value*) ((uint8_t)((((uint32_t)(value)) $>>$ 16) & 0xFFU))

Definition at line 464 of file flash_driver.h.

16.23.3.44   #define GET_BIT_24_31(*value*) ((uint8_t)(((uint32_t)(value)) $>>$ 24))

Definition at line 465 of file flash_driver.h.

16.23.3.45   #define GET_BIT_8_15(*value*) ((uint8_t)((((uint32_t)(value)) $>>$ 8) & 0xFFU))

Definition at line 463 of file flash_driver.h.

16.23.3.46   #define NULL_CALLBACK ((flash_callback_t)0xFFFFFFFFU)

Null callback.

Definition at line 526 of file flash_driver.h.

16.23.3.47   #define RESUME_WAIT_CNT 0x20U

Resume wait count used in FLASH_DRV_EraseResume function.

Definition at line 452 of file flash_driver.h.

16.23.3.48   #define SUSPEND_WAIT_CNT 0x40U

Suspend wait count used in FLASH_DRV_EraseSuspend function.

Definition at line 454 of file flash_driver.h.

16.23.4   Typedef Documentation

16.23.4.1   typedef void($*$ flash_callback_t) (void)

Call back function pointer data type.

If using callback in the application, any code reachable from this function must not be placed in a Flash block

targeted for a program/erase operation. Functions can be placed in RAM section by using the START/END_FUN↩
CTION_DEFINITION/DECLARATION_RAMSECTION macros.

Definition at line 543 of file flash_driver.h.

16.23.5 Enumeration Type Documentation

16.23.5.1 enum flash_flexRam_function_control_code_t

FlexRAM Function control Code.

Implements : flash_flexRAM_function_control_code_t_Class

Enumerator

>   ***EEE_ENABLE*** Make FlexRAM available for emulated EEPROM
>
>   ***EEE_QUICK_WRITE*** Make FlexRAM available for EEPROM quick writes
>
>   ***EEE_STATUS_QUERY*** EEPROM quick write status query
>
>   ***EEE_COMPLETE_INTERRUPT_QUICK_WRITE*** Complete interrupted EEPROM quick write process
>
>   ***EEE_DISABLE*** Make FlexRAM available as RAM

Definition at line 484 of file flash_driver.h.

16.23.6 Function Documentation

16.23.6.1 status_t FLASH_DRV_CheckSum (const flash_ssd_config_t ∗ pSSDConfig, uint32_t dest, uint32_t size, uint32_t ∗ pSum)

Calculates check sum.

This API performs 32 bit sum of each byte data over a specified Flash memory range without carry which provides rapid method for checking data integrity. The callback time period of this API is determined via FLASH_CALLBA↩
CK_CS macro in flash_driver.h which is used as a counter value for the CallBack() function calling in this API. This value can be changed as per the user requirement. User can change this value to obtain the maximum permissible callback time period. This API always returns STATUS_SUCCESS if size provided by user is zero regardless of the input validation.

Parameters

in	pSSDConfig	The SSD configuration structure pointer.
in	dest	Start address of the Flash range to be summed.
in	size	Size in byte of the Flash range to be summed.
in	pSum	To return the sum value.

Returns

>   operation status
>
>   • STATUS_SUCCESS: Operation was successful.
>
>   • STATUS_ERROR: Operation failure was occurred.

16.23.6.2 static void FLASH_DRV_ClearReadColisionFlag (void) `[inline],[static]`

Clear the read collision error flag.

Implements : FLASH_DRV_ClearReadColisionFlag_Activity

Definition at line 1411 of file flash_driver.h.

16.23.6.3   void FLASH_DRV_DisableCmdCompleteInterupt (void)

Disable the command complete interrupt.

16.23.6.4   void FLASH_DRV_DisableReadColisionInterupt (void)

Disable the read collision error interrupt.

16.23.6.5   status_t FLASH_DRV_EnableCmdCompleteInterupt (void)

Enable the command complete interrupt.

This function will enable the command complete interrupt is generated when an FTFC command completes.

Returns

> operation status
>
> > • STATUS_SUCCESS: Operation was successful.

16.23.6.6   status_t FLASH_DRV_EnableReadColisionInterupt (void)

Enable the read collision error interrupt.

This function will enable the read collision error interrupt generation when an FTFC read collision error occurs.

Returns

> operation status
>
> > • STATUS_SUCCESS: Operation was successful.

16.23.6.7   status_t FLASH_DRV_EraseAllBlock (const flash_ssd_config_t ∗ pSSDConfig)

Flash erase all blocks.

This API erases all Flash memory, initializes the FlexRAM, verifies all memory contents, and then releases the MCU security.

Parameters

in	*pSSDConfig*	The SSD configuration structure pointer.

Returns

> operation status
>
> > • STATUS_SUCCESS: Operation was successful.
> > • STATUS_ERROR: Operation failure was occurred.
> > • STATUS_BUSY: Operation was busy.

16.23.6.8   void FLASH_DRV_EraseResume (void)

Flash erase resume.

This API is used to resume a previous suspended operation of Flash erase sector command This function must be located in RAM memory or different Flash blocks which are targeted for writing to avoid RWW error.

16.23.6.9   status_t FLASH_DRV_EraseSector (const flash_ssd_config_t ∗ pSSDConfig, uint32_t dest, uint32_t size)

Flash erase sector.

This API erases one or more sectors in P-Flash or D-Flash memory. This API always returns FTFx_OK if size provided by the user is zero regardless of the input validation.

Parameters

in	pSSDConfig	The SSD configuration structure pointer.
in	dest	Address in the first sector to be erased.
in	size	Size to be erased in bytes. It is used to determine number of sectors to be erased.

Returns

operation status

- STATUS_SUCCESS: Operation was successful.
- STATUS_ERROR: Operation failure was occurred.
- STATUS_UNSUPPORTED: Operation was unsupported.
- STATUS_BUSY: Operation was busy.

16.23.6.10   void FLASH_DRV_EraseSuspend (void)

Flash erase suspend.

This API is used to suspend a current operation of Flash erase sector command. This function must be located in RAM memory or different Flash blocks which are targeted for writing to avoid the RWW error.

16.23.6.11   static bool FLASH_DRV_GetCmdCompleteFlag (void) `[inline],[static]`

Check the command complete flag has completed or not.

Returns

the command complete flag

- true: The FTFC command has completed.
- false: The FTFC command is in progress.

Implements : FLASH_DRV_GetCmdCompleteFlag_Activity

Definition at line 1370 of file flash_driver.h.

16.23.6.12   void FLASH_DRV_GetPFlashProtection (uint32_t ∗ protectStatus)

P-Flash get protection.

This API retrieves the current P-Flash protection status. Considering the time consumption for getting protection is very low and even can be ignored. It is not necessary to utilize the Callback function to support the time-critical events.

Parameters

out	protectStatus	To return the current value of the P-Flash Protection. Each bit is corresponding to protection of 1/32 of the total P-Flash. The least significant bit is corresponding to the lowest address area of P-Flash. The most significant bit is corresponding to the highest address area of P-Flash and so on. There are two possible cases as below: • 0: this area is protected. • 1: this area is unprotected.

16.23.6.13   static bool FLASH_DRV_GetReadColisionFlag (void) `[inline],[static]`

Check the read collision error flag is detected or not.

Returns

> the read collision error flag
>
>> • true: Collision error detected.
>>
>> • false: No collision error detected.

Implements : FLASH_DRV_GetReadColisionFlag_Activity

Definition at line 1401 of file flash_driver.h.

16.23.6.14   void FLASH_DRV_GetSecurityState (uint8_t ∗ *securityState*)

Flash get security state.

This API retrieves the current Flash security status, including the security enabling state and the back door key enabling state.

Parameters

out	*securityState*	To return the current security status code. • FLASH_NOT_SECURE (0x01U): Flash currently not in secure state • FLASH_SECURE_BACKDOOR_ENABLED (0x02U): Flash is secured and back door key access enabled • FLASH_SECURE_BACKDOOR_DISABLED (0x04U): Flash is secured and back door key access disabled.

16.23.6.15   status_t FLASH_DRV_Init (const flash_user_config_t ∗const *pUserConf,* flash_ssd_config_t ∗const *pSSDConfig*)

Initializes Flash.

This API initializes Flash module by clearing status error bit and reporting the memory configuration via SS←
D configuration structure.

Parameters

in	*pUserConf*	The user configuration structure pointer.
in	*pSSDConfig*	The SSD configuration structure pointer.

Returns

> operation status
>
>> • STATUS_SUCCESS: Operation was successful.

16.23.6.16   status_t FLASH_DRV_Program (const flash_ssd_config_t ∗ *pSSDConfig,* uint32_t *dest,* uint32_t *size,* const uint8_t ∗ *pData*)

Flash program.

This API is used to program 4 consecutive bytes (for program long word command) and 8 consecutive bytes (for program phrase command) on P-Flash or D-Flash block. This API always returns FTFx_OK if size provided by user is zero regardless of the input validation

Parameters

in	pSSDConfig	The SSD configuration structure pointer.
in	dest	Start address for the intended program operation.
in	size	Size in byte to be programmed
in	pData	Pointer of source address from which data has to be taken for program operation.

Returns

>   operation status
>
>   • STATUS_SUCCESS: Operation was successful.
>
>   • STATUS_ERROR: Operation failure was occurred.
>
>   • STATUS_UNSUPPORTED: Operation was unsupported.
>
>   • STATUS_BUSY: Operation was busy.

16.23.6.17   status_t FLASH_DRV_ProgramCheck (const flash_ssd_config_t ∗ pSSDConfig, uint32_t dest, uint32_t size, const uint8_t ∗ pExpectedData, uint32_t ∗ pFailAddr, uint8_t marginLevel)

Flash program check.

This API tests a previously programmed P-Flash or D-Flash long word to see if it reads correctly at the specified margin level. This API always returns FTFx_OK if size provided by user is zero regardless of the input validation

Parameters

in	pSSDConfig	The SSD configuration structure pointer.
in	dest	Start address for the intended program check operation.
in	size	Size in byte to check accuracy of program operation
in	pExpectedData	The pointer to the expected data.
in	pFailAddr	Returned the first aligned failing address.
in	marginLevel	Read margin choice as follows: • marginLevel = 0x1U: read at User margin 1/0 level. • marginLevel = 0x2U: read at Factory margin 1/0 level.

Returns

>   operation status
>
>   • STATUS_SUCCESS: Operation was successful.
>
>   • STATUS_ERROR: Operation failure was occurred.
>
>   • STATUS_BUSY: Operation was busy.

16.23.6.18   status_t FLASH_DRV_ProgramOnce (const flash_ssd_config_t ∗ pSSDConfig, uint8_t recordIndex, const uint8_t ∗ pDataArray)

Flash program once.

This API is used to program to a reserved 64 byte field located in the P-Flash IFR via given number of record. See the corresponding reference manual to get correct value of this number.

Parameters

in	pSSDConfig	The SSD configuration structure pointer.
in	recordIndex	The record index will be read. It can be from 0x0U to 0x7U or from 0x0U to 0xF according to specific derivative.
in	pDataArray	Pointer to the array from which data will be taken for program once command.

Returns

operation status

- STATUS_SUCCESS: Operation was successful.

- STATUS_ERROR: Operation failure was occurred.

- STATUS_BUSY: Operation was busy.

16.23.6.19   status_t FLASH_DRV_ReadOnce (const flash_ssd_config_t ∗ pSSDConfig, uint8_t recordIndex, uint8_t ∗ pDataArray)

Flash read once.

This API is used to read out a reserved 64 byte field located in the P-Flash IFR via given number of record. See the corresponding reference manual to get the correct value of this number.

Parameters

in	pSSDConfig	The SSD configuration structure pointer.
in	recordIndex	The record index will be read. It can be from 0x0U to 0x7U or from 0x0U to 0xF according to specific derivative.
in	pDataArray	Pointer to the array to return the data read by the read once command.

Returns

operation status

- STATUS_SUCCESS: Operation was successful.

- STATUS_ERROR: Operation failure was occurred.

- STATUS_BUSY: Operation was busy.

16.23.6.20   status_t FLASH_DRV_SecurityBypass (const flash_ssd_config_t ∗ pSSDConfig, const uint8_t ∗ keyBuffer)

Flash security bypass.

This API un-secures the device by comparing the user's provided back door key with the ones in the Flash Configuration Field. If they are matched, the security is released. Otherwise, an error code is returned.

Parameters

in	pSSDConfig	The SSD configuration structure pointer.
in	keyBuffer	Point to the user buffer containing the back door key.

Returns

operation status

- STATUS_SUCCESS: Operation was successful.

- STATUS_ERROR: Operation failure was occurred.

- STATUS_BUSY: Operation was busy.

16.23.6.21   status_t FLASH_DRV_SetPFlashProtection (uint32_t protectStatus)

P-Flash set protection.

This API sets the P-Flash protection to the intended protection status. Setting P-Flash protection status is subject to a protection, transition restriction. If there is a setting violation, it returns an error code and the current protection status will not be changed.

Parameters

in	*protectStatus*	The expected protect status user wants to set to P-Flash protection register. Each bit is corresponding to protection of 1/32 of the total P-Flash. The least significant bit is corresponding to the lowest address area of P-Flash. The most significant bit is corresponding to the highest address area of P- Flash, and so on. There are two possible cases as shown below:
		• 0: this area is protected.
		• 1: this area is unprotected.

Returns

operation status

- STATUS_SUCCESS: Operation was successful.

- STATUS_ERROR: Operation failure was occurred.

16.23.6.22   status_t FLASH_DRV_VerifyAllBlock (const flash_ssd_config_t ∗ *pSSDConfig,* uint8_t *marginLevel*)

Flash verify all blocks.

This function checks to see if the P-Flash and/or D-Flash, EEPROM backup area, and D-Flash IFR have been erased to the specified read margin level, if applicable, and releases security if the readout passes.

Parameters

in	*pSSDConfig*	The SSD configuration structure pointer.
in	*marginLevel*	Read Margin Choice as follows:
		• marginLevel = 0x0U: use the Normal read level
		• marginLevel = 0x1U: use the User read
		• marginLevel = 0x2U: use the Factory read

Returns

operation status

- STATUS_SUCCESS: Operation was successful.

- STATUS_ERROR: Operation failure was occurred.

- STATUS_BUSY: Operation was busy.

16.23.6.23   status_t FLASH_DRV_VerifySection (const flash_ssd_config_t ∗ *pSSDConfig,* uint32_t *dest,* uint16_t *number,* uint8_t *marginLevel*)

Flash verify section.

This API checks if a section of the P-Flash or the D-Flash memory is erased to the specified read margin level.

Parameters

in	*pSSDConfig*	The SSD configuration structure pointer.
in	*dest*	Start address for the intended verify operation.
in	*number*	Number of alignment unit to be verified. Refer to corresponding reference manual to get correct information of alignment constrain.
in	*marginLevel*	Read Margin Choice as follows: • marginLevel = 0x0U: use Normal read level • marginLevel = 0x1U: use the User read • marginLevel = 0x2U: use the Factory read

Returns

operation status

- STATUS_SUCCESS: Operation was successful.

- STATUS_ERROR: Operation failure was occurred.

- STATUS_BUSY: Operation was busy.

16.23.7 Variable Documentation

16.23.7.1 uint8_t brownOutCode

Brown-out detection code

Definition at line 605 of file flash_driver.h.

16.23.7.2 flash_callback_t CallBack

Call back function to service the time critical events. Any code reachable from this function must not be placed in a Flash block targeted for a program/erase operation

Definition at line 565 of file flash_driver.h.

16.23.7.3 flash_callback_t CallBack

Call back function to service the time critical events. Any code reachable from this function must not be placed in a Flash block targeted for a program/erase operation

Definition at line 594 of file flash_driver.h.

16.23.7.4 uint32_t DFlashBase

For FlexNVM device, this is the base address of D-Flash memory (FlexNVM memory); For non-FlexNVM device, this field is unused

Definition at line 561 of file flash_driver.h.

16.23.7.5 uint32_t DFlashBase

For FlexNVM device, this is the base address of D-Flash memory (FlexNVM memory); For non-FlexNVM device, this field is unused

Definition at line 585 of file flash_driver.h.

16.23.7.6 uint32_t DFlashSize

For FlexNVM device, this is the size in byte of area which is used as D-Flash from FlexNVM memory; For non-↩ FlexNVM device, this field is unused

Definition at line 587 of file flash_driver.h.

16.23.7.7   uint32_t EEESize

For FlexNVM device, this is the size in byte of EEPROM area which was partitioned from FlexRAM; For non-Flex←↩
NVM device, this field is unused

Definition at line 592 of file flash_driver.h.

16.23.7.8   uint32_t EERAMBase

The base address of FlexRAM (for FlexNVM device) or acceleration RAM memory (for non-FlexNVM device)

Definition at line 563 of file flash_driver.h.

16.23.7.9   uint32_t EERAMBase

The base address of FlexRAM (for FlexNVM device) or acceleration RAM memory (for non-FlexNVM device)

Definition at line 590 of file flash_driver.h.

16.23.7.10   uint16_t numOfRecordReqMaintain

Number of EEPROM quick write records requiring maintenance

Definition at line 606 of file flash_driver.h.

16.23.7.11   uint32_t PFlashBase

The base address of P-Flash memory

Definition at line 559 of file flash_driver.h.

16.23.7.12   uint32_t PFlashBase

The base address of P-Flash memory

Definition at line 583 of file flash_driver.h.

16.23.7.13   uint32_t PFlashSize

The size in byte of P-Flash memory

Definition at line 560 of file flash_driver.h.

16.23.7.14   uint32_t PFlashSize

The size in byte of P-Flash memory

Definition at line 584 of file flash_driver.h.

16.23.7.15   uint16_t sectorEraseCount

EEPROM sector erase count

Definition at line 607 of file flash_driver.h.

16.24 Flash Memory (Flash)

16.24.1 Detailed Description

Flash Memory Module provides the general flash APIs.

Flash memory is ideal for single-supply applications, permitting in-the-field erase and reprogramming operations without the need for any external high voltage power sources. The flash module includes a memory controller that executes commands to modify flash memory contents. An erased bit reads '1' and a programmed bit reads '0'. The programming operation is unidirectional; it can only move bits from the '1' state (erased) to the '0' state (programmed). Only the erase operation restores bits from '0' to '1'; bits cannot be programmed from a '0' to a '1'.

C90TFS Flash Driver

The C90TFS flash module includes the following accessible memory regions.

1. Program flash memory for vector space and code store.

2. FlexNVM for data store, additional code store and also non-volatile storage for the EEPROM filing system representing data written to the FlexRAM requiring highest endurance.

3. FlexRAM for high-endurance EEPROM data store or traditional RAM.

Some platforms may be designed to have only program flash memory or all of them.

The S32 SDK provides the C90TFS Flash driver of S32K platforms. The driver includes general APIs to handle specific operations on C90TFS Flash module. The user can use those APIs directly in the application.

EEPROM feature

For platforms with FlexNVM, the flash module provides a built-in hardware emulation scheme to emulate the characteristics of an EEPROM by effectively providing a high-endurance, byte write-able NVM. The EEPROM system is shown in the following figure.

Figure 1. EEPROM Architecture

To handle with various customer's requirements, the FlexRAM and FlexNVM blocks can be split into partitions:

1. EEPROM partition(EEESIZE) — The amount of FlexRAM used for EEPROM can be set from 0 Bytes (no EEPROM) to the maximum FlexRAM size. The remainder of the FlexRAM not used for EEPROM is not accessible while the FlexRAM is configured for EEPROM.The EEPROM partition grows upward from the bottom of the FlexRAM address space.

2. Data flash partition(DEPART) — The amount of FlexNVM memory used for data flash can be programmed from 0 bytes (all of the FlexNVM block is available for EEPROM backup) to the maximum size of the FlexNVM block.

3. FlexNVM EEPROM partition — The amount of FlexNVM memory used for EEPROM backup, which is equal to the FlexNVM block size minus the data flash memory partition size. The EEPROM backup size must be at least 16 times the EEPROM partition size in FlexRAM.

The partition information (EEESIZE, DEPART) is programmed using the **#FLASH_DRV_DEFlashPartition** API.

The function of FlexRAM can be changed from EEPROM usage to traditional RAM for accelerate programming in **#FLASH_DRV_ProgramSection** API and vice versa by **#FLASH_DRV_SetFlexRamFunction** API.

This is example code of EEE usage sequence:

```
/* Provide information about the flash blocks. */
const flash_user_config_t flashUserConfig =
{
    0x00000000u,                          /* Base address of Program Flash block */
    FEATURE_FLS_PF_BLOCK_SIZE,            /* Size of Program Flash block */
```

```
                FEATURE_FLS_DF_START_ADDRESS,               /* Base address of Data Flash block */
                FEATURE_FLS_FLEX_RAM_START_ADDRESS,         /* Base address of FlexRAM block */
                NULL_CALLBACK                               /* Pointer to callback function */
    };

    /* Declare a FLASH configuration structure which is initialized by FlashInit, and will be used by all
        flash APIs */
    flash_ssd_config_t flashSSDConfig;

    /* Always initialize the driver before calling other functions */
    ret = FLASH_DRV_Init(&flashUserConfig, &flashSSDConfig);
    if (ret != STATUS_SUCCESS)
    {
        return ret;
    }

#if ((FEATURE_FLS_HAS_FLEX_NVM == 1u) & (FEATURE_FLS_HAS_FLEX_RAM == 1u))
    /* Configure FlexRAM as EEPROM if it is currently used as traditional RAM */
    if (flashSSDConfig.EEESize == 0u)
    {
        /* Configure FlexRAM as EEPROM and FlexNVM as EEPROM backup region,
           DEFlashPartition will be failed if the IFR region isn't blank.
           Refer to the device document for valid EEPROM Data Size Code
           and FlexNVM Partition Code. For example on S32K144:
           - EEEDataSizeCode = 0x02u: EEPROM size = 4 Kbytes
           - DEPartitionCode = 0x08u: EEPROM backup size = 64 Kbytes */
        ret = FLASH_DRV_DEFlashPartition(&flashSSDConfig, 0x02u, 0x08u, 0x0, false, true);
        if (ret != STATUS_SUCCESS)
        {
            return ret;
        }
        else
        {
            /* Re-initialize the driver to update the new EEPROM configuration */
            ret = FLASH_DRV_Init(&flashUserConfig, &flashSSDConfig);
            if (ret != STATUS_SUCCESS)
            {
                return ret;
            }

            /* Make FlexRAM available for EEPROM */
            ret = FLASH_DRV_SetFlexRamFunction(&flashSSDConfig, EEE_ENABLE, 0x0u, NULL);
            if (ret != STATUS_SUCCESS)
            {
                return ret;
            }
        }
    }
    else    /* FLexRAM is already configured as EEPROM */
    {
        /* Make FlexRAM available for EEPROM, make sure that FlexNVM and FlexRAM
           are already partitioned successfully before */
        ret = FLASH_DRV_SetFlexRamFunction(&flashSSDConfig, EEE_ENABLE, 0x0u, NULL);
        if (ret != STATUS_SUCCESS)
        {
            return ret;
        }
    }
#endif
```

Important Note

1. If using callback in the application, any code reachable from this function must not be placed in a Flash block targeted for a program/erase operation to avoid the RWW error. Functions can be placed in RAM section by using the START/END_FUNCTION_DEFINITION/DECLARATION_RAMSECTION macros.

2. To suspend the sector erase operation for a simple method, invoke the **FLASH_DRV_EraseSuspend** function within callback of **FLASH_DRV_EraseSector**. In this case, the **FLASH_DRV_EraseSuspend** must not be placed in the same block in which the Flash erase sector command is going on.

3. **#FLASH_DRV_CommandSequence**, **FLASH_DRV_EraseSuspend** and **FLASH_DRV_EraseResume** should be executed from RAM or different Flash blocks which are targeted for writing to avoid the RWW error. **FLASH_DRV_EraseSuspend** and **FLASH_DRV_EraseResume** functions should be called in pairs.

4. To guarantee the correct execution of this driver, the Flash cache in the Flash memory controller module should be disabled before invoking any API.

5. Partitioning FlexNVM and FlexRAM for EEPROM usage shall be executed only once in the lifetime of the device.

6. After successfully partitioning FlexNVM and FlexRAM for EEPROM usage, user needs to call **FLASH_DR↩ V_Init** to update memory information in global structure.

7. Can not erase or program flash when MCU is high speed run mode or very low power mode.

Modules

- Flash Memory (Flash)

16.25   FlexCAN Driver

16.25.1   Detailed Description

How to use the FlexCAN driver in your application

In order to be able to use the FlexCAN in your application, the first thing to do is initializing it with the desired configuration. This is done by calling the **FLEXCAN_DRV_Init** function. One of the arguments passed to this function is the configuration which will be used for the FlexCAN module, specified by the **flexcan_user_config_t** structure.

The **flexcan_user_config_t** structure allows you to configure the following:

- the number of message buffers needed;

- the number of Rx FIFO ID filters needed;

- enable/disable the Rx FIFO feature;

- the operation mode, which can be one of the following:

  - normal mode;
  - listen-only mode;
  - loopback mode;
  - freeze mode;
  - disable mode;

- the payload size of the message buffers:

  - 8 bytes;
  - 16 bytes (only available with the FD feature enabled);
  - 32 bytes (only available with the FD feature enabled);
  - 64 bytes (only available with the FD feature enabled);

- enable/disable the Flexible Data-rate feature;

- the clock source of the CAN Protocol Engine (PE);

- the bitrate used for standard frames or for the arbitration phase of FD frames;

- the bitrate used for the data phase of FD frames;

- the Rx transfer type, which can be one of the following:

  - using interrupts;
  - using DMA;

- the DMA channel number to be used for DMA transfers;

The bitrate is represented by a **flexcan_time_segment_t** structure, with the following fields:

- propagation segment;

- phase segment 1;

- phase segment 2;

- clock prescaler division factor;

- resync jump width.

Details about these fields can be found in the reference manual.

In order to use a mailbox for transmission/reception, it should be initialized using either **FLEXCAN_DRV_Config**↩
RxMb, **FLEXCAN_DRV_ConfigRxFifo** or **FLEXCAN_DRV_ConfigTxMb**.

After having the mailbox configured, you can start sending/receiving using it by calling one of the following functions:

- FLEXCAN_DRV_Send;

- FLEXCAN_DRV_SendBlocking;

- FLEXCAN_DRV_Receive;

- FLEXCAN_DRV_ReceiveBlocking;

- FLEXCAN_DRV_RxFifo;

- FLEXCAN_DRV_RxFifoBlocking.

A default FlexCAN configuration can be accesed by calling the **FLEXCAN_DRV_GetDefaultConfig** function. This function takes as argument a **flexcan_user_config_t** structure and fills it according to the following settings:

- 16 message buffers

- flexible data rate disabled

- Rx FIFO disabled

- normal operation mode

- 8 byte payload size

- Protocol Engine clock = Oscillator clock

- bitrate of 500 Kbit/s (computed for PE clock = 8 MHz with sample point = 87.5)

Important Notes

- The FlexCAN driver does not handle clock setup or any kind of pin configuration. This is handled by the Clock Manager and PinSettings modules, respectively. The driver assumes that correct clock configurations have been made, so it is the user's responsibility to set up clocking and pin configurations correctly.

- For some platforms, the clock source of the CAN Protocol Engine (PE) is not configurable from the FlexCAN module. If this feature is not supported, the *pe_clock field* from the FlexCAN configuration structure is not present.

- DMA module has to be initialized prior to FlexCAN Rx FIFO usage in DMA mode; also, the DMA channel needs to be allocated by the application (the driver only takes care of configuring the DMA channel received in the configuration structure).

Example:

```
#define INST_CANCOM1 (0U)
#define RX_MAILBOX   (1U)
#define MSG_ID       (2U)

flexcan_state_t canCom1_State;

const flexcan_user_config_t canCom1_InitConfig0 = {
    .fd_enable = true,
    .pe_clock = FLEXCAN_CLK_SOURCE_OSC,
    .max_num_mb = 16,
    .num_id_filters = FLEXCAN_RX_FIFO_ID_FILTERS_8,
    .is_rx_fifo_needed = false,
    .flexcanMode = FLEXCAN_NORMAL_MODE,
    .payload = FLEXCAN_PAYLOAD_SIZE_8,
    .bitrate = {
        .propSeg = 7,
        .phaseSeg1 = 4,
```

```
        .phaseSeg2 = 1,
        .preDivider = 0,
        .rJumpwidth = 1
    },
    .bitrate_cbt = {
        .propSeg = 11,
        .phaseSeg1 = 1,
        .phaseSeg2 = 1,
        .preDivider = 0,
        .rJumpwidth = 1
    },
    .transfer_type = FLEXCAN_RXFIFO_USING_INTERRUPTS,
    .rxFifoDMAChannel = 0U
};

/* Initialize FlexCAN driver */
FLEXCAN_DRV_Init(INST_CANCOM1, &canCom1_State, &canCom1_InitConfig0);

/* Set information about the data to be received */
flexcan_data_info_t dataInfo =
{
    .data_length = 1U,
    .msg_id_type = FLEXCAN_MSG_ID_STD,
    .enable_brs  = true,
    .fd_enable   = true,
    .fd_padding  = 0U
};

/* Configure Rx message buffer with index 1 to receive frames with ID 2 */
FLEXCAN_DRV_ConfigRxMb(INST_CANCOM1, RX_MAILBOX, &dataInfo, MSG_ID);

/* Receive a frame in the recvBuff variable */
flexcan_msgbuff_t recvBuff;

FLEXCAN_DRV_Receive(INST_CANCOM1, RX_MAILBOX, &recvBuff);
/* Wait for the message to be received */
while (FLEXCAN_DRV_GetTransferStatus(INST_CANCOM1, RX_MAILBOX) == STATUS_BUSY)
    ;

/* De-initialize driver */
FLEXCAN_DRV_Deinit(INST_CANCOM1);
```

Data Structures

- struct flexcan_msgbuff_t

    FlexCAN message buffer structure Implements : flexcan_msgbuff_t_Class. More...

- struct flexcan_mb_handle_t

    Information needed for internal handling of a given MB. Implements : flexcan_mb_handle_t_Class. More...

- struct FlexCANState

    Internal driver state information. More...

- struct flexcan_data_info_t

    FlexCAN data info from user Implements : flexcan_data_info_t_Class. More...

- struct flexcan_id_table_t

    FlexCAN Rx FIFO ID filter table structure Implements : flexcan_id_table_t_Class. More...

- struct flexcan_time_segment_t

    FlexCAN bitrate related structures Implements : flexcan_time_segment_t_Class. More...

- struct flexcan_user_config_t

    FlexCAN configuration. More...

Typedefs

- typedef struct FlexCANState flexcan_state_t

    Internal driver state information.

- typedef void(∗ flexcan_callback_t) (uint8_t instance, flexcan_event_type_t eventType, uint32_t buffIdx, flexcan_state_t ∗flexcanState)

    FlexCAN Driver callback function type Implements : flexcan_callback_t_Class.

Enumerations

- enum flexcan_rxfifo_transfer_type_t { FLEXCAN_RXFIFO_USING_INTERRUPTS }

  The type of the RxFIFO transfer (interrupts/DMA). Implements : flexcan_rxfifo_transfer_type_t_Class.
- enum flexcan_event_type_t { FLEXCAN_EVENT_RX_COMPLETE, FLEXCAN_EVENT_RXFIFO_COMPL↩
  ETE, FLEXCAN_EVENT_TX_COMPLETE }

  The type of the event which occurred when the callback was invoked. Implements : flexcan_event_type_t_Class.
- enum flexcan_mb_state_t { FLEXCAN_MB_IDLE, FLEXCAN_MB_RX_BUSY, FLEXCAN_MB_TX_BUSY }

  The state of a given MB (idle/Rx busy/Tx busy). Implements : flexcan_mb_state_t_Class.
- enum flexcan_msgbuff_id_type_t { FLEXCAN_MSG_ID_STD, FLEXCAN_MSG_ID_EXT }

  FlexCAN Message Buffer ID type Implements : flexcan_msgbuff_id_type_t_Class.
- enum flexcan_rx_fifo_id_filter_num_t {
  FLEXCAN_RX_FIFO_ID_FILTERS_8 = 0x0, FLEXCAN_RX_FIFO_ID_FILTERS_16 = 0x1, FLEXCAN_R↩
  X_FIFO_ID_FILTERS_24 = 0x2, FLEXCAN_RX_FIFO_ID_FILTERS_32 = 0x3,
  FLEXCAN_RX_FIFO_ID_FILTERS_40 = 0x4, FLEXCAN_RX_FIFO_ID_FILTERS_48 = 0x5, FLEXCAN_↩
  RX_FIFO_ID_FILTERS_56 = 0x6, FLEXCAN_RX_FIFO_ID_FILTERS_64 = 0x7,
  FLEXCAN_RX_FIFO_ID_FILTERS_72 = 0x8, FLEXCAN_RX_FIFO_ID_FILTERS_80 = 0x9, FLEXCAN_↩
  RX_FIFO_ID_FILTERS_88 = 0xA, FLEXCAN_RX_FIFO_ID_FILTERS_96 = 0xB,
  FLEXCAN_RX_FIFO_ID_FILTERS_104 = 0xC, FLEXCAN_RX_FIFO_ID_FILTERS_112 = 0xD, FLEXCA↩
  N_RX_FIFO_ID_FILTERS_120 = 0xE, FLEXCAN_RX_FIFO_ID_FILTERS_128 = 0xF }

  FlexCAN Rx FIFO filters number Implements : flexcan_rx_fifo_id_filter_num_t_Class.
- enum flexcan_rx_mask_type_t { FLEXCAN_RX_MASK_GLOBAL, FLEXCAN_RX_MASK_INDIVIDUAL }

  FlexCAN Rx mask type. Implements : flexcan_rx_mask_type_t_Class.
- enum flexcan_rx_fifo_id_element_format_t { FLEXCAN_RX_FIFO_ID_FORMAT_A, FLEXCAN_RX_FIFO↩
  _ID_FORMAT_B, FLEXCAN_RX_FIFO_ID_FORMAT_C, FLEXCAN_RX_FIFO_ID_FORMAT_D }

  ID formats for Rx FIFO Implements : flexcan_rx_fifo_id_element_format_t_Class.
- enum flexcan_operation_modes_t {
  FLEXCAN_NORMAL_MODE, FLEXCAN_LISTEN_ONLY_MODE, FLEXCAN_LOOPBACK_MODE, FLEX↩
  CAN_FREEZE_MODE,
  FLEXCAN_DISABLE_MODE }

  FlexCAN operation modes Implements : flexcan_operation_modes_t_Class.

Bit rate

- void FLEXCAN_DRV_SetBitrate (uint8_t instance, const flexcan_time_segment_t ∗bitrate)

  Sets the FlexCAN bit rate for standard frames or the arbitration phase of FD frames.
- void FLEXCAN_DRV_GetBitrate (uint8_t instance, flexcan_time_segment_t ∗bitrate)

  Gets the FlexCAN bit rate for standard frames or the arbitration phase of FD frames.

Rx MB and Rx FIFO masks

- void FLEXCAN_DRV_SetRxMaskType (uint8_t instance, flexcan_rx_mask_type_t type)

  Sets the Rx masking type.
- void FLEXCAN_DRV_SetRxFifoGlobalMask (uint8_t instance, flexcan_msgbuff_id_type_t id_type, uint32_t
  mask)

  Sets the FlexCAN Rx FIFO global mask (standard or extended).
- void FLEXCAN_DRV_SetRxMbGlobalMask (uint8_t instance, flexcan_msgbuff_id_type_t id_type, uint32_t
  mask)

  Sets the FlexCAN Rx MB global mask (standard or extended).
- void FLEXCAN_DRV_SetRxMb14Mask (uint8_t instance, flexcan_msgbuff_id_type_t id_type, uint32_t mask)

  Sets the FlexCAN Rx MB 14 mask (standard or extended).
- void FLEXCAN_DRV_SetRxMb15Mask (uint8_t instance, flexcan_msgbuff_id_type_t id_type, uint32_t mask)

  Sets the FlexCAN Rx MB 15 mask (standard or extended).

- status_t FLEXCAN_DRV_SetRxIndividualMask (uint8_t instance, flexcan_msgbuff_id_type_t id_type, uint8↵ _t mb_idx, uint32_t mask)

    Sets the FlexCAN Rx individual mask (standard or extended).

Initialization and Shutdown

- void FLEXCAN_DRV_GetDefaultConfig (flexcan_user_config_t ∗config)

    Gets the default configuration structure.

- status_t FLEXCAN_DRV_Init (uint8_t instance, flexcan_state_t ∗state, const flexcan_user_config_t ∗data)

    Initializes the FlexCAN peripheral.

- status_t FLEXCAN_DRV_Deinit (uint8_t instance)

    Shuts down a FlexCAN instance.

Send configuration

- status_t FLEXCAN_DRV_ConfigTxMb (uint8_t instance, uint8_t mb_idx, const flexcan_data_info_t ∗tx_info, uint32_t msg_id)

    FlexCAN transmit message buffer field configuration.

- status_t FLEXCAN_DRV_SendBlocking (uint8_t instance, uint8_t mb_idx, const flexcan_data_info_t ∗tx_info, uint32_t msg_id, const uint8_t ∗mb_data, uint32_t timeout_ms)

    Sends a CAN frame using the specified message buffer, in a blocking manner.

- status_t FLEXCAN_DRV_Send (uint8_t instance, uint8_t mb_idx, const flexcan_data_info_t ∗tx_info, uint32_t msg_id, const uint8_t ∗mb_data)

    Sends a CAN frame using the specified message buffer.

Receive configuration

- status_t FLEXCAN_DRV_ConfigRxMb (uint8_t instance, uint8_t mb_idx, const flexcan_data_info_t ∗rx_info, uint32_t msg_id)

    FlexCAN receive message buffer field configuration.

- void FLEXCAN_DRV_ConfigRxFifo (uint8_t instance, flexcan_rx_fifo_id_element_format_t id_format, const flexcan_id_table_t ∗id_filter_table)

    FlexCAN Rx FIFO field configuration.

- status_t FLEXCAN_DRV_ReceiveBlocking (uint8_t instance, uint8_t mb_idx, flexcan_msgbuff_t ∗data, uint32_t timeout_ms)

    Receives a CAN frame using the specified message buffer, in a blocking manner.

- status_t FLEXCAN_DRV_Receive (uint8_t instance, uint8_t mb_idx, flexcan_msgbuff_t ∗data)

    Receives a CAN frame using the specified message buffer.

- status_t FLEXCAN_DRV_RxFifoBlocking (uint8_t instance, flexcan_msgbuff_t ∗data, uint32_t timeout_ms)

    Receives a CAN frame using the message FIFO, in a blocking manner.

- status_t FLEXCAN_DRV_RxFifo (uint8_t instance, flexcan_msgbuff_t ∗data)

    Receives a CAN frame using the message FIFO.

Transfer status

- status_t FLEXCAN_DRV_AbortTransfer (uint8_t instance, uint8_t mb_idx)

    Ends a non-blocking FlexCAN transfer early.

- status_t FLEXCAN_DRV_GetTransferStatus (uint8_t instance, uint8_t mb_idx)

    Returns whether the previous FlexCAN transfer has finished.

IRQ handler callback

- void FLEXCAN_DRV_InstallEventCallback (uint8_t instance, flexcan_callback_t callback, void *callback↩
Param)

    Installs a callback function for the IRQ handler.

16.25.2 Data Structure Documentation

16.25.2.1 struct flexcan_msgbuff_t

FlexCAN message buffer structure Implements : flexcan_msgbuff_t_Class.

Definition at line 93 of file flexcan_driver.h.

Data Fields

- uint32_t cs
- uint32_t msgId
- uint8_t data [64]
- uint8_t dataLen

Field Documentation

16.25.2.1.1 uint32_t cs

Code and Status

Definition at line 94 of file flexcan_driver.h.

16.25.2.1.2 uint8_t data[64]

Data bytes of the FlexCAN message

Definition at line 96 of file flexcan_driver.h.

16.25.2.1.3 uint8_t dataLen

Length of data in bytes

Definition at line 97 of file flexcan_driver.h.

16.25.2.1.4 uint32_t msgId

Message Buffer ID

Definition at line 95 of file flexcan_driver.h.

16.25.2.2 struct flexcan_mb_handle_t

Information needed for internal handling of a given MB. Implements : flexcan_mb_handle_t_Class.

Definition at line 103 of file flexcan_driver.h.

Data Fields

- flexcan_msgbuff_t * mb_message
- semaphore_t mbSema
- volatile flexcan_mb_state_t state
- bool isBlocking
- bool isRemote

Field Documentation

16.25.2.2.1   bool isBlocking

True if the transfer is blocking

Definition at line 107 of file flexcan_driver.h.

16.25.2.2.2   bool isRemote

True if the frame is a remote frame

Definition at line 108 of file flexcan_driver.h.

16.25.2.2.3   flexcan_msgbuff_t∗ mb_message

The FlexCAN MB structure

Definition at line 104 of file flexcan_driver.h.

16.25.2.2.4   semaphore_t mbSema

Semaphore used for signaling completion of a blocking transfer

Definition at line 105 of file flexcan_driver.h.

16.25.2.2.5   volatile flexcan_mb_state_t state

The state of the current MB (idle/Rx busy/Tx busy)

Definition at line 106 of file flexcan_driver.h.

16.25.2.3   struct FlexCANState

Internal driver state information.

Note

> The contents of this structure are internal to the driver and should not be modified by users. Also, contents of the structure are subject to change in future releases. Implements : flexcan_state_t_Class

Definition at line 119 of file flexcan_driver.h.

Data Fields

- flexcan_mb_handle_t mbs [FEATURE_CAN_MAX_MB_NUM]
- void(∗ callback)(uint8_t instance, flexcan_event_type_t eventType, uint32_t buffIdx, struct FlexCANState ∗driverState)
- void ∗ callbackParam
- flexcan_rxfifo_transfer_type_t transferType

Field Documentation

16.25.2.3.1   void(∗ callback) (uint8_t instance, flexcan_event_type_t eventType, uint32_t buffIdx, struct FlexCANState ∗driverState)

IRQ handler callback function.

Definition at line 121 of file flexcan_driver.h.

16.25.2.3.2   void∗ callbackParam

Parameter used to pass user data when invoking the callback function.

Definition at line 123 of file flexcan_driver.h.

16.25.2.3.3 flexcan_mb_handle_t mbs[FEATURE_CAN_MAX_MB_NUM]

Array containing information related to each MB

Definition at line 120 of file flexcan_driver.h.

16.25.2.3.4 flexcan_rxfifo_transfer_type_t transferType

Type of RxFIFO transfer.

Definition at line 127 of file flexcan_driver.h.

16.25.2.4 struct flexcan_data_info_t

FlexCAN data info from user Implements : flexcan_data_info_t_Class.

Definition at line 133 of file flexcan_driver.h.

Data Fields

- flexcan_msgbuff_id_type_t msg_id_type
- uint32_t data_length
- bool is_remote

Field Documentation

16.25.2.4.1 uint32_t data_length

Length of Data in Bytes

Definition at line 135 of file flexcan_driver.h.

16.25.2.4.2 bool is_remote

Specifies if the frame is standard or remote

Definition at line 142 of file flexcan_driver.h.

16.25.2.4.3 flexcan_msgbuff_id_type_t msg_id_type

Type of message ID (standard or extended)

Definition at line 134 of file flexcan_driver.h.

16.25.2.5 struct flexcan_id_table_t

FlexCAN Rx FIFO ID filter table structure Implements : flexcan_id_table_t_Class.

Definition at line 189 of file flexcan_driver.h.

Data Fields

- bool isRemoteFrame
- bool isExtendedFrame
- uint32_t id

Field Documentation

16.25.2.5.1 uint32_t id

Rx FIFO ID filter element

Definition at line 192 of file flexcan_driver.h.

16.25.2.5.2   bool isExtendedFrame

Extended frame

Definition at line 191 of file flexcan_driver.h.

16.25.2.5.3   bool isRemoteFrame

Remote frame

Definition at line 190 of file flexcan_driver.h.

16.25.2.6   struct flexcan_time_segment_t

FlexCAN bitrate related structures Implements : flexcan_time_segment_t_Class.

Definition at line 221 of file flexcan_driver.h.

Data Fields

- uint32_t propSeg
- uint32_t phaseSeg1
- uint32_t phaseSeg2
- uint32_t preDivider
- uint32_t rJumpwidth

Field Documentation

16.25.2.6.1   uint32_t phaseSeg1

Phase segment 1

Definition at line 223 of file flexcan_driver.h.

16.25.2.6.2   uint32_t phaseSeg2

Phase segment 2

Definition at line 224 of file flexcan_driver.h.

16.25.2.6.3   uint32_t preDivider

Clock prescaler division factor

Definition at line 225 of file flexcan_driver.h.

16.25.2.6.4   uint32_t propSeg

Propagation segment

Definition at line 222 of file flexcan_driver.h.

16.25.2.6.5   uint32_t rJumpwidth

Resync jump width

Definition at line 226 of file flexcan_driver.h.

16.25.2.7   struct flexcan_user_config_t

FlexCAN configuration.

Definition at line 233 of file flexcan_driver.h.

Data Fields

- uint32_t max_num_mb
- flexcan_rx_fifo_id_filter_num_t num_id_filters
- bool is_rx_fifo_needed
- flexcan_operation_modes_t flexcanMode
- flexcan_time_segment_t bitrate
- flexcan_rxfifo_transfer_type_t transfer_type

Field Documentation

16.25.2.7.1 flexcan_time_segment_t bitrate

The bitrate used for standard frames or for the arbitration phase of FD frames.

Definition at line 249 of file flexcan_driver.h.

16.25.2.7.2 flexcan_operation_modes_t flexcanMode

User configurable FlexCAN operation modes.

Definition at line 240 of file flexcan_driver.h.

16.25.2.7.3 bool is_rx_fifo_needed

1 if needed; 0 if not. This controls whether the Rx FIFO feature is enabled or not.

Definition at line 238 of file flexcan_driver.h.

16.25.2.7.4 uint32_t max_num_mb

The maximum number of Message Buffers

Definition at line 234 of file flexcan_driver.h.

16.25.2.7.5 flexcan_rx_fifo_id_filter_num_t num_id_filters

The number of RX FIFO ID filters needed

Definition at line 236 of file flexcan_driver.h.

16.25.2.7.6 flexcan_rxfifo_transfer_type_t transfer_type

Specifies if the Rx FIFO uses interrupts or DMA.

Definition at line 253 of file flexcan_driver.h.

16.25.3 Typedef Documentation

16.25.3.1 typedef void(∗ flexcan_callback_t) (uint8_t instance, flexcan_event_type_t eventType, uint32_t buffIdx, flexcan_state_t ∗flexcanState)

FlexCAN Driver callback function type Implements : flexcan_callback_t_Class.

Definition at line 315 of file flexcan_driver.h.

16.25.3.2 typedef struct FlexCANState flexcan_state_t

Internal driver state information.

Note

> The contents of this structure are internal to the driver and should not be modified by users. Also, contents of the structure are subject to change in future releases. Implements : flexcan_state_t_Class

16.25.4   Enumeration Type Documentation

16.25.4.1   enum flexcan_event_type_t

The type of the event which occurred when the callback was invoked. Implements : flexcan_event_type_t_Class.

Enumerator

> ***FLEXCAN_EVENT_RX_COMPLETE***   A frame was received in the configured Rx MB.
> ***FLEXCAN_EVENT_RXFIFO_COMPLETE***   A frame was received in the RxFIFO.
> ***FLEXCAN_EVENT_TX_COMPLETE***   A frame was sent from the configured Tx MB.

Definition at line 52 of file flexcan_driver.h.

16.25.4.2   enum flexcan_mb_state_t

The state of a given MB (idle/Rx busy/Tx busy). Implements : flexcan_mb_state_t_Class.

Enumerator

> ***FLEXCAN_MB_IDLE***   The MB is not used by any transfer.
> ***FLEXCAN_MB_RX_BUSY***   The MB is used for a reception.
> ***FLEXCAN_MB_TX_BUSY***   The MB is used for a transmission.

Definition at line 66 of file flexcan_driver.h.

16.25.4.3   enum flexcan_msgbuff_id_type_t

FlexCAN Message Buffer ID type Implements : flexcan_msgbuff_id_type_t_Class.

Enumerator

> ***FLEXCAN_MSG_ID_STD***   Standard ID
> ***FLEXCAN_MSG_ID_EXT***   Extended ID

Definition at line 75 of file flexcan_driver.h.

16.25.4.4   enum flexcan_operation_modes_t

FlexCAN operation modes Implements : flexcan_operation_modes_t_Class.

Enumerator

> ***FLEXCAN_NORMAL_MODE***   Normal mode or user mode
> ***FLEXCAN_LISTEN_ONLY_MODE***   Listen-only mode
> ***FLEXCAN_LOOPBACK_MODE***   Loop-back mode
> ***FLEXCAN_FREEZE_MODE***   Freeze mode
> ***FLEXCAN_DISABLE_MODE***   Module disable mode

Definition at line 198 of file flexcan_driver.h.

16.25.4.5   enum flexcan_rx_fifo_id_element_format_t

ID formats for Rx FIFO Implements : flexcan_rx_fifo_id_element_format_t_Class.

Enumerator

> ***FLEXCAN_RX_FIFO_ID_FORMAT_A***   One full ID (standard and extended) per ID Filter Table element.

FLEXCAN_RX_FIFO_ID_FORMAT_B Two full standard IDs or two partial 14-bit (standard and extended) IDs per ID Filter Table element.

FLEXCAN_RX_FIFO_ID_FORMAT_C Four partial 8-bit Standard IDs per ID Filter Table element.

FLEXCAN_RX_FIFO_ID_FORMAT_D All frames rejected.

Definition at line 178 of file flexcan_driver.h.

16.25.4.6   enum flexcan_rx_fifo_id_filter_num_t

FlexCAN Rx FIFO filters number Implements : flexcan_rx_fifo_id_filter_num_t_Class.

Enumerator

FLEXCAN_RX_FIFO_ID_FILTERS_8 8 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_16 16 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_24 24 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_32 32 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_40 40 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_48 48 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_56 56 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_64 64 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_72 72 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_80 80 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_88 88 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_96 96 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_104 104 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_112 112 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_120 120 Rx FIFO Filters.
FLEXCAN_RX_FIFO_ID_FILTERS_128 128 Rx FIFO Filters.

Definition at line 148 of file flexcan_driver.h.

16.25.4.7   enum flexcan_rx_mask_type_t

FlexCAN Rx mask type. Implements : flexcan_rx_mask_type_t_Class.

Enumerator

FLEXCAN_RX_MASK_GLOBAL Rx global mask
FLEXCAN_RX_MASK_INDIVIDUAL Rx individual mask

Definition at line 170 of file flexcan_driver.h.

16.25.4.8   enum flexcan_rxfifo_transfer_type_t

The type of the RxFIFO transfer (interrupts/DMA). Implements : flexcan_rxfifo_transfer_type_t_Class.

Enumerator

FLEXCAN_RXFIFO_USING_INTERRUPTS Use interrupts for RxFIFO.

Definition at line 42 of file flexcan_driver.h.

16.25.5   Function Documentation

16.25.5.1   status_t FLEXCAN_DRV_AbortTransfer (uint8_t *instance,* uint8_t *mb_idx*)

Ends a non-blocking FlexCAN transfer early.

Parameters

instance	A FlexCAN instance number
mb_idx	The index of the message buffer

Returns

> STATUS_SUCCESS if successful; STATUS_FLEXCAN_NO_TRANSFER_IN_PROGRESS if no transfer was running

Definition at line 1309 of file flexcan_driver.c.

16.25.5.2   void FLEXCAN_DRV_ConfigRxFifo (uint8_t *instance,* flexcan_rx_fifo_id_element_format_t *id_format,* const flexcan_id_table_t ∗ *id_filter_table*)

FlexCAN Rx FIFO field configuration.

Note

> The number of elements in the ID filter table is defined by the following formula:
>
> - for format A: the number of Rx FIFO ID filters
> - for format B: twice the number of Rx FIFO ID filters
>
> - for format C: four times the number of Rx FIFO ID filters The user must provide the exact number of elements in order to avoid any misconfiguration.

Parameters

instance	A FlexCAN instance number
id_format	The format of the Rx FIFO ID Filter Table Elements
id_filter_table	The ID filter table elements which contain RTR bit, IDE bit, and Rx message ID

Definition at line 801 of file flexcan_driver.c.

16.25.5.3   status_t FLEXCAN_DRV_ConfigRxMb (uint8_t *instance,* uint8_t *mb_idx,* const flexcan_data_info_t ∗ *rx_info,* uint32_t *msg_id*)

FlexCAN receive message buffer field configuration.

Parameters

instance	A FlexCAN instance number
mb_idx	Index of the message buffer
rx_info	Data info
msg_id	ID of the message to transmit

Returns

> STATUS_SUCCESS if successful; STATUS_FLEXCAN_MB_OUT_OF_RANGE if the index of a message buffer is invalid;

Definition at line 753 of file flexcan_driver.c.

16.25.5.4   status_t FLEXCAN_DRV_ConfigTxMb (uint8_t *instance,* uint8_t *mb_idx,* const flexcan_data_info_t ∗ *tx_info,* uint32_t *msg_id*)

FlexCAN transmit message buffer field configuration.

Parameters

instance	A FlexCAN instance number
mb_idx	Index of the message buffer
tx_info	Data info
msg_id	ID of the message to transmit

Returns

> STATUS_SUCCESS if successful; STATUS_FLEXCAN_MB_OUT_OF_RANGE if the index of the message buffer is invalid

Definition at line 626 of file flexcan_driver.c.

16.25.5.5   status_t FLEXCAN_DRV_Deinit (uint8_t *instance*)

Shuts down a FlexCAN instance.

Parameters

instance	A FlexCAN instance number

Returns

> STATUS_SUCCESS if successful; STATUS_ERROR if failed

Definition at line 965 of file flexcan_driver.c.

16.25.5.6   void FLEXCAN_DRV_GetBitrate (uint8_t *instance,* flexcan_time_segment_t ∗ *bitrate*)

Gets the FlexCAN bit rate for standard frames or the arbitration phase of FD frames.

Parameters

instance	A FlexCAN instance number
bitrate	A pointer to a variable for returning the FlexCAN bit rate settings

Definition at line 199 of file flexcan_driver.c.

16.25.5.7   void FLEXCAN_DRV_GetDefaultConfig (flexcan_user_config_t ∗ *config*)

Gets the default configuration structure.

This function gets the default configuration structure, with the following settings:

- 16 message buffers

- flexible data rate disabled

- Rx FIFO disabled

- normal operation mode

- 8 byte payload size

- Protocol Engine clock = Oscillator clock

- bitrate of 500 Kbit/s (computed for PE clock = 8 MHz with sample point = 87.5)

Parameters

out	config	The configuration structure

Definition at line 1750 of file flexcan_driver.c.

16.25.5.8 status_t FLEXCAN_DRV_GetTransferStatus (uint8_t *instance,* uint8_t *mb_idx*)

Returns whether the previous FlexCAN transfer has finished.

When performing an async transfer, call this function to ascertain the state of the current transfer: in progress (or busy) or complete (success).

Parameters

instance	The FlexCAN instance number.
mb_idx	The index of the message buffer.

Returns

> STATUS_SUCCESS if successful; STATUS_BUSY if a resource is busy;

Definition at line 1280 of file flexcan_driver.c.

16.25.5.9 status_t FLEXCAN_DRV_Init (uint8_t *instance,* flexcan_state_t ∗ *state,* const flexcan_user_config_t ∗ *data*)

Initializes the FlexCAN peripheral.

This function initializes

Parameters

instance	A FlexCAN instance number
state	Pointer to the FlexCAN driver state structure.
data	The FlexCAN platform data

Returns

> STATUS_SUCCESS if successful; STATUS_FLEXCAN_MB_OUT_OF_RANGE if the index of a message buffer is invalid; STATUS_ERROR if other error occurred

Definition at line 462 of file flexcan_driver.c.

16.25.5.10 void FLEXCAN_DRV_InstallEventCallback (uint8_t *instance,* flexcan_callback_t *callback,* void ∗ *callbackParam*)

Installs a callback function for the IRQ handler.

Parameters

instance	The FlexCAN instance number.
callback	The callback function.
callbackParam	User parameter passed to the callback function through the state parameter.

Definition at line 1627 of file flexcan_driver.c.

16.25.5.11 status_t FLEXCAN_DRV_Receive (uint8_t *instance,* uint8_t *mb_idx,* flexcan_msgbuff_t ∗ *data*)

Receives a CAN frame using the specified message buffer.

This function receives a CAN frame using a configured message buffer. The function returns immediately. If a callback is installed, it will be invoked after the frame was received and read into the specified buffer.

Parameters

instance	A FlexCAN instance number
mb_idx	Index of the message buffer
data	The FlexCAN receive message buffer data.

Returns

> STATUS_SUCCESS if successful; STATUS_FLEXCAN_MB_OUT_OF_RANGE if the index of a message buffer is invalid; STATUS_BUSY if a resource is busy

Definition at line 876 of file flexcan_driver.c.

16.25.5.12   status_t FLEXCAN_DRV_ReceiveBlocking (uint8_t *instance,* uint8_t *mb_idx,* flexcan_msgbuff_t ∗ *data,* uint32_t *timeout_ms*)

Receives a CAN frame using the specified message buffer, in a blocking manner.

This function receives a CAN frame using a configured message buffer. The function blocks until either a frame was received, or the specified timeout expired.

Parameters

instance	A FlexCAN instance number
mb_idx	Index of the message buffer
data	The FlexCAN receive message buffer data.
timeout_ms	A timeout for the transfer in milliseconds.

Returns

> STATUS_SUCCESS if successful; STATUS_FLEXCAN_MB_OUT_OF_RANGE if the index of a message buffer is invalid; STATUS_BUSY if a resource is busy; STATUS_TIMEOUT if the timeout is reached

Definition at line 827 of file flexcan_driver.c.

16.25.5.13   status_t FLEXCAN_DRV_RxFifo (uint8_t *instance,* flexcan_msgbuff_t ∗ *data*)

Receives a CAN frame using the message FIFO.

This function receives a CAN frame using the Rx FIFO. The function returns immediately. If a callback is installed, it will be invoked after the frame was received and read into the specified buffer.

Parameters

instance	A FlexCAN instance number
data	The FlexCAN receive message buffer data.

Returns

> STATUS_SUCCESS if successful; STATUS_BUSY if a resource is busy; STATUS_ERROR if other error occurred

Definition at line 944 of file flexcan_driver.c.

16.25.5.14   status_t FLEXCAN_DRV_RxFifoBlocking (uint8_t *instance,* flexcan_msgbuff_t ∗ *data,* uint32_t *timeout_ms*)

Receives a CAN frame using the message FIFO, in a blocking manner.

This function receives a CAN frame using the Rx FIFO. The function blocks until either a frame was received, or the specified timeout expired.

Parameters

instance	A FlexCAN instance number
data	The FlexCAN receive message buffer data.
timeout_ms	A timeout for the transfer in milliseconds.

Returns

> STATUS_SUCCESS if successful; STATUS_BUSY if a resource is busy; STATUS_TIMEOUT if the timeout is reached; STATUS_ERROR if other error occurred

Definition at line 899 of file flexcan_driver.c.

16.25.5.15   status_t FLEXCAN_DRV_Send (uint8_t *instance,* uint8_t *mb_idx,* const flexcan_data_info_t ∗ *tx_info,* uint32_t *msg_id,* const uint8_t ∗ *mb_data*)

Sends a CAN frame using the specified message buffer.

This function sends a CAN frame using a configured message buffer. The function returns immediately. If a callback is installed, it will be invoked after the frame was sent.

Parameters

instance	A FlexCAN instance number
mb_idx	Index of the message buffer
tx_info	Data info
msg_id	ID of the message to transmit
mb_data	Bytes of the FlexCAN message.

Returns

> STATUS_SUCCESS if successful; STATUS_FLEXCAN_MB_OUT_OF_RANGE if the index of a message buffer is invalid; STATUS_BUSY if a resource is busy

Definition at line 714 of file flexcan_driver.c.

16.25.5.16   status_t FLEXCAN_DRV_SendBlocking (uint8_t *instance,* uint8_t *mb_idx,* const flexcan_data_info_t ∗ *tx_info,* uint32_t *msg_id,* const uint8_t ∗ *mb_data,* uint32_t *timeout_ms*)

Sends a CAN frame using the specified message buffer, in a blocking manner.

This function sends a CAN frame using a configured message buffer. The function blocks until either the frame was sent, or the specified timeout expired.

Parameters

instance	A FlexCAN instance number
mb_idx	Index of the message buffer
tx_info	Data info
msg_id	ID of the message to transmit
mb_data	Bytes of the FlexCAN message
timeout_ms	A timeout for the transfer in milliseconds.

Returns

> STATUS_SUCCESS if successful; STATUS_FLEXCAN_MB_OUT_OF_RANGE if the index of a message buffer is invalid; STATUS_BUSY if a resource is busy; STATUS_TIMEOUT if the timeout is reached

Definition at line 661 of file flexcan_driver.c.

16.25.5.17   void FLEXCAN_DRV_SetBitrate (uint8_t *instance,* const flexcan_time_segment_t ∗ *bitrate*)

Sets the FlexCAN bit rate for standard frames or the arbitration phase of FD frames.

Parameters

instance	A FlexCAN instance number
bitrate	A pointer to the FlexCAN bit rate settings.

Definition at line 135 of file flexcan_driver.c.

16.25.5.18 void FLEXCAN_DRV_SetRxFifoGlobalMask (uint8_t *instance,* flexcan_msgbuff_id_type_t *id_type,* uint32_t *mask*)

Sets the FlexCAN Rx FIFO global mask (standard or extended).

Parameters

instance	A FlexCAN instance number
id_type	Standard ID or extended ID
mask	Mask value

Definition at line 270 of file flexcan_driver.c.

16.25.5.19 status_t FLEXCAN_DRV_SetRxIndividualMask (uint8_t *instance,* flexcan_msgbuff_id_type_t *id_type,* uint8_t *mb_idx,* uint32_t *mask*)

Sets the FlexCAN Rx individual mask (standard or extended).

Parameters

instance	A FlexCAN instance number
id_type	A standard ID or an extended ID
mb_idx	Index of the message buffer
mask	Mask value

Returns

> STATUS_SUCCESS if successful; STATUS_FLEXCAN_MB_OUT_OF_RANGE if the index of the message buffer is invalid

Definition at line 414 of file flexcan_driver.c.

16.25.5.20 void FLEXCAN_DRV_SetRxMaskType (uint8_t *instance,* flexcan_rx_mask_type_t *type*)

Sets the Rx masking type.

Parameters

instance	A FlexCAN instance number
type	The FlexCAN RX mask type

Definition at line 249 of file flexcan_driver.c.

16.25.5.21 void FLEXCAN_DRV_SetRxMb14Mask (uint8_t *instance,* flexcan_msgbuff_id_type_t *id_type,* uint32_t *mask*)

Sets the FlexCAN Rx MB 14 mask (standard or extended).

Parameters

instance	A FlexCAN instance number
id_type	Standard ID or extended ID
mask	Mask value

Definition at line 342 of file flexcan_driver.c.

16.25.5.22 void FLEXCAN_DRV_SetRxMb15Mask (uint8_t *instance,* flexcan_msgbuff_id_type_t *id_type,* uint32_t *mask*)

Sets the FlexCAN Rx MB 15 mask (standard or extended).

Parameters

instance	A FlexCAN instance number
id_type	Standard ID or extended ID
mask	Mask value

Definition at line 378 of file flexcan_driver.c.

16.25.5.23 void FLEXCAN_DRV_SetRxMbGlobalMask (uint8_t *instance,* flexcan_msgbuff_id_type_t *id_type,* uint32_t *mask*)

Sets the FlexCAN Rx MB global mask (standard or extended).

Parameters

instance	A FlexCAN instance number
id_type	Standard ID or extended ID
mask	Mask value

Definition at line 306 of file flexcan_driver.c.

16.26 FlexIO Common Driver

16.26.1 Detailed Description

Common services for FlexIO drivers.

The Flexio Common driver layer contains services used by all Flexio drivers. The need for this layer derives from the requirement to allow multiple Flexio drivers to run in parallel on the same device, to the extent that enough hardware resources (shifters and timers) are available.

Functionality

The Flexio Common driver layer provides functions for device initialization and reset. Before using any Flexio driver the device must first be initialized using function FLEXIO_DRV_InitDevice(). Then any number of Flexio drivers can be initialized on the same device, to the extent that enough hardware resources (shifters and timers) are available. Driver initialization functions will return STATUS_ERROR if not enough resources are available for a new driver.

Important Notes

Calling any Flexio common function will destroy any driver that is active on that device. Normally these functions should be called only when there are no active driver instances on the device.

Enumerations

- enum flexio_driver_type_t { FLEXIO_DRIVER_TYPE_INTERRUPTS = 0U, FLEXIO_DRIVER_TYPE_POL↩
LING = 1U, FLEXIO_DRIVER_TYPE_DMA = 2U }

    Driver type: interrupts/polling/DMA Implements : flexio_driver_type_t_Class.

FLEXIO_I2C Driver

- status_t FLEXIO_DRV_InitDevice (uint32_t instance, flexio_device_state_t ∗deviceState)

    Initializes the FlexIO device.
- status_t FLEXIO_DRV_DeinitDevice (uint32_t instance)

    De-initializes the FlexIO device.
- status_t FLEXIO_DRV_Reset (uint32_t instance)

    Resets the FlexIO device.

16.26.2 Enumeration Type Documentation

16.26.2.1 enum flexio_driver_type_t

Driver type: interrupts/polling/DMA Implements : flexio_driver_type_t_Class.

Enumerator

    FLEXIO_DRIVER_TYPE_INTERRUPTS   Driver uses interrupts for data transfers

    FLEXIO_DRIVER_TYPE_POLLING   Driver is based on polling

    FLEXIO_DRIVER_TYPE_DMA   Driver uses DMA for data transfers

Definition at line 49 of file flexio.h.

16.26.3 Function Documentation

16.26.3.1 status_t FLEXIO_DRV_DeinitDevice (uint32_t *instance*)

De-initializes the FlexIO device.

This function de-initializes the FlexIO device.

Parameters

instance	FLEXIO peripheral instance number

Returns

> Error or success status returned by API

Definition at line 128 of file flexio_common.c.

16.26.3.2   status_t FLEXIO_DRV_InitDevice (uint32_t *instance,* flexio_device_state_t ∗ *deviceState*)

Initializes the FlexIO device.

This function resets the FlexIO device, enables interrupts in interrupt manager and enables the device.

Parameters

instance	FLEXIO peripheral instance number
deviceState	Pointer to the FLEXIO device context structure. The driver uses this memory area for its internal logic. The application must make no assumptions about the content of this structure, and must not free this memory until the device is de-initialized using FLEXIO_DRV_Deinit↩ Device().

Returns

> Error or success status returned by API

Definition at line 89 of file flexio_common.c.

16.26.3.3   status_t FLEXIO_DRV_Reset (uint32_t *instance*)

Resets the FlexIO device.

This function resets the FlexIO device.

Parameters

instance	FLEXIO peripheral instance number

Returns

> Error or success status returned by API

Definition at line 153 of file flexio_common.c.

16.27   FlexIO I2C Driver

16.27.1   Detailed Description

I2C communication over FlexIO module (FLEXIO_I2C)

The FLEXIO_I2C Driver allows communication on an I2C bus using the FlexIO module in the S32144K processor.

Features

- Master operation only

- Interrupt, DMA or polling mode

- Provides blocking and non-blocking transmit and receive functions

- 7-bit addressing

- Clock stretching

- Configurable baud rate

Functionality

Before using any Flexio driver the device must first be initialized using function FLEXIO_DRV_InitDevice. Then the FLEXIO_I2C Driver must be initialized using functions FLEXIO_I2C_DRV_MasterInit(). It is possible to use more driver instances on the same FlexIO device, as long as sufficient resources are available. Different driver instances on the same FlexIO device can function independently of each other. When it is no longer needed, the driver can be de-initialized, using FLEXIO_I2C_DRV_MasterDeinit(). This will release the hardware resources, allowing other driver instances to be initialized.

Master Mode

Master Mode provides functions for transmitting or receiving data to/from any I2C slave. Slave address and baud rate are provided at initialization time through the master configuration structure, but they can be changed at runtime by using FLEXIO_I2C_DRV_MasterSetBaudRate() or FLEXIO_I2C_DRV_MasterSetSlaveAddr(). Note that due to module limitation not any baud rate can be achieved. The driver will set a baud rate as close as possible to the requested baud rate, but there may still be substantial differences, for example if requesting a high baud rate while using a low-frequency FlexIO clock. The application should call FLEXIO_I2C_DRV_MasterGetBaudRate() after FLEXIO_I2C_DRV_MasterSetBaudRate() to check what baud rate was actually set.

To send or receive data to/from the currently configured slave address, use functions FLEXIO_I2C_DRV_Master$\hookleftarrow$ SendData() or FLEXIO_I2C_DRV_MasterReceiveData() (or their blocking counterparts). Parameter `sendStop` can be used to chain multiple transfers with repeated START condition between them, for example when sending a command and then immediately receiving a response. The application should ensure that any send or receive transfer with `sendStop` set to `false` is followed by another transfer. The last transfer from a chain should always have `sendStop` set to `true`. This driver does not support continuous send/receive using a user callback function. The callback function is only used to signal the end of a transfer.

Blocking operations will return only when the transfer is completed, either successfully or with error. Non-blocking operations will initiate the transfer and return STATUS_SUCCESS, but the module is still busy with the transfer and another transfer can't be initiated until the current transfer is complete. The application will be notified through the user callback when the transfer completes, or it can check the status of the current transfer by calling FLEXIO_I2$\hookleftarrow$ C_DRV_MasterGetStatus(). If the transfer is still ongoing this function will return STATUS_BUSY. If the transfer is completed, the function will return either STATUS_SUCCESS or an error code, depending on the outcome of the last transfer.

The driver supports interrupt, DMA and polling mode. In polling mode the function FLEXIO_I2C_DRV_Master$\hookleftarrow$ GetStatus() ensures the progress of the transfer by checking and handling transmit and receive events reported by the FlexIO module. The application should ensure that this function is called often enough (at least once per transferred byte) to avoid Tx underflows or Rx overflows. In DMA mode the DMA channels that will be used by the driver are received through the configuration structure. The channels must be initialized by the application before the flexio_i2c driver is initialized. The flexio_i2c driver will only set the DMA request source.

Important Notes

- Before using the FLEXIO_I2C Driver the FlexIO clock must be configured. Refer to Clock Manager for clock configuration.

- Before using the FLEXIO_I2C Driver the pins must be routed to the FlexIO module. Refer to PINS Driver for pin routing configuration. Note that any of the available FlexIO pins can be used for SDA and SCL (configurable at initialization time).

- The driver enables the interrupts for the corresponding FlexIO module, but any interrupt priority setting must be done by the application.

- Aborting a transfer with the function FLEXIO_I2C_DRV_MasterTransferAbort() can't generally be done safely due to device limitation; there is no way to know the exact stage of the transfer, and if we disable the module during the ACK bit (transmit) or during a 0 data bit (receive) the slave will hold the SDA line low forever and block the I2C bus. Therefore this function should only be used in extreme circumstances, and the application must have a way to reset the I2C slave. NACK reception is the only exception, as there is no slave to hold the line low, so in this case the driver will automatically abort the transfer.

- The module can handle clock stretching done by the slave, but will not do clock stretching when the application does not provide data fast enough, so Tx underflows and Rx overflows are possible. This can be an issue especially in polling mode if the function FLEXIO_I2C_DRV_MasterGetStatus() is not called often enough.

- Due to device limitations it is not always possible to tell the difference between NACK reception and receiver overflow. When in doubt, the driver will treat these events as overflow and continue the transfer, in order to avoid the risk of blocking the i2c bus.

- Due to device limitations there is a maximum limit of 13 bytes (FLEXIO_I2C_MAX_SIZE) on the size of any transfer.

- The driver does not support multi-master mode. It does not detect arbitration loss condition.

- Timeout feature for blocking transfers does not work in polling mode.

- This driver needs two shifters and two timers for its operation. Initialization will fail if there are not enough shifters and timers available on the FlexIO device.

- This driver needs two DMA channels for its operation when it is initialized in DMA mode. The DMA channels must be initialized by the application before initializing the driver. Refer to EDMA driver for DMA channels initialization.

- If the application uses an RTOS, this driver uses a semaphore for blocking transfers. Initialization will fail if the semaphore cannot be created. If the driver uses polling mode no semaphore is used.

- If the application uses an RTOS, the FlexIO drivers use a mutex for channel allocation. Only one mutex per device is needed, not per driver instance. Device initialization will fail if the mutex cannot be created.

Data Structures

- struct flexio_i2c_master_user_config_t

    Master configuration structure. More...

- struct flexio_i2c_master_state_t

    Master internal context structure. More...

Macros

- #define FLEXIO_I2C_MAX_SIZE (((uint32_t)((0xFFU - 1U) / 18U)) - 1U)

    Maximum size of a transfer. The restriction is that the total number of SCL edges must not exceed 8 bits, such that it can be programmed in the upper part of the timer compare register. There are 2 SCL edges per bit, 9 bits per byte (including ACK). The extra 1 is for the STOP condition.

FLEXIO_I2C Driver

- status_t FLEXIO_I2C_DRV_MasterInit (uint32_t instance, const flexio_i2c_master_user_config_t ∗user↩
  ConfigPtr, flexio_i2c_master_state_t ∗master)

    Initialize the FLEXIO_I2C master mode driver.

- status_t FLEXIO_I2C_DRV_MasterDeinit (flexio_i2c_master_state_t ∗master)

    De-initialize the FLEXIO_I2C master mode driver.

- status_t FLEXIO_I2C_DRV_MasterSetBaudRate (flexio_i2c_master_state_t ∗master, uint32_t baudRate)

    Set the baud rate for any subsequent I2C communication.

- status_t FLEXIO_I2C_DRV_MasterGetBaudRate (flexio_i2c_master_state_t ∗master, uint32_t ∗baudRate)

    Get the currently configured baud rate.

- status_t FLEXIO_I2C_DRV_MasterSetSlaveAddr (flexio_i2c_master_state_t ∗master, const uint16_t ad-
  dress)

    Set the slave address for any subsequent I2C communication.

- status_t FLEXIO_I2C_DRV_MasterSendData (flexio_i2c_master_state_t ∗master, const uint8_t ∗txBuff,
  uint32_t txSize, bool sendStop)

    Perform a non-blocking send transaction on the I2C bus.

- status_t FLEXIO_I2C_DRV_MasterSendDataBlocking (flexio_i2c_master_state_t ∗master, const uint8↩
  t ∗txBuff, uint32_t txSize, bool sendStop, uint32_t timeout)

    Perform a blocking send transaction on the I2C bus.

- status_t FLEXIO_I2C_DRV_MasterReceiveData (flexio_i2c_master_state_t ∗master, uint8_t ∗rxBuff,
  uint32_t rxSize, bool sendStop)

    Perform a non-blocking receive transaction on the I2C bus.

- status_t FLEXIO_I2C_DRV_MasterReceiveDataBlocking (flexio_i2c_master_state_t ∗master, uint8_t ∗rx↩
  Buff, uint32_t rxSize, bool sendStop, uint32_t timeout)

    Perform a blocking receive transaction on the I2C bus.

- status_t FLEXIO_I2C_DRV_MasterTransferAbort (flexio_i2c_master_state_t ∗master)

    Aborts a non-blocking I2C master transaction.

- status_t FLEXIO_I2C_DRV_MasterGetStatus (flexio_i2c_master_state_t ∗master, uint32_t ∗bytes↩
  Remaining)

    Get the status of the current non-blocking I2C master transaction.

16.27.2   Data Structure Documentation

16.27.2.1   struct flexio_i2c_master_user_config_t

Master configuration structure.

This structure is used to provide configuration parameters for the flexio_i2c master at initialization time. Implements : flexio_i2c_master_user_config_t_Class

Definition at line 93 of file flexio_i2c_driver.h.

Data Fields

- uint16_t slaveAddress
- flexio_driver_type_t driverType
- uint32_t baudRate
- uint8_t sdaPin
- uint8_t sclPin
- i2c_master_callback_t callback
- void ∗ callbackParam
- uint8_t rxDMAChannel
- uint8_t txDMAChannel

Field Documentation

16.27.2.1.1   uint32_t baudRate

Baud rate in hertz

Definition at line 97 of file flexio_i2c_driver.h.

16.27.2.1.2   i2c_master_callback_t callback

User callback function. Note that this function will be called from the interrupt service routine, so its execution time should be as small as possible. It can be NULL if it is not needed

Definition at line 100 of file flexio_i2c_driver.h.

16.27.2.1.3   void∗ callbackParam

Parameter for the callback function

Definition at line 104 of file flexio_i2c_driver.h.

16.27.2.1.4   flexio_driver_type_t driverType

Driver type: interrupts/polling/DMA

Definition at line 96 of file flexio_i2c_driver.h.

16.27.2.1.5   uint8_t rxDMAChannel

Rx DMA channel number. Only used in DMA mode

Definition at line 105 of file flexio_i2c_driver.h.

16.27.2.1.6   uint8_t sclPin

Flexio pin to use as I2C SCL pin

Definition at line 99 of file flexio_i2c_driver.h.

16.27.2.1.7   uint8_t sdaPin

Flexio pin to use as I2C SDA pin

Definition at line 98 of file flexio_i2c_driver.h.

16.27.2.1.8   uint16_t slaveAddress

Slave address, 7-bit

Definition at line 95 of file flexio_i2c_driver.h.

16.27.2.1.9   uint8_t txDMAChannel

Tx DMA channel number. Only used in DMA mode

Definition at line 106 of file flexio_i2c_driver.h.

16.27.2.2   struct flexio_i2c_master_state_t

Master internal context structure.

This structure is used by the driver for its internal logic. It must be provided by the application through the FLEXI↩O_I2C_DRV_MasterInit() function, then it cannot be freed until the driver is de-initialized using FLEXIO_I2C_DR↩V_MasterDeinit(). The application should make no assumptions about the content of this structure.

Definition at line 118 of file flexio_i2c_driver.h.

16.27.3   Macro Definition Documentation

16.27.3.1   #define FLEXIO_I2C_MAX_SIZE (((uint32_t)((0xFFU - 1U) / 18U)) - 1U)

Maximum size of a transfer. The restriction is that the total number of SCL edges must not exceed 8 bits, such that it can be programmed in the upper part of the timer compare register. There are 2 SCL edges per bit, 9 bits per byte (including ACK). The extra 1 is for the STOP condition.

Definition at line 68 of file flexio_i2c_driver.h.

16.27.4   Function Documentation

16.27.4.1   status_t FLEXIO_I2C_DRV_MasterDeinit (flexio_i2c_master_state_t ∗ master)

De-initialize the FLEXIO_I2C master mode driver.

This function de-initializes the FLEXIO_I2C driver in master mode. The driver can't be used again until reinitialized. The context structure is no longer needed by the driver and can be freed after calling this function.

Parameters

master	Pointer to the FLEXIO_I2C master driver context structure.

Returns

Error or success status returned by API

Definition at line 1205 of file flexio_i2c_driver.c.

16.27.4.2   status_t FLEXIO_I2C_DRV_MasterGetBaudRate (flexio_i2c_master_state_t ∗ master, uint32_t ∗ baudRate)

Get the currently configured baud rate.

This function returns the currently configured I2C baud rate.

Parameters

master	Pointer to the FLEXIO_I2C master driver context structure.
baudRate	the current baud rate in hertz

Returns

Error or success status returned by API

Definition at line 1269 of file flexio_i2c_driver.c.

16.27.4.3   status_t FLEXIO_I2C_DRV_MasterGetStatus (flexio_i2c_master_state_t ∗ master, uint32_t ∗ bytesRemaining)

Get the status of the current non-blocking I2C master transaction.

This function returns the current status of a non-blocking I2C master transaction. A return code of STATUS_BUSY means the transfer is still in progress. Otherwise the function returns a status reflecting the outcome of the last transfer. When the driver is initialized in polling mode this function also advances the transfer by checking and handling the transmit and receive events, so it must be called frequently to avoid overflows or underflows.

Parameters

master	Pointer to the FLEXIO_I2C master driver context structure.

bytesRemaining	The remaining number of bytes to be transferred

Returns

Error or success status returned by API

Definition at line 1468 of file flexio_i2c_driver.c.

16.27.4.4   status_t FLEXIO_I2C_DRV_MasterInit (uint32_t *instance,* const **flexio_i2c_master_user_config_t *∗ userConfigPtr,* **flexio_i2c_master_state_t** *∗ master*)**

Initialize the FLEXIO_I2C master mode driver.

This function initializes the FLEXIO_I2C driver in master mode.

Parameters

instance	FLEXIO peripheral instance number
userConfigPtr	Pointer to the FLEXIO_I2C master user configuration structure. The function reads configuration data from this structure and initializes the driver accordingly. The application may free this structure after the function returns.
master	Pointer to the FLEXIO_I2C master driver context structure. The driver uses this memory area for its internal logic. The application must make no assumptions about the content of this structure, and must not free this memory until the driver is de-initialized using FLEXIO_I2↩C_DRV_MasterDeinit().

Returns

Error or success status returned by API

Definition at line 1114 of file flexio_i2c_driver.c.

16.27.4.5   status_t FLEXIO_I2C_DRV_MasterReceiveData (flexio_i2c_master_state_t *∗ master,* uint8_t *∗ rxBuff,* uint32_t *rxSize,* bool *sendStop*)

Perform a non-blocking receive transaction on the I2C bus.

This function starts the reception of a block of data from the currently configured slave address and returns immediately. The rest of the transmission is handled by the interrupt service routine (if the driver is initialized in interrupt mode) or by the FLEXIO_I2C_DRV_MasterGetStatus function (if the driver is initialized in polling mode). Use FLEXIO_I2C_DRV_MasterGetStatus() to check the progress of the reception.

Parameters

master	Pointer to the FLEXIO_I2C master driver context structure.
rxBuff	pointer to the buffer where to store received data
rxSize	length in bytes of the data to be transferred
sendStop	specifies whether or not to generate stop condition after the reception

Returns

Error or success status returned by API

Definition at line 1384 of file flexio_i2c_driver.c.

16.27.4.6   status_t FLEXIO_I2C_DRV_MasterReceiveDataBlocking (flexio_i2c_master_state_t *∗ master,* uint8_t *∗ rxBuff,* uint32_t *rxSize,* bool *sendStop,* uint32_t *timeout*)

Perform a blocking receive transaction on the I2C bus.

This function receives a block of data from the currently configured slave address, and only returns when the transmission is complete.

Parameters

master	Pointer to the FLEXIO_I2C master driver context structure.
rxBuff	pointer to the buffer where to store received data
rxSize	length in bytes of the data to be transferred
sendStop	specifies whether or not to generate stop condition after the reception
timeout	timeout for the transfer in milliseconds

Returns

Error or success status returned by API

Definition at line 1404 of file flexio_i2c_driver.c.

16.27.4.7 status_t FLEXIO_I2C_DRV_MasterSendData (flexio_i2c_master_state_t ∗ master, const uint8_t ∗ txBuff, uint32_t txSize, bool sendStop)

Perform a non-blocking send transaction on the I2C bus.

This function starts the transmission of a block of data to the currently configured slave address and returns immediately. The rest of the transmission is handled by the interrupt service routine (if the driver is initialized in interrupt mode) or by the FLEXIO_I2C_DRV_MasterGetStatus function (if the driver is initialized in polling mode). Use FLEXIO_I2C_DRV_MasterGetStatus() to check the progress of the transmission.

Parameters

master	Pointer to the FLEXIO_I2C master driver context structure.
txBuff	pointer to the data to be transferred
txSize	length in bytes of the data to be transferred
sendStop	specifies whether or not to generate stop condition after the transmission

Returns

Error or success status returned by API

Definition at line 1326 of file flexio_i2c_driver.c.

16.27.4.8 status_t FLEXIO_I2C_DRV_MasterSendDataBlocking (flexio_i2c_master_state_t ∗ master, const uint8_t ∗ txBuff, uint32_t txSize, bool sendStop, uint32_t timeout)

Perform a blocking send transaction on the I2C bus.

This function sends a block of data to the currently configured slave address, and only returns when the transmission is complete.

Parameters

master	Pointer to the FLEXIO_I2C master driver context structure.
txBuff	pointer to the data to be transferred
txSize	length in bytes of the data to be transferred
sendStop	specifies whether or not to generate stop condition after the transmission
timeout	timeout for the transfer in milliseconds

Returns

Error or success status returned by API

Definition at line 1346 of file flexio_i2c_driver.c.

16.27.4.9 status_t FLEXIO_I2C_DRV_MasterSetBaudRate (flexio_i2c_master_state_t ∗ master, uint32_t baudRate)

Set the baud rate for any subsequent I2C communication.

This function sets the baud rate (SCL frequency) for the I2C master. Note that due to module limitation not any baud rate can be achieved. The driver will set a baud rate as close as possible to the requested baud rate, but there may still be substantial differences, for example if requesting a high baud rate while using a low-frequency FlexIO clock. The application should call FLEXIO_I2C_DRV_MasterGetBaudRate() after FLEXIO_I2C_DRV_MasterSet↩BaudRate() to check what baud rate was actually set.

Parameters

master	Pointer to the FLEXIO_I2C master driver context structure.
baudRate	the desired baud rate in hertz

Returns

> Error or success status returned by API

Definition at line 1228 of file flexio_i2c_driver.c.

16.27.4.10   status_t FLEXIO_I2C_DRV_MasterSetSlaveAddr (flexio_i2c_master_state_t ∗ *master,* const uint16_t *address*)

Set the slave address for any subsequent I2C communication.

This function sets the slave address which will be used for any future transfer.

Parameters

master	Pointer to the FLEXIO_I2C master driver context structure.
address	slave address, 7-bit

Returns

> Error or success status returned by API

Definition at line 1307 of file flexio_i2c_driver.c.

16.27.4.11   status_t FLEXIO_I2C_DRV_MasterTransferAbort (flexio_i2c_master_state_t ∗ *master*)

Aborts a non-blocking I2C master transaction.

This function aborts a non-blocking I2C transfer.

Parameters

master	Pointer to the FLEXIO_I2C master driver context structure.

Returns

> Error or success status returned by API

Definition at line 1442 of file flexio_i2c_driver.c.

16.28 FlexIO I2S Driver

16.28.1 Detailed Description

I2S communication over FlexIO module (FLEXIO_I2S)

The FLEXIO_I2S Driver allows communication on an I2S bus using the FlexIO module in the S32144K processor.

Features

- Master or slave operation

- Interrupt, DMA or polling mode

- Provides blocking and non-blocking transmit and receive functions

- Configurable baud rate and bit count

Functionality

Before using any Flexio driver the device must first be initialized using function FLEXIO_DRV_InitDevice. Then the FLEXIO_I2S Driver must be initialized, using functions FLEXIO_I2S_DRV_MasterInit() or FLEXIO_I2S_DRV↩ _SlaveInit(). It is possible to use more driver instances on the same FlexIO device, as long as sufficient resources are available. Different driver instances on the same FlexIO device can function independently of each other. When it is no longer needed, the driver can be de-initialized, using FLEXIO_I2S_DRV_MasterDeinit() or FLEXIO_I2S_↩ DRV_SlaveDeinit(). This will release the hardware resources, allowing other driver instances to be initialized.

Master Mode

Master Mode provides functions for transmitting or receiving data to/from any I2S slave. The number of bits per word and the baud rate are provided at initialization time through the master configuration structure, but they can be changed at runtime by using FLEXIO_I2S_DRV_MasterSetConfig(). Note that due to module limitation not any baud rate can be achieved. The driver will set a baud rate as close as possible to the requested baud rate, but there may still be substantial differences, for example if requesting a high baud rate while using a low-frequency FlexIO clock. The application should call FLEXIO_I2S_DRV_MasterGetBaudRate() to check what baud rate was actually set.

To send or receive data to/from the currently configured slave address, use functions FLEXIO_I2S_DRV_Master↩ SendData() or FLEXIO_I2S_DRV_MasterReceiveData() (or their blocking counterparts). The driver is not full-duplex, only one direction (send or receive) can be used at one time. It is possible to configure both Rx and Tx pin to use the same Flexio pin.

Continuous send/receive can be realized by registering a user callback function. When the driver completes the transmission or reception of the current buffer, it will invoke the user callback with an appropriate event. The callback function can the use FLEXIO_I2S_DRV_MasterSetTxBuffer() or FLEXIO_I2S_DRV_MasterSetRxBuffer() to provide a new buffer.

Blocking operations will return only when the transfer is completed, either successfully or with error. Non-blocking operations will initiate the transfer and return STATUS_SUCCESS, but the module is still busy with the transfer and another transfer can't be initiated until the current transfer is complete. The application will be notified through the user callback when the transfer completes, or it can check the status of the current transfer by calling FLEXIO_I2↩ S_DRV_MasterGetStatus(). If the transfer is still ongoing this function will return STATUS_BUSY. If the transfer is completed, the function will return either STATUS_SUCCESS or an error code, depending on the outcome of the last transfer.

The driver supports interrupt, DMA and polling mode. In polling mode the function FLEXIO_I2S_DRV_Master↩ GetStatus() ensures the progress of the transfer by checking and handling transmit and receive events reported by the FlexIO module. The application should ensure that this function is called often enough (at least once per transferred byte) to avoid Tx underflows or Rx overflows. In DMA mode the DMA channels that will be used by the driver are received through the configuration structure. The channels must be initialized by the application before the flexio_i2s driver is initialized. The flexio_i2s driver will only set the DMA request source.

Slave Mode

Slave Mode is very similar to master mode, the main difference being that the FLEXIO_I2S_DRV_SlaveInit() function initializes the FlexIO module to use the clock signal received from the master instead of generating it. Consequently, there is no baud rate setting in slave mode. Other than that, the slave mode offers a similar interface to the master mode. FLEXIO_I2S_DRV_MasterSendData() or FLEXIO_I2S_DRV_MasterReceiveData() (or their blocking counterparts) can be used to initiate transfers, and FLEXIO_I2S_DRV_SlaveGetStatus() is used to check the status of the transfer and advance the transfer in polling mode. All other specifications from the Master Mode description apply for Slave Mode too.

Important Notes

- Before using the FLEXIO_I2S Driver the FlexIO clock must be configured. Refer to Clock Manager for clock configuration.

- Before using the FLEXIO_I2S Driver the pins must be routed to the FlexIO module. Refer to PINS Driver for pin routing configuration. Note that any of the available FlexIO pins can be used for any of the TX, RX, SCK and WS signals (configurable at initialization time). If more than one driver instance is used on the same Flexio module, it is the responsibility of the application to ensure there are no conflicts between pins.

- The driver enables the interrupts for the corresponding FlexIO module, but any interrupt priority setting must be done by the application.

- Timeout feature for blocking transfers does not work in polling mode.

- This driver needs two shifters and two timers for its operation. Initialization will fail if there are not enough shifters and timers available on the FlexIO device.

- This driver needs two DMA channels for its operation when it is initialized in DMA mode. The DMA channels must be initialized by the application before initializing the driver. Refer to EDMA driver for DMA channels initialization.

- If the application uses an RTOS, this driver uses a semaphore for blocking transfers. Initialization will fail if the semaphore cannot be created. If the driver uses polling mode no semaphore is used.

- If the application uses an RTOS, the FlexIO drivers use a mutex for channel allocation. Only one mutex per device is needed, not per driver instance. Device initialization will fail if the mutex cannot be created.

Data Structures

- struct flexio_i2s_master_user_config_t

    Master configuration structure. More...
- struct flexio_i2s_slave_user_config_t

    Slave configuration structure. More...
- struct flexio_i2s_master_state_t

    Master internal context structure. More...

Typedefs

- typedef flexio_i2s_master_state_t flexio_i2s_slave_state_t

    Slave internal context structure.

FLEXIO_I2S Driver

- status_t FLEXIO_I2S_DRV_MasterInit (uint32_t instance, const flexio_i2s_master_user_config_t ∗user↩
ConfigPtr, flexio_i2s_master_state_t ∗master)

    Initialize the FLEXIO_I2S master mode driver.
- status_t FLEXIO_I2S_DRV_MasterDeinit (flexio_i2s_master_state_t ∗master)

De-initialize the FLEXIO_I2S master mode driver.

- status_t FLEXIO_I2S_DRV_MasterSetConfig (flexio_i2s_master_state_t *master, uint32_t baudRate, uint8_t bitsWidth)

    Set the baud rate and bit width for any subsequent I2S communication.

- status_t FLEXIO_I2S_DRV_MasterGetBaudRate (flexio_i2s_master_state_t *master, uint32_t *baudRate)

    Get the currently configured baud rate.

- status_t FLEXIO_I2S_DRV_MasterSendData (flexio_i2s_master_state_t *master, const uint8_t *txBuff, uint32_t txSize)

    Perform a non-blocking send transaction on the I2S bus.

- status_t FLEXIO_I2S_DRV_MasterSendDataBlocking (flexio_i2s_master_state_t *master, const uint8_↩ t *txBuff, uint32_t txSize, uint32_t timeout)

    Perform a blocking send transaction on the I2S bus.

- status_t FLEXIO_I2S_DRV_MasterReceiveData (flexio_i2s_master_state_t *master, uint8_t *rxBuff, uint32_t rxSize)

    Perform a non-blocking receive transaction on the I2S bus.

- status_t FLEXIO_I2S_DRV_MasterReceiveDataBlocking (flexio_i2s_master_state_t *master, uint8_t *rx↩ Buff, uint32_t rxSize, uint32_t timeout)

    Perform a blocking receive transaction on the I2S bus.

- status_t FLEXIO_I2S_DRV_MasterTransferAbort (flexio_i2s_master_state_t *master)

    Aborts a non-blocking I2S master transaction.

- status_t FLEXIO_I2S_DRV_MasterGetStatus (flexio_i2s_master_state_t *master, uint32_t *bytes↩ Remaining)

    Get the status of the current non-blocking I2S master transaction.

- status_t FLEXIO_I2S_DRV_MasterSetRxBuffer (flexio_i2s_master_state_t *master, uint8_t *rxBuff, uint32↩ _t rxSize)

    Provide a buffer for receiving data.

- status_t FLEXIO_I2S_DRV_MasterSetTxBuffer (flexio_i2s_master_state_t *master, const uint8_t *txBuff, uint32_t txSize)

    Provide a buffer for transmitting data.

- status_t FLEXIO_I2S_DRV_SlaveInit (uint32_t instance, const flexio_i2s_slave_user_config_t *userConfig↩ Ptr, flexio_i2s_slave_state_t *slave)

    Initialize the FLEXIO_I2S slave mode driver.

- static status_t FLEXIO_I2S_DRV_SlaveDeinit (flexio_i2s_slave_state_t *slave)

    De-initialize the FLEXIO_I2S slave mode driver.

- status_t FLEXIO_I2S_DRV_SlaveSetConfig (flexio_i2s_slave_state_t *slave, uint8_t bitsWidth)

    Set the bit width for any subsequent I2S communication.

- static status_t FLEXIO_I2S_DRV_SlaveSendData (flexio_i2s_slave_state_t *slave, const uint8_t *txBuff, uint32_t txSize)

    Perform a non-blocking send transaction on the I2S bus.

- static status_t FLEXIO_I2S_DRV_SlaveSendDataBlocking (flexio_i2s_slave_state_t *slave, const uint8_↩ t *txBuff, uint32_t txSize, uint32_t timeout)

    Perform a blocking send transaction on the I2S bus.

- static status_t FLEXIO_I2S_DRV_SlaveReceiveData (flexio_i2s_slave_state_t *slave, uint8_t *rxBuff, uint32_t rxSize)

    Perform a non-blocking receive transaction on the I2S bus.

- static status_t FLEXIO_I2S_DRV_SlaveReceiveDataBlocking (flexio_i2s_slave_state_t *slave, uint8_t *rx↩ Buff, uint32_t rxSize, uint32_t timeout)

    Perform a blocking receive transaction on the I2S bus.

- static status_t FLEXIO_I2S_DRV_SlaveTransferAbort (flexio_i2s_slave_state_t *slave)

    Aborts a non-blocking I2S slave transaction.

- static status_t FLEXIO_I2S_DRV_SlaveGetStatus (flexio_i2s_slave_state_t *slave, uint32_t *bytes↩ Remaining)

Get the status of the current non-blocking I2S slave transaction.

- static status_t FLEXIO_I2S_DRV_SlaveSetRxBuffer (flexio_i2s_slave_state_t ∗slave, uint8_t ∗rxBuff, uint32_t rxSize)

    Provide a buffer for receiving data.

- static status_t FLEXIO_I2S_DRV_SlaveSetTxBuffer (flexio_i2s_slave_state_t ∗slave, const uint8_t ∗txBuff, uint32_t txSize)

    Provide a buffer for transmitting data.

16.28.2 Data Structure Documentation

16.28.2.1 struct flexio_i2s_master_user_config_t

Master configuration structure.

This structure is used to provide configuration parameters for the flexio_i2s master at initialization time. Implements : flexio_i2s_master_user_config_t_Class

Definition at line 70 of file flexio_i2s_driver.h.

Data Fields

- flexio_driver_type_t driverType
- uint32_t baudRate
- uint8_t bitsWidth
- uint8_t txPin
- uint8_t rxPin
- uint8_t sckPin
- uint8_t wsPin
- i2s_callback_t callback
- void ∗ callbackParam
- uint8_t rxDMAChannel
- uint8_t txDMAChannel

Field Documentation

16.28.2.1.1 uint32_t baudRate

Baud rate in hertz

Definition at line 73 of file flexio_i2s_driver.h.

16.28.2.1.2 uint8_t bitsWidth

Number of bits in a word - multiple of 8

Definition at line 74 of file flexio_i2s_driver.h.

16.28.2.1.3 i2s_callback_t callback

User callback function. Note that this function will be called from the interrupt service routine, so its execution time should be as small as possible. It can be NULL if it is not needed

Definition at line 79 of file flexio_i2s_driver.h.

16.28.2.1.4 void∗ callbackParam

Parameter for the callback function

Definition at line 83 of file flexio_i2s_driver.h.

16.28.2.1.5   flexio_driver_type_t driverType

Driver type: interrupts/polling/DMA

Definition at line 72 of file flexio_i2s_driver.h.

16.28.2.1.6   uint8_t rxDMAChannel

Rx DMA channel number. Only used in DMA mode

Definition at line 84 of file flexio_i2s_driver.h.

16.28.2.1.7   uint8_t rxPin

Flexio pin to use for receive

Definition at line 76 of file flexio_i2s_driver.h.

16.28.2.1.8   uint8_t sckPin

Flexio pin to use for serial clock

Definition at line 77 of file flexio_i2s_driver.h.

16.28.2.1.9   uint8_t txDMAChannel

Tx DMA channel number. Only used in DMA mode

Definition at line 85 of file flexio_i2s_driver.h.

16.28.2.1.10   uint8_t txPin

Flexio pin to use for transmit

Definition at line 75 of file flexio_i2s_driver.h.

16.28.2.1.11   uint8_t wsPin

Flexio pin to use for word select

Definition at line 78 of file flexio_i2s_driver.h.

16.28.2.2   struct flexio_i2s_slave_user_config_t

Slave configuration structure.

This structure is used to provide configuration parameters for the flexio_i2s slave at initialization time. Implements : flexio_i2s_slave_user_config_t_Class

Definition at line 95 of file flexio_i2s_driver.h.

Data Fields

- flexio_driver_type_t driverType
- uint8_t bitsWidth
- uint8_t txPin
- uint8_t rxPin
- uint8_t sckPin
- uint8_t wsPin
- i2s_callback_t callback
- void ∗ callbackParam
- uint8_t rxDMAChannel
- uint8_t txDMAChannel

Field Documentation

16.28.2.2.1 uint8_t bitsWidth

Number of bits in a word - multiple of 8

Definition at line 98 of file flexio_i2s_driver.h.

16.28.2.2.2 i2s_callback_t callback

User callback function. Note that this function will be called from the interrupt service routine, so its execution time should be as small as possible. It can be NULL if it is not needed

Definition at line 103 of file flexio_i2s_driver.h.

16.28.2.2.3 void∗ callbackParam

Parameter for the callback function

Definition at line 107 of file flexio_i2s_driver.h.

16.28.2.2.4 flexio_driver_type_t driverType

Driver type: interrupts/polling/DMA

Definition at line 97 of file flexio_i2s_driver.h.

16.28.2.2.5 uint8_t rxDMAChannel

Rx DMA channel number. Only used in DMA mode

Definition at line 108 of file flexio_i2s_driver.h.

16.28.2.2.6 uint8_t rxPin

Flexio pin to use for receive

Definition at line 100 of file flexio_i2s_driver.h.

16.28.2.2.7 uint8_t sckPin

Flexio pin to use for serial clock

Definition at line 101 of file flexio_i2s_driver.h.

16.28.2.2.8 uint8_t txDMAChannel

Tx DMA channel number. Only used in DMA mode

Definition at line 109 of file flexio_i2s_driver.h.

16.28.2.2.9 uint8_t txPin

Flexio pin to use for transmit

Definition at line 99 of file flexio_i2s_driver.h.

16.28.2.2.10 uint8_t wsPin

Flexio pin to use for word select

Definition at line 102 of file flexio_i2s_driver.h.

16.28.2.3 struct flexio_i2s_master_state_t

Master internal context structure.

This structure is used by the driver for its internal logic. It must be provided by the application through the FLEXI↩
O_I2S_DRV_MasterInit() function, then it cannot be freed until the driver is de-initialized using FLEXIO_I2S_DR↩
V_MasterDeinit(). The application should make no assumptions about the content of this structure.

Definition at line 121 of file flexio_i2s_driver.h.

16.28.3   Typedef Documentation

16.28.3.1   typedef **flexio_i2s_master_state_t flexio_i2s_slave_state_t**

Slave internal context structure.

This structure is used by the driver for its internal logic. It must be provided by the application through the FLEXI↩
O_I2S_DRV_SlaveInit() function, then it cannot be freed until the driver is de-initialized using FLEXIO_I2S_DRV↩
_SlaveDeinit(). The application should make no assumptions about the content of this structure.

Definition at line 153 of file flexio_i2s_driver.h.

16.28.4   Function Documentation

16.28.4.1   status_t FLEXIO_I2S_DRV_MasterDeinit (**flexio_i2s_master_state_t** ∗ *master*)

De-initialize the FLEXIO_I2S master mode driver.

This function de-initializes the FLEXIO_I2S driver in master mode. The driver can't be used again until reinitialized. The context structure is no longer needed by the driver and can be freed after calling this function.

Parameters

master	Pointer to the FLEXIO_I2S master driver context structure.

Returns

    Error or success status returned by API

Definition at line 1032 of file flexio_i2s_driver.c.

16.28.4.2   status_t FLEXIO_I2S_DRV_MasterGetBaudRate (**flexio_i2s_master_state_t** ∗ *master,* uint32_t ∗ *baudRate*)

Get the currently configured baud rate.

This function returns the currently configured I2S baud rate.

Parameters

master	Pointer to the FLEXIO_I2S master driver context structure.
baudRate	the current baud rate in hertz

Returns

    Error or success status returned by API

Definition at line 1105 of file flexio_i2s_driver.c.

16.28.4.3   status_t FLEXIO_I2S_DRV_MasterGetStatus (**flexio_i2s_master_state_t** ∗ *master,* uint32_t ∗ *bytesRemaining*)

Get the status of the current non-blocking I2S master transaction.

This function returns the current status of a non-blocking I2S master transaction. A return code of STATUS_BUSY means the transfer is still in progress. Otherwise the function returns a status reflecting the outcome of the last transfer. When the driver is initialized in polling mode this function also advances the transfer by checking and handling the transmit and receive events, so it must be called frequently to avoid overflows or underflows.

Parameters

master	Pointer to the FLEXIO_I2S master driver context structure.
bytesRemaining	the remaining number of bytes to be transferred

Returns

      Error or success status returned by API

Definition at line 1358 of file flexio_i2s_driver.c.

16.28.4.4 status_t FLEXIO_I2S_DRV_MasterInit (uint32_t *instance,* const flexio_i2s_master_user_config_t ∗ ***userConfigPtr,* flexio_i2s_master_state_t ∗ *master*)**

Initialize the FLEXIO_I2S master mode driver.

This function initializes the FLEXIO_I2S driver in master mode.

Parameters

instance	FLEXIO peripheral instance number
userConfigPtr	Pointer to the FLEXIO_I2S master user configuration structure. The function reads configuration data from this structure and initializes the driver accordingly. The application may free this structure after the function returns.
master	Pointer to the FLEXIO_I2S master driver context structure. The driver uses this memory area for its internal logic. The application must make no assumptions about the content of this structure, and must not free this memory until the driver is de-initialized using FLEXIO_I2S↩ _DRV_MasterDeinit().

Returns

      Error or success status returned by API

Definition at line 941 of file flexio_i2s_driver.c.

16.28.4.5 status_t FLEXIO_I2S_DRV_MasterReceiveData (flexio_i2s_master_state_t ∗ *master,* uint8_t ∗ *rxBuff,* uint32_t ***rxSize*)**

Perform a non-blocking receive transaction on the I2S bus.

This function starts the reception of a block of data and returns immediately. The rest of the reception is handled by the interrupt service routine (if the driver is initialized in interrupt mode) or by the FLEXIO_I2S_DRV_MasterGet↩ Status function (if the driver is initialized in polling mode). Use FLEXIO_I2S_DRV_MasterGetStatus() to check the progress of the reception.

Parameters

master	Pointer to the FLEXIO_I2S master driver context structure.
rxBuff	pointer to the buffer where to store received data
rxSize	length in bytes of the data to be transferred

Returns

      Error or success status returned by API

Definition at line 1241 of file flexio_i2s_driver.c.

16.28.4.6 status_t FLEXIO_I2S_DRV_MasterReceiveDataBlocking (flexio_i2s_master_state_t ∗ *master,* uint8_t ∗ *rxBuff,* ***uint32_t rxSize,* uint32_t *timeout*)**

Perform a blocking receive transaction on the I2S bus.

This function receives a block of data and only returns when the reception is complete.

Parameters

master	Pointer to the FLEXIO_I2S master driver context structure.
rxBuff	pointer to the buffer where to store received data
rxSize	length in bytes of the data to be transferred
timeout	timeout for the transfer in milliseconds

Returns

> Error or success status returned by API

Definition at line 1307 of file flexio_i2s_driver.c.

16.28.4.7   status_t FLEXIO_I2S_DRV_MasterSendData (flexio_i2s_master_state_t * master, const uint8_t * txBuff, uint32_t txSize)

Perform a non-blocking send transaction on the I2S bus.

This function starts the transmission of a block of data and returns immediately. The rest of the transmission is handled by the interrupt service routine (if the driver is initialized in interrupt mode) or by the FLEXIO_I2S_DRV_↩
MasterGetStatus function (if the driver is initialized in polling mode). Use FLEXIO_I2S_DRV_MasterGetStatus() to check the progress of the transmission.

Parameters

master	Pointer to the FLEXIO_I2S master driver context structure.
txBuff	pointer to the data to be transferred
txSize	length in bytes of the data to be transferred

Returns

> Error or success status returned by API

Definition at line 1144 of file flexio_i2s_driver.c.

16.28.4.8   status_t FLEXIO_I2S_DRV_MasterSendDataBlocking (flexio_i2s_master_state_t * master, const uint8_t * txBuff, uint32_t txSize, uint32_t timeout)

Perform a blocking send transaction on the I2S bus.

This function sends a block of data, and only returns when the transmission is complete.

Parameters

master	Pointer to the FLEXIO_I2S master driver context structure.
txBuff	pointer to the data to be transferred
txSize	length in bytes of the data to be transferred
timeout	timeout for the transfer in milliseconds

Returns

> Error or success status returned by API

Definition at line 1208 of file flexio_i2s_driver.c.

16.28.4.9   status_t FLEXIO_I2S_DRV_MasterSetConfig (flexio_i2s_master_state_t * master, uint32_t baudRate, uint8_t bitsWidth)

Set the baud rate and bit width for any subsequent I2S communication.

This function sets the baud rate (SCK frequency) and bit width for the I2S master. Note that due to module limitation not any baud rate can be achieved. The driver will set a baud rate as close as possible to the requested baud rate,

but there may still be substantial differences, for example if requesting a high baud rate while using a low-frequency FlexIO clock. The application should call FLEXIO_I2S_DRV_MasterGetBaudRate() after FLEXIO_I2S_DRV_↩ MasterSetConfig() to check what baud rate was actually set.

Parameters

master	Pointer to the FLEXIO_I2S master driver context structure.
baudRate	the desired baud rate in hertz
bitsWidth	number of bits per word

Returns

Error or success status returned by API

Definition at line 1055 of file flexio_i2s_driver.c.

16.28.4.10   status_t FLEXIO_I2S_DRV_MasterSetRxBuffer (flexio_i2s_master_state_t ∗ *master,* uint8_t ∗ *rxBuff,* uint32_t *rxSize*)

Provide a buffer for receiving data.

This function can be used to provide a new buffer for receiving data to the driver. It can be called from the user callback when event STATUS_I2S_RX_OVERRUN is reported. This way the reception will continue without interruption.

Parameters

master	Pointer to the FLEXIO_I2S master driver context structure.
rxBuff	pointer to the buffer where to store received data
rxSize	length in bytes of the data to be transferred

Returns

Error or success status returned by API

Definition at line 1413 of file flexio_i2s_driver.c.

16.28.4.11   status_t FLEXIO_I2S_DRV_MasterSetTxBuffer (flexio_i2s_master_state_t ∗ *master,* const uint8_t ∗ *txBuff,* uint32_t *txSize*)

Provide a buffer for transmitting data.

This function can be used to provide a new buffer for transmitting data to the driver. It can be called from the user callback when event STATUS_I2S_TX_UNDERRUN is reported. This way the transmission will continue without interruption.

Parameters

master	Pointer to the FLEXIO_I2S master driver context structure.
txBuff	pointer to the buffer containing transmit data
txSize	length in bytes of the data to be transferred

Returns

Error or success status returned by API

Definition at line 1435 of file flexio_i2s_driver.c.

16.28.4.12   status_t FLEXIO_I2S_DRV_MasterTransferAbort (flexio_i2s_master_state_t ∗ *master*)

Aborts a non-blocking I2S master transaction.

This function aborts a non-blocking I2S transfer.

Parameters

master	Pointer to the FLEXIO_I2S master driver context structure.

Returns

> Error or success status returned by API

Definition at line 1340 of file flexio_i2s_driver.c.

16.28.4.13   static status_t FLEXIO_I2S_DRV_SlaveDeinit (flexio_i2s_slave_state_t ∗ slave) `[inline],[static]`

De-initialize the FLEXIO_I2S slave mode driver.

This function de-initializes the FLEXIO_I2S driver in slave mode. The driver can't be used again until reinitialized. The context structure is no longer needed by the driver and can be freed after calling this function.

Parameters

slave	Pointer to the FLEXIO_I2S slave driver context structure.

Returns

> Error or success status returned by API Implements : FLEXIO_I2S_DRV_SlaveDeinit_Activity

Definition at line 403 of file flexio_i2s_driver.h.

16.28.4.14   static status_t FLEXIO_I2S_DRV_SlaveGetStatus (flexio_i2s_slave_state_t ∗ slave, uint32_t ∗ bytesRemaining) `[inline],[static]`

Get the status of the current non-blocking I2S slave transaction.

This function returns the current status of a non-blocking I2S slave transaction. A return code of STATUS_BUSY means the transfer is still in progress. Otherwise the function returns a status reflecting the outcome of the last transfer. When the driver is initialized in polling mode this function also advances the transfer by checking and handling the transmit and receive events, so it must be called frequently to avoid overflows or underflows.

Parameters

slave	Pointer to the FLEXIO_I2S slave driver context structure.
bytesRemaining	the remaining number of bytes to be transferred

Returns

> Error or success status returned by API Implements : FLEXIO_I2S_DRV_SlaveGetStatus_Activity

Definition at line 542 of file flexio_i2s_driver.h.

16.28.4.15   status_t FLEXIO_I2S_DRV_SlaveInit (uint32_t instance, const flexio_i2s_slave_user_config_t ∗ userConfigPtr, flexio_i2s_slave_state_t ∗ slave)

Initialize the FLEXIO_I2S slave mode driver.

This function initializes the FLEXIO_I2S driver in slave mode.

Parameters

instance	FLEXIO peripheral instance number
userConfigPtr	Pointer to the FLEXIO_I2S slave user configuration structure. The function reads configuration data from this structure and initializes the driver accordingly. The application may free this structure after the function returns.
slave	Pointer to the FLEXIO_I2S slave driver context structure. The driver uses this memory area for its internal logic. The application must make no assumptions about the content of this structure, and must not free this memory until the driver is de-initialized using FLEXIO_I2S↩_DRV_SlaveDeinit().

Returns

> Error or success status returned by API

Definition at line 1459 of file flexio_i2s_driver.c.

16.28.4.16   static status_t FLEXIO_I2S_DRV_SlaveReceiveData (flexio_i2s_slave_state_t ∗ *slave,* uint8_t ∗ *rxBuff,* uint32_t *rxSize*) `[inline],[static]`

Perform a non-blocking receive transaction on the I2S bus.

This function starts the reception of a block of data and returns immediately. The rest of the reception is handled by the interrupt service routine (if the driver is initialized in interrupt mode) or by the FLEXIO_I2S_DRV_SlaveGet↩ Status function (if the driver is initialized in polling mode). Use FLEXIO_I2S_DRV_SlaveGetStatus() to check the progress of the reception.

Parameters

slave	Pointer to the FLEXIO_I2S slave driver context structure.
rxBuff	pointer to the buffer where to store received data
rxSize	length in bytes of the data to be transferred

Returns

> Error or success status returned by API Implements : FLEXIO_I2S_DRV_SlaveReceiveData_Activity

Definition at line 483 of file flexio_i2s_driver.h.

16.28.4.17   static status_t FLEXIO_I2S_DRV_SlaveReceiveDataBlocking (flexio_i2s_slave_state_t ∗ *slave,* uint8_t ∗ *rxBuff,* uint32_t *rxSize,* uint32_t *timeout*) `[inline],[static]`

Perform a blocking receive transaction on the I2S bus.

This function receives a block of data and only returns when the reception is complete.

Parameters

slave	Pointer to the FLEXIO_I2S slave driver context structure.
rxBuff	pointer to the buffer where to store received data
rxSize	length in bytes of the data to be transferred
timeout	timeout for the transfer in milliseconds

Returns

> Error or success status returned by API Implements : FLEXIO_I2S_DRV_SlaveReceiveDataBlocking_Activity

Definition at line 503 of file flexio_i2s_driver.h.

16.28.4.18   static status_t FLEXIO_I2S_DRV_SlaveSendData (flexio_i2s_slave_state_t ∗ *slave,* const uint8_t ∗ *txBuff,* uint32_t *txSize*) `[inline],[static]`

Perform a non-blocking send transaction on the I2S bus.

This function starts the transmission of a block of data and returns immediately. The rest of the transmission is handled by the interrupt service routine (if the driver is initialized in interrupt mode) or by the FLEXIO_I2S_DRV↩ _SlaveGetStatus function (if the driver is initialized in polling mode). Use FLEXIO_I2S_DRV_SlaveGetStatus() to check the progress of the transmission.

Parameters

slave	Pointer to the FLEXIO_I2S slave driver context structure.
txBuff	pointer to the data to be transferred
txSize	length in bytes of the data to be transferred

Returns

>   Error or success status returned by API Implements : FLEXIO_I2S_DRV_SlaveSendData_Activity

Definition at line 437 of file flexio_i2s_driver.h.

16.28.4.19   static status_t FLEXIO_I2S_DRV_SlaveSendDataBlocking (flexio_i2s_slave_state_t ∗ *slave,* const uint8_t ∗ *txBuff,* uint32_t *txSize,* uint32_t *timeout*)   `[inline],[static]`

Perform a blocking send transaction on the I2S bus.

This function sends a block of data, and only returns when the transmission is complete.

Parameters

slave	Pointer to the FLEXIO_I2S slave driver context structure.
txBuff	pointer to the data to be transferred
txSize	length in bytes of the data to be transferred
timeout	timeout for the transfer in milliseconds

Returns

>   Error or success status returned by API Implements : FLEXIO_I2S_DRV_SlaveSendDataBlocking_Activity

Definition at line 459 of file flexio_i2s_driver.h.

16.28.4.20   status_t FLEXIO_I2S_DRV_SlaveSetConfig (flexio_i2s_slave_state_t ∗ *slave,* uint8_t *bitsWidth*)

Set the bit width for any subsequent I2S communication.

This function sets the bit width for the I2S slave.

Parameters

slave	Pointer to the FLEXIO_I2S slave driver context structure.
bitsWidth	number of bits per word

Returns

>   Error or success status returned by API

Definition at line 1541 of file flexio_i2s_driver.c.

16.28.4.21   static status_t FLEXIO_I2S_DRV_SlaveSetRxBuffer (flexio_i2s_slave_state_t ∗ *slave,* uint8_t ∗ *rxBuff,* uint32_t *rxSize*)   `[inline],[static]`

Provide a buffer for receiving data.

This function can be used to provide a driver with a new buffer for receiving data. It can be called from the user callback when event STATUS_I2S_RX_OVERRUN is reported. This way the reception will continue without interruption.

Parameters

slave	Pointer to the FLEXIO_I2S slave driver context structure.
rxBuff	pointer to the buffer where to store received data
rxSize	length in bytes of the data to be transferred

Returns

Error or success status returned by API Implements : FLEXIO_I2S_DRV_SlaveSetRxBuffer_Activity

Definition at line 561 of file flexio_i2s_driver.h.

16.28.4.22 static status_t FLEXIO_I2S_DRV_SlaveSetTxBuffer (flexio_i2s_slave_state_t ∗ slave, const uint8_t ∗ txBuff, uint32_t txSize) `[inline],[static]`

Provide a buffer for transmitting data.

This function can be used to provide a driver with a new buffer for transmitting data. It can be called from the user callback when event STATUS_I2S_TX_UNDERRUN is reported. This way the transmission will continue without interruption.

Parameters

slave	Pointer to the FLEXIO_I2S slave driver context structure.
txBuff	pointer to the buffer containing transmit data
txSize	length in bytes of the data to be transferred

Returns

Error or success status returned by API Implements : FLEXIO_I2S_DRV_SlaveSetTxBuffer_Activity

Definition at line 582 of file flexio_i2s_driver.h.

16.28.4.23 static status_t FLEXIO_I2S_DRV_SlaveTransferAbort (flexio_i2s_slave_state_t ∗ slave) `[inline],` `[static]`

Aborts a non-blocking I2S slave transaction.

This function aborts a non-blocking I2S transfer.

Parameters

slave	Pointer to the FLEXIO_I2S slave driver context structure.

Returns

Error or success status returned by API Implements : FLEXIO_I2S_DRV_SlaveTransferAbort_Activity

Definition at line 521 of file flexio_i2s_driver.h.

16.29 FlexIO SPI Driver

16.29.1 Detailed Description

SPI communication over FlexIO module (FLEXIO_SPI)

The FLEXIO_SPI Driver allows communication on an SPI bus using the FlexIO module in the S32144K processor.

Features

- Master or slave operation

- Interrupt, DMA or polling mode

- Provides blocking and non-blocking transfer functions

- Configurable baud rate

- Configurable clock polarity and phase

- Configurable bit order and data size

Functionality

Before using any Flexio driver the device must first be initialized using function FLEXIO_DRV_InitDevice. Then the FLEXIO_SPI Driver must be initialized, using functions FLEXIO_SPI_DRV_MasterInit() or FLEXIO_SPI_DRV_↩ SlaveInit(). It is possible to use more driver instances on the same FlexIO device, as long as sufficient resources are available. Different driver instances on the same FlexIO device can function independently of each other. When it is no longer needed, the driver can be de-initialized, using FLEXIO_SPI_DRV_MasterDeinit() or FLEXIO_S↩ PI_DRV_SlaveDeinit(). This will release the hardware resources, allowing other driver instances to be initialized other.

Master Mode

Master Mode provides functions for transmitting or receiving data to/from an SPI slave. Baud rate is provided at initialization time through the master configuration structure, but can be changed at runtime by using FLEXIO_SP↩ I_DRV_MasterSetBaudRate() function. Note that due to module limitation not any baud rate can be achieved. The driver will set a baud rate as close as possible to the requested baud rate, but there may still be substantial differ- ences, for example if requesting a high baud rate while using a low-frequency FlexIO clock. The application should call FLEXIO_SPI_DRV_MasterGetBaudRate() after FLEXIO_SPI_DRV_MasterSetBaudRate() to check what baud rate was actually set.

To send or receive data, use function FLEXIO_SPI_DRV_MasterTransfer(). The transmit and receive buffers, to- gether with parameters for the transfer are provided through the flexio_spi_transfer_t structure. If only transmit or receive is desired, any one of the Rx/Tx buffers can be set to NULL. This driver does not support continuous send/receive using a user callback function. The callback function is only used to signal the end of a transfer.

Blocking operations will return only when the transfer is completed, either successfully or with error. Non-blocking operations will initiate the transfer and return STATUS_SUCCESS, but the module is still busy with the transfer and another transfer can't be initiated until the current transfer is complete. The application will be notified through the user callback when the transfer completes, or it can check the status of the current transfer by calling FLEXIO_S↩ PI_DRV_MasterGetStatus(). If the transfer is still ongoing this function will return STATUS_BUSY. If the transfer is completed, the function will return either STATUS_SUCCESS or an error code, depending on the outcome of the last transfer.

The driver supports interrupt, DMA and polling mode. In polling mode the function FLEXIO_SPI_DRV_Master↩ GetStatus() ensures the progress of the transfer by checking and handling transmit and receive events reported by the FlexIO module. The application should ensure that this function is called often enough (at least once per transferred byte)to avoid Tx underflows or Rx overflows. In DMA mode the DMA channels that will be used by the driver are received through the configuration structure. The channels must be initialized by the application before the flexio_spi driver is initialized. The flexio_spi driver will only set the DMA request source.

Slave Mode

Slave Mode is very similar to master mode, the main difference being that the FLEXIO_SPI_DRV_SlaveInit() function initializes the FlexIO module to use the clock signal received from the master instead of generating it. Consequently, there is no `SetBaudRate` function in slave mode. Other than that, the slave mode offers a similar interface to the master mode. FLEXIO_SPI_DRV_MasterTransfer() can be used to initiate transfers, and FLEXIO←_SPI_DRV_SlaveGetStatus() is used to check the status of the transfer and advance the transfer in polling mode. All other specifications from the Master Mode description apply for Slave Mode too

Important Notes

- Before using the FLEXIO_SPI Driver the protocol clock of the module must be configured. Refer to Clock Manager for clock configuration.

- Before using the FLEXIO_SPI Driver the pins must be routed to the FlexIO module. Refer to PINS Driver for pin routing configuration. Note that any of the available FlexIO pins can be used for MOSI, MISO, SCK and SS (configurable at initialization time).

- The driver enables the interrupts for the corresponding FlexIO module, but any interrupt priority setting must be done by the application.

- The driver does not support back-to-back transmission mode for CPHA = 1

- The driver does not support configurable polarity for SS signal (only active-low is supported)

- Timeout feature for blocking transfers does not work in polling mode.

- This driver needs two shifters and two timers for its operation. Initialization will fail if there are not enough shifters and timers available on the FlexIO device.

- This driver needs two DMA channels for its operation when it is initialized in DMA mode. The DMA channels must be initialized by the application before initializing the driver. Refer to EDMA driver for DMA channels initialization.

- If the application uses an RTOS, this driver uses a semaphore for blocking transfers. Initialization will fail if the semaphore cannot be created. If the driver uses polling mode no semaphore is used.

- If the application uses an RTOS, the FlexIO drivers use a mutex for channel allocation. Only one mutex per device is needed, not per driver instance. Device initialization will fail if the mutex cannot be created.

Data Structures

- struct flexio_spi_master_user_config_t

    Master configuration structure. More...
- struct flexio_spi_slave_user_config_t

    Slave configuration structure. More...
- struct flexio_spi_master_state_t

    Master internal context structure. More...

Typedefs

- typedef flexio_spi_master_state_t flexio_spi_slave_state_t

    Slave internal context structure.

Enumerations

- enum flexio_spi_transfer_bit_order_t { FLEXIO_SPI_TRANSFER_MSB_FIRST = 0U, FLEXIO_SPI_TRAN↩
SFER_LSB_FIRST = 1U }

    Order in which the data bits are transferred Implements : flexio_spi_transfer_bit_order_t_Class.

- enum flexio_spi_transfer_size_t { FLEXIO_SPI_TRANSFER_1BYTE = 1U, FLEXIO_SPI_TRANSFER_2B↩
YTE = 2U, FLEXIO_SPI_TRANSFER_4BYTE = 4U }

    Size of transferred data in bytes Implements : flexio_spi_transfer_size_t_Class.

FLEXIO_SPI Driver

- status_t FLEXIO_SPI_DRV_MasterInit (uint32_t instance, const flexio_spi_master_user_config_t ∗user↩
ConfigPtr, flexio_spi_master_state_t ∗master)

    Initialize the FLEXIO_SPI master mode driver.

- status_t FLEXIO_SPI_DRV_MasterDeinit (flexio_spi_master_state_t ∗master)

    De-initialize the FLEXIO_SPI master mode driver.

- status_t FLEXIO_SPI_DRV_MasterSetBaudRate (flexio_spi_master_state_t ∗master, uint32_t baudRate)

    Set the baud rate for any subsequent SPI communication.

- status_t FLEXIO_SPI_DRV_MasterGetBaudRate (flexio_spi_master_state_t ∗master, uint32_t ∗baudRate)

    Get the currently configured baud rate.

- status_t FLEXIO_SPI_DRV_MasterTransfer (flexio_spi_master_state_t ∗master, const uint8_t ∗txData, uint8_t ∗rxData, uint32_t dataSize)

    Perform a non-blocking SPI master transaction.

- status_t FLEXIO_SPI_DRV_MasterTransferBlocking (flexio_spi_master_state_t ∗master, const uint8_t ∗tx↩
Data, uint8_t ∗rxData, uint32_t dataSize, uint32_t timeout)

    Perform a blocking SPI master transaction.

- status_t FLEXIO_SPI_DRV_MasterTransferAbort (flexio_spi_master_state_t ∗master)

    Aborts a non-blocking SPI master transaction.

- status_t FLEXIO_SPI_DRV_MasterGetStatus (flexio_spi_master_state_t ∗master, uint32_t ∗bytes↩
Remaining)

    Get the status of the current non-blocking SPI master transaction.

- status_t FLEXIO_SPI_DRV_SlaveInit (uint32_t instance, const flexio_spi_slave_user_config_t ∗userConfig↩
Ptr, flexio_spi_slave_state_t ∗slave)

    Initialize the FLEXIO_SPI slave mode driver.

- static status_t FLEXIO_SPI_DRV_SlaveDeinit (flexio_spi_slave_state_t ∗slave)

    De-initialize the FLEXIO_SPI slave mode driver.

- static status_t FLEXIO_SPI_DRV_SlaveTransfer (flexio_spi_slave_state_t ∗slave, const uint8_t ∗txData, uint8_t ∗rxData, uint32_t dataSize)

    Perform a non-blocking SPI slave transaction.

- static status_t FLEXIO_SPI_DRV_SlaveTransferBlocking (flexio_spi_slave_state_t ∗slave, const uint8_t ∗tx↩
Data, uint8_t ∗rxData, uint32_t dataSize, uint32_t timeout)

    Perform a blocking SPI slave transaction.

- static status_t FLEXIO_SPI_DRV_SlaveTransferAbort (flexio_spi_slave_state_t ∗slave)

    Aborts a non-blocking SPI slave transaction.

- static status_t FLEXIO_SPI_DRV_SlaveGetStatus (flexio_spi_slave_state_t ∗slave, uint32_t ∗bytes↩
Remaining)

    Get the status of the current non-blocking SPI slave transaction.

16.29.2   Data Structure Documentation

16.29.2.1   struct flexio_spi_master_user_config_t

Master configuration structure.

This structure is used to provide configuration parameters for the flexio_spi master at initialization time. Implements : flexio_spi_master_user_config_t_Class

Definition at line 70 of file flexio_spi_driver.h.

Data Fields

- uint32_t baudRate
- flexio_driver_type_t driverType
- flexio_spi_transfer_bit_order_t bitOrder
- flexio_spi_transfer_size_t transferSize
- uint8_t clockPolarity
- uint8_t clockPhase
- uint8_t mosiPin
- uint8_t misoPin
- uint8_t sckPin
- uint8_t ssPin
- spi_callback_t callback
- void ∗ callbackParam
- uint8_t rxDMAChannel
- uint8_t txDMAChannel

Field Documentation

16.29.2.1.1   uint32_t baudRate

Baud rate in hertz

Definition at line 72 of file flexio_spi_driver.h.

16.29.2.1.2   flexio_spi_transfer_bit_order_t bitOrder

Bit order: LSB-first / MSB-first

Definition at line 74 of file flexio_spi_driver.h.

16.29.2.1.3   spi_callback_t callback

User callback function. Note that this function will be called from the interrupt service routine, so its execution time should be as small as possible. It can be NULL if it is not needed

Definition at line 82 of file flexio_spi_driver.h.

16.29.2.1.4   void∗ callbackParam

Parameter for the callback function

Definition at line 86 of file flexio_spi_driver.h.

16.29.2.1.5   uint8_t clockPhase

Clock Phase (CPHA) 0 = sample on leading clock edge; 1 = sample on trailing clock edge

Definition at line 77 of file flexio_spi_driver.h.

16.29.2.1.6   uint8_t clockPolarity

Clock Polarity (CPOL) 0 = active-high clock; 1 = active-low clock

Definition at line 76 of file flexio_spi_driver.h.

16.29.2.1.7   flexio_driver_type_t driverType

Driver type: interrupts/polling/DMA

Definition at line 73 of file flexio_spi_driver.h.

16.29.2.1.8   uint8_t misoPin

Flexio pin to use as MISO pin

Definition at line 79 of file flexio_spi_driver.h.

16.29.2.1.9   uint8_t mosiPin

Flexio pin to use as MOSI pin

Definition at line 78 of file flexio_spi_driver.h.

16.29.2.1.10   uint8_t rxDMAChannel

Rx DMA channel number. Only used in DMA mode

Definition at line 87 of file flexio_spi_driver.h.

16.29.2.1.11   uint8_t sckPin

Flexio pin to use as SCK pin

Definition at line 80 of file flexio_spi_driver.h.

16.29.2.1.12   uint8_t ssPin

Flexio pin to use as SS pin

Definition at line 81 of file flexio_spi_driver.h.

16.29.2.1.13   flexio_spi_transfer_size_t transferSize

Transfer size in bytes: 1/2/4

Definition at line 75 of file flexio_spi_driver.h.

16.29.2.1.14   uint8_t txDMAChannel

Tx DMA channel number. Only used in DMA mode

Definition at line 88 of file flexio_spi_driver.h.

16.29.2.2   struct flexio_spi_slave_user_config_t

Slave configuration structure.

This structure is used to provide configuration parameters for the flexio_spi slave at initialization time. Implements : flexio_spi_slave_user_config_t_Class

Definition at line 97 of file flexio_spi_driver.h.

Data Fields

- flexio_driver_type_t driverType
- flexio_spi_transfer_bit_order_t bitOrder

- flexio_spi_transfer_size_t transferSize
- uint8_t clockPolarity
- uint8_t clockPhase
- uint8_t mosiPin
- uint8_t misoPin
- uint8_t sckPin
- uint8_t ssPin
- spi_callback_t callback
- void ∗ callbackParam
- uint8_t rxDMAChannel
- uint8_t txDMAChannel

Field Documentation

16.29.2.2.1   flexio_spi_transfer_bit_order_t bitOrder

Bit order: LSB-first / MSB-first

Definition at line 100 of file flexio_spi_driver.h.

16.29.2.2.2   spi_callback_t callback

User callback function. Note that this function will be called from the interrupt service routine, so its execution time should be as small as possible. It can be NULL if it is not needed

Definition at line 108 of file flexio_spi_driver.h.

16.29.2.2.3   void∗ callbackParam

Parameter for the callback function

Definition at line 112 of file flexio_spi_driver.h.

16.29.2.2.4   uint8_t clockPhase

Clock Phase (CPHA) 0 = sample on leading clock edge; 1 = sample on trailing clock edge

Definition at line 103 of file flexio_spi_driver.h.

16.29.2.2.5   uint8_t clockPolarity

Clock Polarity (CPOL) 0 = active-low clock; 1 = active-high clock

Definition at line 102 of file flexio_spi_driver.h.

16.29.2.2.6   flexio_driver_type_t driverType

Driver type: interrupts/polling/DMA

Definition at line 99 of file flexio_spi_driver.h.

16.29.2.2.7   uint8_t misoPin

Flexio pin to use as MISO pin

Definition at line 105 of file flexio_spi_driver.h.

16.29.2.2.8   uint8_t mosiPin

Flexio pin to use as MOSI pin

Definition at line 104 of file flexio_spi_driver.h.

16.29.2.2.9 uint8_t rxDMAChannel

Rx DMA channel number. Only used in DMA mode

Definition at line 113 of file flexio_spi_driver.h.

16.29.2.2.10 uint8_t sckPin

Flexio pin to use as SCK pin

Definition at line 106 of file flexio_spi_driver.h.

16.29.2.2.11 uint8_t ssPin

Flexio pin to use as SS pin

Definition at line 107 of file flexio_spi_driver.h.

16.29.2.2.12 flexio_spi_transfer_size_t transferSize

Transfer size in bytes: 1/2/4

Definition at line 101 of file flexio_spi_driver.h.

16.29.2.2.13 uint8_t txDMAChannel

Tx DMA channel number. Only used in DMA mode

Definition at line 114 of file flexio_spi_driver.h.

16.29.2.3 struct flexio_spi_master_state_t

Master internal context structure.

This structure is used by the master-mode driver for its internal logic. It must be provided by the application through the FLEXIO_SPI_DRV_MasterInit() function, then it cannot be freed until the driver is de-initialized using FLEXI←O_SPI_DRV_MasterDeinit(). The application should make no assumptions about the content of this structure.

Definition at line 126 of file flexio_spi_driver.h.

16.29.3 Typedef Documentation

16.29.3.1 typedef flexio_spi_master_state_t flexio_spi_slave_state_t

Slave internal context structure.

This structure is used by the slave-mode driver for its internal logic. It must be provided by the application through the FLEXIO_SPI_DRV_SlaveInit() function, then it cannot be freed until the driver is de-initialized using FLEXIO←_SPI_DRV_SlaveDeinit(). The application should make no assumptions about the content of this structure.

Definition at line 158 of file flexio_spi_driver.h.

16.29.4 Enumeration Type Documentation

16.29.4.1 enum flexio_spi_transfer_bit_order_t

Order in which the data bits are transferred Implements : flexio_spi_transfer_bit_order_t_Class.

Enumerator

> ***FLEXIO_SPI_TRANSFER_MSB_FIRST*** Transmit data starting with most significant bit
> ***FLEXIO_SPI_TRANSFER_LSB_FIRST*** Transmit data starting with least significant bit

Definition at line 42 of file flexio_spi_driver.h.

16.29.4.2   enum flexio_spi_transfer_size_t

Size of transferred data in bytes Implements : flexio_spi_transfer_size_t_Class.

Enumerator

FLEXIO_SPI_TRANSFER_1BYTE	Data size is 1-byte
FLEXIO_SPI_TRANSFER_2BYTE	Data size is 2-bytes
FLEXIO_SPI_TRANSFER_4BYTE	Data size is 4-bytes

Definition at line 51 of file flexio_spi_driver.h.

16.29.5   Function Documentation

16.29.5.1   status_t FLEXIO_SPI_DRV_MasterDeinit (flexio_spi_master_state_t ∗ *master*)

De-initialize the FLEXIO_SPI master mode driver.

This function de-initializes the FLEXIO_SPI driver in master mode. The driver can't be used again until reinitialized. The context structure is no longer needed by the driver and can be freed after calling this function.

Parameters

master	Pointer to the FLEXIO_SPI master driver context structure.

Returns

> Error or success status returned by API

Definition at line 985 of file flexio_spi_driver.c.

16.29.5.2   status_t FLEXIO_SPI_DRV_MasterGetBaudRate (flexio_spi_master_state_t ∗ *master,* uint32_t ∗ *baudRate*)

Get the currently configured baud rate.

This function returns the currently configured SPI baud rate.

Parameters

master	Pointer to the FLEXIO_SPI master driver context structure.
baudRate	the current baud rate in hertz

Returns

> Error or success status returned by API

Definition at line 1051 of file flexio_spi_driver.c.

16.29.5.3   status_t FLEXIO_SPI_DRV_MasterGetStatus (flexio_spi_master_state_t ∗ *master,* uint32_t ∗ *bytesRemaining*)

Get the status of the current non-blocking SPI master transaction.

This function returns the current status of a non-blocking SPI master transaction. A return code of STATUS_BUSY means the transfer is still in progress. Otherwise the function returns a status reflecting the outcome of the last transfer. When the driver is initialized in polling mode this function also advances the transfer by checking and handling the transmit and receive events, so it must be called frequently to avoid overflows or underflows.

Parameters

master	Pointer to the FLEXIO_SPI master driver context structure.
bytesRemaining	the remaining number of bytes to be transferred

Returns

      Error or success status returned by API

Definition at line 1204 of file flexio_spi_driver.c.

16.29.5.4   status_t FLEXIO_SPI_DRV_MasterInit (uint32_t *instance,* const **flexio_spi_master_user_config_t ∗ *userConfigPtr,* **flexio_spi_master_state_t** ∗ *master*)**

Initialize the FLEXIO_SPI master mode driver.

This function initializes the FLEXIO_SPI driver in master mode.

Parameters

instance	FLEXIO peripheral instance number
userConfigPtr	Pointer to the FLEXIO_SPI master user configuration structure. The function reads configuration data from this structure and initializes the driver accordingly. The application may free this structure after the function returns.
master	Pointer to the FLEXIO_SPI master driver context structure. The driver uses this memory area for its internal logic. The application must make no assumptions about the content of this structure, and must not free this memory until the driver is de-initialized using FLEXIO↩ _SPI_DRV_MasterDeinit().

Returns

      Error or success status returned by API

Definition at line 896 of file flexio_spi_driver.c.

16.29.5.5   status_t FLEXIO_SPI_DRV_MasterSetBaudRate (flexio_spi_master_state_t ∗ *master,* uint32_t *baudRate*)

Set the baud rate for any subsequent SPI communication.

This function sets the baud rate for the SPI master. Note that due to module limitation not any baud rate can be achieved. The driver will set a baud rate as close as possible to the requested baud rate, but there may still be substantial differences, for example if requesting a high baud rate while using a low-frequency FlexIO clock. The application should call FLEXIO_SPI_DRV_MasterGetBaudRate() after FLEXIO_SPI_DRV_MasterSetBaudRate() to check what baud rate was actually set.

Parameters

master	Pointer to the FLEXIO_SPI master driver context structure.
baudRate	the desired baud rate in hertz

Returns

      Error or success status returned by API

Definition at line 1009 of file flexio_spi_driver.c.

16.29.5.6   status_t FLEXIO_SPI_DRV_MasterTransfer (flexio_spi_master_state_t ∗ *master,* const uint8_t ∗ *txData,* uint8_t ∗ *rxData,* uint32_t *dataSize*)

Perform a non-blocking SPI master transaction.

This function performs an SPI full-duplex transaction, transmit and receive in parallel. If only transmit or receive are required, it is possible to provide NULL pointers for txData or rxData. The transfer is non-blocking, the function

only initiates the transfer and then returns, leaving the transfer to complete asynchronously). FLEXIO_SPI_DRV↩
_MasterGetStatus() can be called to check the status of the transfer.

Parameters

master	Pointer to the FLEXIO_SPI master driver context structure.
txData	pointer to the data to be transmitted
rxData	pointer to the buffer where to store received data
dataSize	length in bytes of the data to be transferred

Returns

     Error or success status returned by API

Definition at line 1091 of file flexio_spi_driver.c.

16.29.5.7 status_t FLEXIO_SPI_DRV_MasterTransferAbort (flexio_spi_master_state_t ∗ *master*)

Aborts a non-blocking SPI master transaction.

This function aborts a non-blocking SPI transfer.

Parameters

master	Pointer to the FLEXIO_SPI master driver context structure.

Returns

     Error or success status returned by API

Definition at line 1186 of file flexio_spi_driver.c.

16.29.5.8 status_t FLEXIO_SPI_DRV_MasterTransferBlocking (flexio_spi_master_state_t ∗ *master,* const uint8_t ∗ *txData,* uint8_t ∗ *rxData,* uint32_t *dataSize,* uint32_t *timeout*)

Perform a blocking SPI master transaction.

This function performs an SPI full-duplex transaction, transmit and receive in parallel. If only transmit or receive are required, it is possible to provide NULL pointers for txData or rxData. The transfer is blocking, the function only returns when the transfer is complete.

Parameters

master	Pointer to the FLEXIO_SPI master driver context structure.
txData	pointer to the data to be transmitted
rxData	pointer to the buffer where to store received data
dataSize	length in bytes of the data to be transferred
timeout	timeout for the transfer in milliseconds

Returns

     Error or success status returned by API

Definition at line 1153 of file flexio_spi_driver.c.

16.29.5.9 static status_t FLEXIO_SPI_DRV_SlaveDeinit (flexio_spi_slave_state_t ∗ *slave*) `[inline],[static]`

De-initialize the FLEXIO_SPI slave mode driver.

This function de-initializes the FLEXIO_SPI driver in slave mode. The driver can't be used again until reinitialized. The context structure is no longer needed by the driver and can be freed after calling this function.

Parameters

slave	Pointer to the FLEXIO_SPI slave driver context structure.

Returns

> Error or success status returned by API Implements : FLEXIO_SPI_DRV_SlaveDeinit_Activity

Definition at line 341 of file flexio_spi_driver.h.

16.29.5.10   static status_t FLEXIO_SPI_DRV_SlaveGetStatus (flexio_spi_slave_state_t ∗ slave, uint32_t ∗ bytesRemaining) [inline],[static]

Get the status of the current non-blocking SPI slave transaction.

This function returns the current status of a non-blocking SPI slave transaction. A return code of STATUS_BUSY means the transfer is still in progress. Otherwise the function returns a status reflecting the outcome of the last transfer. When the driver is initialized in polling mode this function also advances the transfer by checking and handling the transmit and receive events, so it must be called frequently to avoid overflows or underflows.

Parameters

slave	Pointer to the FLEXIO_SPI slave driver context structure.
bytesRemaining	the remaining number of bytes to be transferred

Returns

> Error or success status returned by API Implements : FLEXIO_SPI_DRV_SlaveGetStatus_Activity

Definition at line 428 of file flexio_spi_driver.h.

16.29.5.11   status_t FLEXIO_SPI_DRV_SlaveInit (uint32_t instance, const flexio_spi_slave_user_config_t ∗ userConfigPtr, flexio_spi_slave_state_t ∗ slave)

Initialize the FLEXIO_SPI slave mode driver.

This function initializes the FLEXIO_SPI driver in slave mode.

Parameters

instance	FLEXIO peripheral instance number
userConfigPtr	Pointer to the FLEXIO_SPI slave user configuration structure. The function reads configuration data from this structure and initializes the driver accordingly. The application may free this structure after the function returns.
slave	Pointer to the FLEXIO_SPI slave driver context structure. The driver uses this memory area for its internal logic. The application must make no assumptions about the content of this structure, and must not free this memory until the driver is de-initialized using FLEXIO_SP↩ I_DRV_SlaveDeinit().

Returns

> Error or success status returned by API

Definition at line 1252 of file flexio_spi_driver.c.

16.29.5.12   static status_t FLEXIO_SPI_DRV_SlaveTransfer (flexio_spi_slave_state_t ∗ slave, const uint8_t ∗ txData, uint8_t ∗ rxData, uint32_t dataSize) [inline],[static]

Perform a non-blocking SPI slave transaction.

This function performs an SPI full-duplex transaction, transmit and receive in parallel. If only transmit or receive are required, it is possible to provide NULL pointers for txData or rxData. The transfer is non-blocking, the function only initiates the transfer and then returns, leaving the transfer to complete asynchronously). FLEXIO_SPI_DRV↩ _SlaveGetStatus() can be called to check the status of the transfer.

Parameters

slave	Pointer to the FLEXIO_SPI slave driver context structure.
txData	pointer to the data to be transmitted
rxData	pointer to the buffer where to store received data
dataSize	length in bytes of the data to be transferred

Returns

> Error or success status returned by API Implements : FLEXIO_SPI_DRV_SlaveTransfer_Activity

Definition at line 363 of file flexio_spi_driver.h.

16.29.5.13   static status_t FLEXIO_SPI_DRV_SlaveTransferAbort (flexio_spi_slave_state_t ∗ slave) `[inline],` `[static]`

Aborts a non-blocking SPI slave transaction.

This function aborts a non-blocking SPI transfer.

Parameters

slave	Pointer to the FLEXIO_SPI slave driver context structure.

Returns

> Error or success status returned by API Implements : FLEXIO_SPI_DRV_SlaveTransferAbort_Activity

Definition at line 407 of file flexio_spi_driver.h.

16.29.5.14   static status_t FLEXIO_SPI_DRV_SlaveTransferBlocking (flexio_spi_slave_state_t ∗ slave, const uint8_t ∗ txData, uint8_t ∗ rxData, uint32_t dataSize, uint32_t timeout) `[inline],[static]`

Perform a blocking SPI slave transaction.

This function performs an SPI full-duplex transaction, transmit and receive in parallel. If only transmit or receive are required, it is possible to provide NULL pointers for txData or rxData. The transfer is blocking, the function only returns when the transfer is complete.

Parameters

slave	Pointer to the FLEXIO_SPI slave driver context structure.
txData	pointer to the data to be transmitted
rxData	pointer to the buffer where to store received data
dataSize	length in bytes of the data to be transferred
timeout	timeout for the transfer in milliseconds

Returns

> Error or success status returned by API Implements : FLEXIO_SPI_DRV_SlaveTransferBlocking_Activity

Definition at line 388 of file flexio_spi_driver.h.

16.30 FlexIO UART Driver

16.30.1 Detailed Description

UART communication over FlexIO module (FLEXIO_UART)

The FLEXIO_UART Driver allows UART communication using the FlexIO module in the S32144K processor.

Features

- Interrupt, DMA or polling mode

- Provides blocking and non-blocking transmit and receive functions

- Configurable baud rate and number of bits

- Single stop bit only

- Parity bit not supported

Functionality

Initialization

Before using any Flexio driver the device must first be initialized using function FLEXIO_DRV_InitDevice. Then the FLEXIO_UART Driver must be initialized, using function FLEXIO_UART_DRV_Init(). It is possible to use more driver instances on the same FlexIO device, as long as sufficient resources are available. Different driver instances on the same FlexIO device can function independently of each other. When it is no longer needed, the driver can be de-initialized, using FLEXIO_UART_DRV_Deinit(). This will release the hardware resources, allowing other driver instances to be initialized.

Choosing transmit/receive mode

To initialize the UART driver in transmit / receive mode the `direction` field of the configuration structure must be set to `FLEXIO_UART_DIRECTION_TX` / `FLEXIO_UART_DIRECTION_RX` when calling FLEXIO_UAR↩
T_DRV_Init(). Once configured for one direction the driver must be used only for the chosen direction until it is de-initialized. One driver instance can only work in one direction at a time, but more driver instances can be created on the same device, up to the number of shifters present on the device (for example on S32K144 up to 4 driver instances can run in parallel on one device).

Setting the baud rate and bit count

The baud rate and bit count are provided at initialization time through the master configuration structure, but they can be changed at runtime by using function FLEXIO_UART_DRV_SetConfig(). Note that due to module limitation not any baud rate can be achieved. The driver will set a baud rate as close as possible to the requested baud rate, but there may still be substantial differences, for example if requesting a high baud rate while using a low-frequency FlexIO clock. The application should call FLEXIO_UART_DRV_GetBaudRate() to check what baud rate was actually set.

Transmitting / Receiving

To send or receive data to/from the currently configured slave address, use functions FLEXIO_UART_DRV_Send↩
Data() or FLEXIO_UART_DRV_ReceiveData() (or their blocking counterparts). Continuous send/receive can be realized by registering a user callback function. When the driver completes the transmission or reception of the current buffer, it will invoke the user callback with an appropriate event. The callback function can the use FLEXI↩
O_UART_DRV_SetTxBuffer() orFLEXIO_UART_DRV_SetRxBuffer() to provide a new buffer.

Blocking operations will return only when the transfer is completed, either successfully or with error. Non-blocking operations will initiate the transfer and return STATUS_SUCCESS, but the module is still busy with the transfer and another transfer can't be initiated until the current transfer is complete. The application will be notified through the user callback when the transfer completes, or it can check the status of the current transfer by calling FLEXIO↩
_UART_DRV_GetStatus(). If the transfer is still ongoing this function will return STATUS_BUSY. If the transfer is

completed, the function will return either STATUS_SUCCESS or an error code, depending on the outcome of the last transfer.

The driver supports interrupt, DMA and polling mode. In polling mode the function FLEXIO_UART_DRV_Get↩
Status() ensures the progress of the transfer by checking and handling transmit and receive events reported by the FlexIO module. The application should ensure that this function is called often enough (at least once per transferred byte) to avoid Tx underflows or Rx overflows. In DMA mode the DMA channel that will be used by the driver is received through the configuration structure. The channel must be initialized by the application before the flexio_↩
uart driver is initialized. The flexio_uart driver will only set the DMA request source.

Important Notes

- Before using the FLEXIO_UART Driver the FlexIO clock must be configured. Refer to Clock Manager for clock configuration.

- Before using the FLEXIO_UART Driver the pins must be routed to the FlexIO module. Refer to PINS Driver for pin routing configuration. Note that any of the available FlexIO pins can be used for the UART TX / RX line (configurable at initialization time). If more than one driver instance is used on the same Flexio module, it is the responsibility of the application to ensure there are no conflicts between pins.

- The driver enables the interrupts for the corresponding FlexIO module, but any interrupt priority setting must be done by the application.

- Timeout feature for blocking transfers does not work in polling mode.

- This driver needs one shifter and one timer for its operation. Initialization will fail if there are not enough shifters and timers available on the FlexIO device.

- This driver needs one DMA channel for its operation when it is initialized in DMA mode. The DMA channels must be initialized by the application before initializing the driver. Refer to EDMA driver for DMA channels initialization.

- If the application uses an RTOS, this driver uses a semaphore for blocking transfers. Initialization will fail if the semaphore cannot be created. If the driver uses polling mode no semaphore is used.

- If the application uses an RTOS, the FlexIO drivers use a mutex for channel allocation. Only one mutex per device is needed, not per driver instance. Device initialization will fail if the mutex cannot be created.

Data Structures

- struct flexio_uart_user_config_t

    Driver configuration structure. More...

- struct flexio_uart_state_t

    Driver internal context structure. More...

Enumerations

- enum flexio_uart_driver_direction_t { FLEXIO_UART_DIRECTION_TX = 0x01U, FLEXIO_UART_DIRECT↩
  ION_RX = 0x00U }

    flexio_uart driver direction (tx or rx)

FLEXIO_UART Driver

- status_t FLEXIO_UART_DRV_Init (uint32_t instance, const flexio_uart_user_config_t ∗userConfigPtr, flexio_uart_state_t ∗state)

    Initialize the FLEXIO_UART driver.

- status_t FLEXIO_UART_DRV_Deinit (flexio_uart_state_t ∗state)

    De-initialize the FLEXIO_UART driver.

- status_t FLEXIO_UART_DRV_SetConfig (flexio_uart_state_t ∗state, uint32_t baudRate, uint8_t bitCount)

    Set the baud rate and bit width for any subsequent UART communication.
- status_t FLEXIO_UART_DRV_GetBaudRate (flexio_uart_state_t ∗state, uint32_t ∗baudRate)

    Get the currently configured baud rate.
- status_t FLEXIO_UART_DRV_SendDataBlocking (flexio_uart_state_t ∗state, const uint8_t ∗txBuff, uint32←‐
_t txSize, uint32_t timeout)

    Perform a blocking UART transmission.
- status_t FLEXIO_UART_DRV_SendData (flexio_uart_state_t ∗state, const uint8_t ∗txBuff, uint32_t txSize)

    Perform a non-blocking UART transmission.
- status_t FLEXIO_UART_DRV_ReceiveDataBlocking (flexio_uart_state_t ∗state, uint8_t ∗rxBuff, uint32←‐
t rxSize, uint32_t timeout)

    Perform a blocking UART reception.
- status_t FLEXIO_UART_DRV_ReceiveData (flexio_uart_state_t ∗state, uint8_t ∗rxBuff, uint32_t rxSize)

    Perform a non-blocking UART reception.
- status_t FLEXIO_UART_DRV_GetStatus (flexio_uart_state_t ∗state, uint32_t ∗bytesRemaining)

    Get the status of the current non-blocking UART transfer.
- status_t FLEXIO_UART_DRV_TransferAbort (flexio_uart_state_t ∗state)

    Aborts a non-blocking UART transfer.
- status_t FLEXIO_UART_DRV_SetRxBuffer (flexio_uart_state_t ∗state, uint8_t ∗rxBuff, uint32_t rxSize)

    Provide a buffer for receiving data.
- status_t FLEXIO_UART_DRV_SetTxBuffer (flexio_uart_state_t ∗state, const uint8_t ∗txBuff, uint32_t txSize)

    Provide a buffer for transmitting data.

16.30.2   Data Structure Documentation

16.30.2.1   struct flexio_uart_user_config_t

Driver configuration structure.

This structure is used to provide configuration parameters for the flexio_uart driver at initialization time. Implements
: flexio_uart_user_config_t_Class

Definition at line 63 of file flexio_uart_driver.h.

Data Fields

- flexio_driver_type_t driverType
- uint32_t baudRate
- uint8_t bitCount
- flexio_uart_driver_direction_t direction
- uint8_t dataPin
- uart_callback_t callback
- void ∗ callbackParam
- uint8_t dmaChannel

Field Documentation

16.30.2.1.1   uint32_t baudRate

Baud rate in hertz

Definition at line 66 of file flexio_uart_driver.h.

16.30.2.1.2   uint8_t bitCount

Number of bits per word

Definition at line 67 of file flexio_uart_driver.h.

16.30.2.1.3   uart_callback_t callback

User callback function. Note that this function will be called from the interrupt service routine, so its execution time should be as small as possible. It can be NULL if it is not needed

Definition at line 70 of file flexio_uart_driver.h.

16.30.2.1.4   void∗ callbackParam

Parameter for the callback function

Definition at line 74 of file flexio_uart_driver.h.

16.30.2.1.5   uint8_t dataPin

Flexio pin to use as Tx or Rx pin

Definition at line 69 of file flexio_uart_driver.h.

16.30.2.1.6   flexio_uart_driver_direction_t direction

Driver direction: Tx or Rx

Definition at line 68 of file flexio_uart_driver.h.

16.30.2.1.7   uint8_t dmaChannel

DMA channel number. Only used in DMA mode

Definition at line 75 of file flexio_uart_driver.h.

16.30.2.1.8   flexio_driver_type_t driverType

Driver type: interrupts/polling/DMA

Definition at line 65 of file flexio_uart_driver.h.

16.30.2.2   struct flexio_uart_state_t

Driver internal context structure.

This structure is used by the flexio_uart driver for its internal logic. It must be provided by the application through the FLEXIO_UART_DRV_Init() function, then it cannot be freed until the driver is de-initialized using FLEXIO_U↩ART_DRV_DeInit(). The application should make no assumptions about the content of this structure.

Definition at line 87 of file flexio_uart_driver.h.

16.30.3   Enumeration Type Documentation

16.30.3.1   enum flexio_uart_driver_direction_t

flexio_uart driver direction (tx or rx)

This structure describes the direction configuration options for the flexio_uart driver. Implements : flexio_uart_↩driver_direction_t_Class

Enumerator

> ***FLEXIO_UART_DIRECTION_TX***   Tx UART driver
> ***FLEXIO_UART_DIRECTION_RX***   Rx UART driver

Definition at line 45 of file flexio_uart_driver.h.

16.30.4   Function Documentation

16.30.4.1   status_t FLEXIO_UART_DRV_Deinit (flexio_uart_state_t ∗ state)

De-initialize the FLEXIO_UART driver.

This function de-initializes the FLEXIO_UART driver. The driver can't be used again until reinitialized. The context structure is no longer needed by the driver and can be freed after calling this function.

Parameters

state	Pointer to the FLEXIO_UART driver context structure.

Returns

> Error or success status returned by API

Definition at line 1046 of file flexio_uart_driver.c.

16.30.4.2   status_t FLEXIO_UART_DRV_GetBaudRate (flexio_uart_state_t ∗ state, uint32_t ∗ baudRate)

Get the currently configured baud rate.

This function returns the currently configured UART baud rate.

Parameters

state	Pointer to the FLEXIO_UART driver context structure.
baudRate	the current baud rate in hertz

Returns

> Error or success status returned by API

Definition at line 1125 of file flexio_uart_driver.c.

16.30.4.3   status_t FLEXIO_UART_DRV_GetStatus (flexio_uart_state_t ∗ state, uint32_t ∗ bytesRemaining)

Get the status of the current non-blocking UART transfer.

This function returns the current status of a non-blocking UART transfer. A return code of STATUS_BUSY means the transfer is still in progress. Otherwise the function returns a status reflecting the outcome of the last transfer. When the driver is initialized in polling mode this function also advances the transfer by checking and handling the transmit and receive events, so it must be called frequently to avoid overflows or underflows.

Parameters

state	Pointer to the FLEXIO_UART driver context structure.
bytesRemaining	the remaining number of bytes to be transferred

Returns

> Error or success status returned by API

Definition at line 1362 of file flexio_uart_driver.c.

16.30.4.4   status_t FLEXIO_UART_DRV_Init (uint32_t instance, const flexio_uart_user_config_t ∗ userConfigPtr, flexio_uart_state_t ∗ state)

Initialize the FLEXIO_UART driver.

This function initializes the FLEXIO_UART driver.

Parameters

instance	FLEXIO peripheral instance number
userConfigPtr	Pointer to the FLEXIO_UART user configuration structure. The function reads configuration data from this structure and initializes the driver accordingly. The application may free this structure after the function returns.
state	Pointer to the FLEXIO_UART driver context structure. The driver uses this memory area for its internal logic. The application must make no assumptions about the content of this structure, and must not free this memory until the driver is de-initialized using FLEXIO_UA↩RT_DRV_Deinit().

Returns

      Error or success status returned by API

Definition at line 941 of file flexio_uart_driver.c.

16.30.4.5   status_t FLEXIO_UART_DRV_ReceiveData (flexio_uart_state_t ∗ *state,* uint8_t ∗ *rxBuff,* uint32_t *rxSize*)

Perform a non-blocking UART reception.

This function receives a block of data and returns immediately. The rest of the transmission is handled by the interrupt service routine (if the driver is initialized in interrupt mode) or by the FLEXIO_UART_DRV_GetReceive↩Status() function (if the driver is initialized in polling mode).

Parameters

state	Pointer to the FLEXIO_UART driver context structure.
rxBuff	pointer to the receive buffer
rxSize	length in bytes of the data to be received

Returns

      Error or success status returned by API

Definition at line 1255 of file flexio_uart_driver.c.

16.30.4.6   status_t FLEXIO_UART_DRV_ReceiveDataBlocking (flexio_uart_state_t ∗ *state,* uint8_t ∗ *rxBuff,* uint32_t *rxSize,* uint32_t *timeout*)

Perform a blocking UART reception.

This function receives a block of data and only returns when the transmission is complete.

Parameters

state	Pointer to the FLEXIO_UART driver context structure.
rxBuff	pointer to the receive buffer
rxSize	length in bytes of the data to be received
timeout	timeout for the transfer in milliseconds

Returns

      Error or success status returned by API

Definition at line 1311 of file flexio_uart_driver.c.

16.30.4.7   status_t FLEXIO_UART_DRV_SendData (flexio_uart_state_t ∗ *state,* const uint8_t ∗ *txBuff,* uint32_t *txSize*)

Perform a non-blocking UART transmission.

This function sends a block of data and returns immediately. The rest of the transmission is handled by the interrupt service routine (if the driver is initialized in interrupt mode) or by the FLEXIO_UART_DRV_GetTransmitStatus() function (if the driver is initialized in polling mode).

Parameters

state	Pointer to the FLEXIO_UART driver context structure.
txBuff	pointer to the data to be transferred
txSize	length in bytes of the data to be transferred

Returns

> Error or success status returned by API

Definition at line 1164 of file flexio_uart_driver.c.

16.30.4.8   status_t FLEXIO_UART_DRV_SendDataBlocking (flexio_uart_state_t ∗ *state,* const uint8_t ∗ *txBuff,* uint32_t *txSize,* uint32_t *timeout*)

Perform a blocking UART transmission.

This function sends a block of data and only returns when the transmission is complete.

Parameters

state	Pointer to the FLEXIO_UART driver context structure.
txBuff	pointer to the data to be transferred
txSize	length in bytes of the data to be transferred
timeout	timeout for the transfer in milliseconds

Returns

> Error or success status returned by API

Definition at line 1222 of file flexio_uart_driver.c.

16.30.4.9   status_t FLEXIO_UART_DRV_SetConfig (flexio_uart_state_t ∗ *state,* uint32_t *baudRate,* uint8_t *bitCount*)

Set the baud rate and bit width for any subsequent UART communication.

This function sets the baud rate and bit width for the UART driver. Note that due to module limitation not any baud rate can be achieved. The driver will set a baud rate as close as possible to the requested baud rate, but there may still be substantial differences, for example if requesting a high baud rate while using a low-frequency FlexIO clock. The application should call FLEXIO_UART_DRV_GetBaudRate() after FLEXIO_UART_DRV_SetConfig() to check what baud rate was actually set.

Parameters

state	Pointer to the FLEXIO_UART driver context structure.
baudRate	the desired baud rate in hertz
bitCount	number of bits per word

Returns

> Error or success status returned by API

Definition at line 1070 of file flexio_uart_driver.c.

16.30.4.10   status_t FLEXIO_UART_DRV_SetRxBuffer (flexio_uart_state_t ∗ *state,* uint8_t ∗ *rxBuff,* uint32_t *rxSize*)

Provide a buffer for receiving data.

This function can be used to provide a new buffer for receiving data to the driver. It can be called from the user callback when event STATUS_UART_RX_OVERRUN is reported. This way the reception will continue without interruption.

Parameters

state	Pointer to the FLEXIO_UART driver context structure.
rxBuff	pointer to the buffer where to store received data
rxSize	length in bytes of the data to be transferred

Returns

> Error or success status returned by API

Definition at line 1415 of file flexio_uart_driver.c.

16.30.4.11 status_t FLEXIO_UART_DRV_SetTxBuffer (flexio_uart_state_t ∗ *state,* const uint8_t ∗ *txBuff,* uint32_t *txSize*)

Provide a buffer for transmitting data.

This function can be used to provide a new buffer for transmitting data to the driver. It can be called from the user callback when event STATUS_UART_TX_UNDERRUN is reported. This way the transmission will continue without interruption.

Parameters

state	Pointer to the FLEXIO_UART driver context structure.
txBuff	pointer to the buffer containing transmit data
txSize	length in bytes of the data to be transferred

Returns

> Error or success status returned by API

Definition at line 1437 of file flexio_uart_driver.c.

16.30.4.12 status_t FLEXIO_UART_DRV_TransferAbort (flexio_uart_state_t ∗ *state*)

Aborts a non-blocking UART transfer.

This function aborts a non-blocking UART transfer.

Parameters

state	Pointer to the FLEXIO_UART driver context structure.

Returns

> Error or success status returned by API

Definition at line 1344 of file flexio_uart_driver.c.

16.31 FlexTimer (FTM)

16.31.1 Detailed Description

FlexTimer Peripheral Driver.

Hardware background

The FTM of the S32K1xx is based on a 16 bits counter and supports: input capture, output compare, PWM and some instances include quadrature decoder. The main features are:

•FTM source clock is selectable (Source clock can be the system clock, the fixed frequency clock, or an external clock)

•Prescaler: 1, 2, 4, 8, 16, 32, 64, 128

•16 bit counter (up and up-down counting)

•Each channel can be configured for input capture, output compare, or PWM mode.

•Input Capture mode (single edge, dual edge or measure period/duty cycle)

•Output Compare mode (set, cleared or toggle on match)

•All channels can be configured for center-aligned PWM mode.

•Each pair of channels can be combined to generate a PWM signal with independent control of both edges of PWM signal and with dead-time insertion.

•Up to 4 fault inputs for global fault control

•Dual edge capture for pulse and period width measurement

•Quadrature decoder with input filters, relative position counting, and interrupt on position count or capture of position count on external event.

Modules

- FlexTimer Input Capture Driver (FTM_IC)

  FlexTimer Input Capture Peripheral Driver.
- FlexTimer Module Counter Driver (FTM_MC)

  FlexTimer Module Counter Peripheral Driver.
- FlexTimer Output Compare Driver (FTM_OC)

  FlexTimer Output Compare Peripheral Driver.
- FlexTimer Pulse Width Modulation Driver (FTM_PWM)

  FlexTimer Pulse Width Modulation Peripheral Driver.
- FlexTimer Quadrature Decoder Driver (FTM_QD)

  FlexTimer Quadrature Decoder Peripheral Driver.

16.32 FlexTimer Input Capture Driver (FTM_IC)

16.32.1 Detailed Description

FlexTimer Input Capture Peripheral Driver.

Hardware background

The FTM of the S32K1xx is based on a 16 bits counter and supports: input capture, output compare, PWM and some instances include quadrature decoder.

How to use FTM driver in your application

For all operation modes (without Quadrature Decoder mode) the user need to configure ftm_user_config_t. This structure will be used for initialization (FTM_DRV_Init). The next functions used are specific for each operation mode.

Single edge input capture mode

For this mode the user needs to configure parameters such: maximum counter value, number of channels, input capture operation mode (for single edge input are used edge detect mode) and edge alignment. All this information is included in the ftm_input_param_t structure.

Example:

```
/* The state structure of instance in the input capture mode */
ftm_state_t stateInputCapture;
#define FTM_IC_INSTANCE        0UL
/* Channels configuration structure for inputCapture input capture */
ftm_input_ch_param_t inputCapture_InputCaptureChannelConfig[1] =
{
    {
    0U,                               /* Channel Id */
    FTM_EDGE_DETECT,                  /* Input capture operation Mode */
    FTM_RISING_EDGE,                  /* Edge alignment Mode */
    FTM_NO_MEASUREMENT,               /* Signal measurement operation type */
    0U,                               /* Filter value */
    false,                            /* Filter disabled */
    true                              /* Continuous mode measurement */
    NULL,                             /* Vector of callbacks  parameters for channels events */
    NULL                              /* Vector of callbacks for channels events */
    }
};
/* Input capture configuration for inputCapture */
ftm_input_param_t inputCapture_InputCaptureConfig =
{
    1U,                                   /* Number of channels */
    65535U,                               /* Maximum count value */
    inputCapture_InputCaptureChannelConfig  /* Channels configuration */
};
/* Timer mode configuration for inputCapture */
/* Global configuration of inputCapture */
ftm_user_config_t  inputCapture_InitConfig =
{
    {
        false,                    /* Software trigger state */
        false,                    /* Hardware trigger 1 state */
        false,                    /* Hardware trigger 2 state */
        false,                    /* Hardware trigger 3 state */
        false,                    /* Maximum loading point state */
        false,                    /* Min loading point state */
        FTM_SYSTEM_CLOCK,         /* Update mode for INVCTRL register */
        FTM_SYSTEM_CLOCK,         /* Update mode for SWOCTRL register */
        FTM_SYSTEM_CLOCK,         /* Update mode for OUTMASK register */
        FTM_SYSTEM_CLOCK,         /* Update mode for CNTIN register */
        false,                    /* Auto clear trigger state for hardware trigger */
        FTM_UPDATE_NOW,           /* Select synchronization method */
    },
    FTM_MODE_INPUT_CAPTURE,       /* Mode of operation for FTM */
    FTM_CLOCK_DIVID_BY_4,         /* FTM clock pre-scaler */
    FTM_CLOCK_SOURCE_SYSTEMCLK,   /* FTM clock source */
    FTM_BDM_MODE_00,              /* FTM debug mode */
    false,                        /* Interrupt state */
    false                         /* Initialization trigger */
};
FTM_DRV_Init(FTM_IC_INSTANCE, &inputCapture_InitConfig, &stateInputCapture);
FTM_DRV_InitInputCapture(FTM_IC_INSTANCE, &inputCapture_InputCaptureConfig);
```

```
counter = FTM_DRV_GetInputCaptureMeasurement(FTM_IC_INSTANCE, 0UL);
```

FTM_DRV_GetInputCaptureMeasurement is now used in interrupt mode and this function is used to save time stamps in internal buffers.

Data Structures

- struct ftm_input_ch_param_t

    FlexTimer driver Input capture parameters for each channel. More...

- struct ftm_input_param_t

    FlexTimer driver input capture parameters. More...

Enumerations

- enum ftm_input_op_mode_t { FTM_EDGE_DETECT = 0U, FTM_SIGNAL_MEASUREMENT = 1U, FTM_↩ NO_OPERATION = 2U }

    FTM status.

- enum ftm_signal_measurement_mode_t {
    FTM_NO_MEASUREMENT = 0x00U, FTM_RISING_EDGE_PERIOD_MEASUREMENT = 0x01U, FTM_F↩ ALLING_EDGE_PERIOD_MEASUREMENT = 0x02U, FTM_PERIOD_ON_MEASUREMENT = 0x03U,
    FTM_PERIOD_OFF_MEASUREMENT = 0x04U }

    FlexTimer input capture measurement type for dual edge input capture.

- enum ftm_edge_alignment_mode_t { FTM_NO_PIN_CONTROL = 0x00U, FTM_RISING_EDGE = 0x01U, FTM_FALLING_EDGE = 0x02U, FTM_BOTH_EDGES = 0x03U }

    FlexTimer input capture edge mode, rising edge, or falling edge.

- enum ftm_ic_op_mode_t {
    FTM_DISABLE_OPERATION = 0x00U, FTM_TIMESTAMP_RISING_EDGE = 0x01U, FTM_TIMESTAMP↩ _FALLING_EDGE = 0x02U, FTM_TIMESTAMP_BOTH_EDGES = 0x03U,
    FTM_MEASURE_RISING_EDGE_PERIOD = 0x04U, FTM_MEASURE_FALLING_EDGE_PERIOD = 0x05U, FTM_MEASURE_PULSE_HIGH = 0x06U, FTM_MEASURE_PULSE_LOW = 0x07U }

    The measurement type for input capture mode Implements : ftm_ic_op_mode_t_Class.

Functions

- status_t FTM_DRV_InitInputCapture (uint32_t instance, const ftm_input_param_t ∗param)

    Configures Channel Input Capture for either getting time-stamps on edge detection or on signal measurement . When the edge specified in the captureMode argument occurs on the channel the FTM counter is captured into the CnV register. The user will have to read the CnV register separately to get this value. The filter function is disabled if the filterVal argument passed in is 0. The filter function is available only on channels 0,1,2,3.

- status_t FTM_DRV_DeinitInputCapture (uint32_t instance, const ftm_input_param_t ∗param)

    Disables input capture mode and clears FTM timer configuration.

- uint16_t FTM_DRV_GetInputCaptureMeasurement (uint32_t instance, uint8_t channel)

    This function is used to calculate the measurement and/or time stamps values which are read from the C(n, n+1)V registers and stored to the static buffers.

- status_t FTM_DRV_StartNewSignalMeasurement (uint32_t instance, uint8_t channel)

    Starts new single-shot signal measurement of the given channel.

- status_t FTM_IC_DRV_SetChannelMode (uint32_t instance, uint8_t channel, ftm_ic_op_mode_t inputMode, bool enableContinuousCapture)

    Set mode operation for channel in the input captue mode.

16.32.2   Data Structure Documentation

16.32.2.1   struct ftm_input_ch_param_t

FlexTimer driver Input capture parameters for each channel.

Implements : ftm_input_ch_param_t_Class

Definition at line 99 of file ftm_ic_driver.h.

Data Fields

- uint8_t hwChannelId
- ftm_input_op_mode_t inputMode
- ftm_edge_alignment_mode_t edgeAlignement
- ftm_signal_measurement_mode_t measurementType
- uint16_t filterValue
- bool filterEn
- bool continuousModeEn
- void ∗ channelsCallbacksParams
- ic_callback_t channelsCallbacks

Field Documentation

16.32.2.1.1   ic_callback_t channelsCallbacks

The callback function for channels events

Definition at line 109 of file ftm_ic_driver.h.

16.32.2.1.2   void∗ channelsCallbacksParams

The parameters of callback functions for channels events

Definition at line 108 of file ftm_ic_driver.h.

16.32.2.1.3   bool continuousModeEn

Continuous measurement state

Definition at line 107 of file ftm_ic_driver.h.

16.32.2.1.4   ftm_edge_alignment_mode_t edgeAlignement

Edge alignment Mode for signal measurement

Definition at line 103 of file ftm_ic_driver.h.

16.32.2.1.5   bool filterEn

Input capture filter state

Definition at line 106 of file ftm_ic_driver.h.

16.32.2.1.6   uint16_t filterValue

Filter Value

Definition at line 105 of file ftm_ic_driver.h.

16.32.2.1.7   uint8_t hwChannelId

Physical hardware channel ID

Definition at line 101 of file ftm_ic_driver.h.

16.32.2.1.8   ftm_input_op_mode_t inputMode

FlexTimer module mode of operation

Definition at line 102 of file ftm_ic_driver.h.

16.32.2.1.9   ftm_signal_measurement_mode_t measurementType

Measurement Mode for signal measurement

Definition at line 104 of file ftm_ic_driver.h.

16.32.2.2   struct ftm_input_param_t

FlexTimer driver input capture parameters.

Implements : ftm_input_param_t_Class

Definition at line 117 of file ftm_ic_driver.h.

Data Fields

- uint8_t nNumChannels
- uint16_t nMaxCountValue
- ftm_input_ch_param_t ∗ inputChConfig

Field Documentation

16.32.2.2.1   ftm_input_ch_param_t∗ inputChConfig

Input capture channels configuration

Definition at line 121 of file ftm_ic_driver.h.

16.32.2.2.2   uint16_t nMaxCountValue

Maximum counter value. Minimum value is 0 for this mode

Definition at line 120 of file ftm_ic_driver.h.

16.32.2.2.3   uint8_t nNumChannels

Number of input capture channel used

Definition at line 119 of file ftm_ic_driver.h.

16.32.3   Enumeration Type Documentation

16.32.3.1   enum ftm_edge_alignment_mode_t

FlexTimer input capture edge mode, rising edge, or falling edge.

Implements : ftm_edge_alignment_mode_t_Class

Enumerator

    FTM_NO_PIN_CONTROL   No trigger

    FTM_RISING_EDGE   Rising edge trigger

    FTM_FALLING_EDGE   Falling edge trigger

    FTM_BOTH_EDGES   Rising and falling edge trigger

Definition at line 70 of file ftm_ic_driver.h.

16.32.3.2   enum ftm_ic_op_mode_t

The measurement type for input capture mode Implements : ftm_ic_op_mode_t_Class.

Enumerator

    FTM_DISABLE_OPERATION Have no operation

    FTM_TIMESTAMP_RISING_EDGE Rising edge trigger

    FTM_TIMESTAMP_FALLING_EDGE Falling edge trigger

    FTM_TIMESTAMP_BOTH_EDGES Rising and falling edge trigger

    FTM_MEASURE_RISING_EDGE_PERIOD Period measurement between two consecutive rising edges

    FTM_MEASURE_FALLING_EDGE_PERIOD Period measurement between two consecutive falling edges

    FTM_MEASURE_PULSE_HIGH The time measurement taken for the pulse to remain ON or HIGH state

    FTM_MEASURE_PULSE_LOW The time measurement taken for the pulse to remain OFF or LOW state

Definition at line 82 of file ftm_ic_driver.h.

16.32.3.3   enum ftm_input_op_mode_t

FTM status.

Implements : ftm_input_op_mode_t_Class

Enumerator

    FTM_EDGE_DETECT FTM edge detect

    FTM_SIGNAL_MEASUREMENT FTM signal measurement

    FTM_NO_OPERATION FTM no operation

Definition at line 44 of file ftm_ic_driver.h.

16.32.3.4   enum ftm_signal_measurement_mode_t

FlexTimer input capture measurement type for dual edge input capture.

Implements : ftm_signal_measurement_mode_t_Class

Enumerator

    FTM_NO_MEASUREMENT No measurement

    FTM_RISING_EDGE_PERIOD_MEASUREMENT Period measurement between two consecutive rising edges

    FTM_FALLING_EDGE_PERIOD_MEASUREMENT Period measurement between two consecutive falling edges

    FTM_PERIOD_ON_MEASUREMENT The time measurement taken for the pulse to remain ON or HIGH state

    FTM_PERIOD_OFF_MEASUREMENT The time measurement taken for the pulse to remain OFF or LOW state

Definition at line 56 of file ftm_ic_driver.h.

16.32.4   Function Documentation

16.32.4.1   status_t FTM_DRV_DeinitInputCapture (uint32_t *instance,* const ftm_input_param_t ∗ *param*)

Disables input capture mode and clears FTM timer configuration.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*param*	Configuration of the output compare channel.

Returns

> success

> > • STATUS_SUCCESS : Completed successfully.

> > • STATUS_ERROR : Error occurred.

Definition at line 285 of file ftm_ic_driver.c.

16.32.4.2   uint16_t FTM_DRV_GetInputCaptureMeasurement (uint32_t *instance,* uint8_t *channel*)

This function is used to calculate the measurement and/or time stamps values which are read from the C(n, n+1)V registers and stored to the static buffers.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*channel*	For getting the time stamp of the last edge (in normal input capture) this parameter represents the channel number. For getting the last measured value (in dual edge input capture) this parameter is the lowest channel number of the pair (EX: 0, 2, 4, 6).

Returns

> value The measured value

Definition at line 337 of file ftm_ic_driver.c.

16.32.4.3   status_t FTM_DRV_InitInputCapture (uint32_t *instance,* const ftm_input_param_t ∗ *param*)

Configures Channel Input Capture for either getting time-stamps on edge detection or on signal measurement . When the edge specified in the captureMode argument occurs on the channel the FTM counter is captured into the CnV register. The user will have to read the CnV register separately to get this value. The filter function is disabled if the filterVal argument passed in is 0. The filter function is available only on channels 0,1,2,3.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*param*	Configuration of the input capture channel.

Returns

> success

> > • STATUS_SUCCESS : Completed successfully.

> > • STATUS_ERROR : Error occurred.

Definition at line 136 of file ftm_ic_driver.c.

16.32.4.4   status_t FTM_DRV_StartNewSignalMeasurement (uint32_t *instance,* uint8_t *channel*)

Starts new single-shot signal measurement of the given channel.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*channel*	Configuration of the output compare channel.

Returns

success

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 356 of file ftm_ic_driver.c.

16.32.4.5   status_t FTM_IC_DRV_SetChannelMode (uint32_t *instance,* **uint8_t** *channel,* **ftm_ic_op_mode_t** *inputMode,* **bool** *enableContinuousCapture* **)**

Set mode operation for channel in the input captue mode.

This function will change the channel mode at run time or when stopping channel. The channel mode is selected in the ftm_ic_op_mode_t enumeration type.

Parameters

in	*instance*	The input capture instance number.
in	*channel*	The channel number.
in	*inputMode*	The channel operation mode.
in	*enable↩ Continuous↩ Capture*	Enable/disable the continuous capture mode.

Returns

success

- STATUS_SUCCESS : Completed successfully.

Definition at line 394 of file ftm_ic_driver.c.

16.33   FlexTimer Module Counter Driver (FTM_MC)

16.33.1   Detailed Description

FlexTimer Module Counter Peripheral Driver.

Hardware background

The FTM of the S32K1xx is based on a 16 bits counter and supports: input capture, output compare, PWM and some instances include quadrature decoder.

How to use FTM driver in your application

For all operation modes (without Quadrature Decoder mode) the user need to configure ftm_user_config_t. This structure will be used for initialization (FTM_DRV_Init). The next functions used are specific for each operation mode.

Counter mode

For this mode the user needs to configure parameters like: counter mode (up-counting or up-down counting), maximum counter value, initial counter value. All this information is included in the ftm_timer_param_t structure.

Example:

```
/* The state structure of instance in the input capture mode */
ftm_state_t stateTimer;
#define FTM_TIMER_INSTANCE        1UL
/* Timer mode configuration for Timer */
ftm_timer_param_t Timer_TimerConfig =
{
    FTM_MODE_UP_TIMER,              /* Counter mode */
    0U,                            /* Initial counter value */
    0x8000U                        /* Final counter value */
};

/* Global configuration of Timer*/
ftm_user_config_t  Timer_InitConfig =
{
    {
        false,                     /* Software trigger state */
        false,                     /* Hardware trigger 1 state */
        false,                     /* Hardware trigger 2 state */
        false,                     /* Hardware trigger 3 state */
        false,                     /* Maximum loading point state */
        false,                     /* Min loading point state */
        FTM_SYSTEM_CLOCK,          /* Update mode for INVCTRL register */
        FTM_SYSTEM_CLOCK,          /* Update mode for SWOCTRL register */
        FTM_SYSTEM_CLOCK,          /* Update mode for OUTMASK register */
        FTM_SYSTEM_CLOCK,          /* Update mode for CNTIN register */
        false,                     /* Auto clear trigger state for hardware trigger */
        FTM_UPDATE_NOW,            /* Select synchronization method */
    },
    FTM_MODE_UP_TIMER,             /* Mode of operation for FTM */
    FTM_CLOCK_DIVID_BY_2,          /* FTM clock pre-scaler */
    FTM_CLOCK_SOURCE_SYSTEMCLK,    /* FTM clock source */
    FTM_BDM_MODE_11,               /* FTM debug mode */
    false,                         /* Interrupt state */
    false                          /* Initialization trigger */
};
FTM_DRV_Init(FTM_TIMER_INSTANCE,&Timer_InitConfig, &stateTimer);
FTM_DRV_InitCounter(FTM_TIMER_INSTANCE, &Timer_TimerConfig);
FTM_DRV_CounterStart(FTM_TIMER_INSTANCE);
```

Data Structures

- struct ftm_timer_param_t

    FlexTimer driver timer mode configuration structure. More...

Functions

- status_t FTM_DRV_InitCounter (uint32_t instance, const ftm_timer_param_t ∗timer)

Initialize the FTM counter.

- status_t FTM_DRV_CounterStart (uint32_t instance)

    Starts the FTM counter.

- status_t FTM_DRV_CounterStop (uint32_t instance)

    Stops the FTM counter.

- uint32_t FTM_DRV_CounterRead (uint32_t instance)

    Reads back the current value of the FTM counter.

16.33.2   Data Structure Documentation

16.33.2.1   struct ftm_timer_param_t

FlexTimer driver timer mode configuration structure.

Implements : ftm_timer_param_t_Class

Definition at line 44 of file ftm_mc_driver.h.

Data Fields

- ftm_config_mode_t mode
- uint16_t initialValue
- uint16_t finalValue

Field Documentation

16.33.2.1.1   uint16_t finalValue

Final counter value

Definition at line 48 of file ftm_mc_driver.h.

16.33.2.1.2   uint16_t initialValue

Initial counter value

Definition at line 47 of file ftm_mc_driver.h.

16.33.2.1.3   ftm_config_mode_t mode

FTM mode

Definition at line 46 of file ftm_mc_driver.h.

16.33.3   Function Documentation

16.33.3.1   uint32_t FTM_DRV_CounterRead (uint32_t *instance*)

Reads back the current value of the FTM counter.

Parameters

in	*instance*	The FTM peripheral instance number.

Returns

    The current counter value

Definition at line 153 of file ftm_mc_driver.c.

16.33.3.2 status_t FTM_DRV_CounterStart (uint32_t *instance*)

Starts the FTM counter.

Parameters

in	*instance*	The FTM peripheral instance number.

Returns

operation status

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 112 of file ftm_mc_driver.c.

16.33.3.3   status_t FTM_DRV_CounterStop (uint32_t *instance*)

Stops the FTM counter.

Parameters

in	*instance*	The FTM peripheral instance number.

Returns

operation status

- STATUS_SUCCESS : Completed successfully.

Definition at line 133 of file ftm_mc_driver.c.

16.33.3.4   status_t FTM_DRV_InitCounter (uint32_t *instance,* const **ftm_timer_param_t ∗ *timer*)**

Initialize the FTM counter.

Starts the FTM counter. This function provides access to the FTM counter settings. The counter can be run in Up counting and Up-down counting modes. To run the counter in Free running mode, choose Up counting option and provide 0x0 for the countStartVal and 0xFFFF for countFinalVal. Please call this function only when FTM is used as timer/counter.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*timer*	Timer configuration structure.

Returns

operation status

- STATUS_SUCCESS : Initialized successfully.

Definition at line 53 of file ftm_mc_driver.c.

16.34 FlexTimer Output Compare Driver (FTM_OC)

16.34.1 Detailed Description

FlexTimer Output Compare Peripheral Driver.

Hardware background

The FTM of the S32K1xx is based on a 16 bits counter and supports: input capture, output compare, PWM and some instances include quadrature decoder.

How to use FTM driver in your application

For all operation modes (without Quadrature Decoder mode) the user need to configure ftm_user_config_t. This structure will be used for initialization (FTM_DRV_Init). The next functions used are specific for each operation mode.

Output compare mode

For this mode the user needs to configure maximum counter value, number of channels used and output mode for each channel (toggle/clear/set on match). This information is stored in ftm_output_cmp_param_t structure type and are used in FTM_DRV_InitOutputCompare function. Next step is to set a value for comparison with the FTM_DR↩ V_UpdateOutputCompareChannel function.

Example:

```
/* The state structure of instance in the output compare mode */
ftm_state_t stateOutputCompare;
#define FTM_OUTPUT_COMPARE_INSTANCE        1UL
/* Channels configuration structure for PWM output compare */
ftm_output_cmp_ch_param_t PWM_OutputCompareChannelConfig[2] =
{
    {
        0U,                        /* Channel id */
        FTM_TOGGLE_ON_MATCH,       /* Output mode */
        10000U,                    /* Compared value */
        false,                     /* External Trigger */
    },
    {
        1U,                        /* Channel id */
        FTM_TOGGLE_ON_MATCH,       /* Output mode */
        20000U,                    /* Compared value */
        false,                     /* External Trigger */
    }
};

/* Output compare configuration for PWM */
ftm_output_cmp_param_t PWM_OutputCompareConfig =
{
    2U,                            /* Number of channels */
    FTM_MODE_OUTPUT_COMPARE,       /* FTM mode */
    40000U,                        /* Maximum count value */
    PWM_OutputCompareChannelConfig /* Channels configuration */
};
/* Timer mode configuration for PWM */
/* Global configuration of PWM */
ftm_user_config_t  PWM_InitConfig =
{
    {
        true,                      /* Software trigger state */
        false,                     /* Hardware trigger 1 state */
        false,                     /* Hardware trigger 2 state */
        false,                     /* Hardware trigger 3 state */
        true,                      /* Maximum loading point state */
        true,                      /* Min loading point state */
        FTM_SYSTEM_CLOCK,          /* Update mode for INVCTRL register */
        FTM_SYSTEM_CLOCK,          /* Update mode for SWOCTRL register */
        FTM_SYSTEM_CLOCK,          /* Update mode for OUTMASK register */
        FTM_SYSTEM_CLOCK,          /* Update mode for CNTIN register */
        false,                     /* Auto clear trigger state for hardware trigger */
        FTM_UPDATE_NOW,            /* select synchronization method */
    },
    FTM_MODE_OUTPUT_COMPARE,       /* Mode of operation for FTM */
    FTM_CLOCK_DIVID_BY_4,          /* FTM clock pre-scaler */
    FTM_CLOCK_SOURCE_SYSTEMCLK,    /* FTM clock source */
    FTM_BDM_MODE_11,               /* FTM debug mode */
```

```
    false,                          /* Interrupt state */
    false                           /* Initialization trigger */
};
FTM_DRV_Init(FTM_OUTPUT_COMPARE_INSTANCE, &PWM_InitConfig, &stateOutputCompare);
FTM_DRV_InitOutputCompare(FTM_OUTPUT_COMPARE_INSTANCE, &PWM_OutputCompareConfig);
/* If you want to change compared value */
FTM_DRV_UpdateOutputCompareChannel(FTM_OUTPUT_COMPARE_INSTANCE, 0UL, 1500
    0U );
```

Data Structures

- struct ftm_output_cmp_ch_param_t

    FlexTimer driver PWM parameters. More...

- struct ftm_output_cmp_param_t

    FlexTimer driver PWM parameters. More...

Enumerations

- enum ftm_output_compare_mode_t { FTM_DISABLE_OUTPUT = 0x00U, FTM_TOGGLE_ON_MATCH = 0x01U, FTM_CLEAR_ON_MATCH = 0x02U, FTM_SET_ON_MATCH = 0x03U }

    FlexTimer Mode configuration for output compare mode.

- enum ftm_output_compare_update_t { FTM_RELATIVE_VALUE = 0x00U, FTM_ABSOLUTE_VALUE = 0x01U }

    FlexTimer input capture type of the next output compare value.

Functions

- status_t FTM_DRV_InitOutputCompare (uint32_t instance, const ftm_output_cmp_param_t ∗param)

    Configures the FTM to generate timed pulses(Output compare mode).

- status_t FTM_DRV_DeinitOutputCompare (uint32_t instance, const ftm_output_cmp_param_t ∗param)

    Disables compare match output control and clears FTM timer configuration.

- status_t FTM_DRV_UpdateOutputCompareChannel (uint32_t instance, uint8_t channel, uint16_t next↩ComparematchValue, ftm_output_compare_update_t update, bool softwareTrigger)

    Sets the next compare match value based on the current counter value.

16.34.2   Data Structure Documentation

16.34.2.1   struct ftm_output_cmp_ch_param_t

FlexTimer driver PWM parameters.

Implements : ftm_output_cmp_ch_param_t_Class

Definition at line 68 of file ftm_oc_driver.h.

Data Fields

- uint8_t hwChannelId
- ftm_output_compare_mode_t chMode
- uint16_t comparedValue
- bool enableExternalTrigger

Field Documentation

16.34.2.1.1   ftm_output_compare_mode_t chMode

Channel output mode

Definition at line 71 of file ftm_oc_driver.h.

16.34.2.1.2   uint16_t comparedValue

The compared value

Definition at line 72 of file ftm_oc_driver.h.

16.34.2.1.3   bool enableExternalTrigger

true: enable the generation of a trigger is used for on-chip modules false: disable the generation of a trigger

Definition at line 73 of file ftm_oc_driver.h.

16.34.2.1.4   uint8_t hwChannelId

Physical hardware channel ID

Definition at line 70 of file ftm_oc_driver.h.

16.34.2.2   struct ftm_output_cmp_param_t

FlexTimer driver PWM parameters.

Implements : ftm_output_cmp_param_t_Class

Definition at line 82 of file ftm_oc_driver.h.

Data Fields

- uint8_t nNumOutputChannels
- ftm_config_mode_t mode
- uint16_t maxCountValue
- ftm_output_cmp_ch_param_t ∗ outputChannelConfig

Field Documentation

16.34.2.2.1   uint16_t maxCountValue

Maximum count value in ticks

Definition at line 86 of file ftm_oc_driver.h.

16.34.2.2.2   ftm_config_mode_t mode

FlexTimer PWM operation mode

Definition at line 85 of file ftm_oc_driver.h.

16.34.2.2.3   uint8_t nNumOutputChannels

Number of output compare channels

Definition at line 84 of file ftm_oc_driver.h.

16.34.2.2.4   ftm_output_cmp_ch_param_t∗ outputChannelConfig

Output compare channels configuration

Definition at line 87 of file ftm_oc_driver.h.

16.34.3   Enumeration Type Documentation

16.34.3.1   enum ftm_output_compare_mode_t

FlexTimer Mode configuration for output compare mode.

Implements : ftm_output_compare_mode_t_Class

Enumerator

> **FTM_DISABLE_OUTPUT** No action on output pin
>
> **FTM_TOGGLE_ON_MATCH** Toggle on match
>
> **FTM_CLEAR_ON_MATCH** Clear on match
>
> **FTM_SET_ON_MATCH** Set on match

Definition at line 44 of file ftm_oc_driver.h.

16.34.3.2   enum **ftm_output_compare_update_t**

FlexTimer input capture type of the next output compare value.

Implements : ftm_output_compare_update_t_Class

Enumerator

> **FTM_RELATIVE_VALUE** Next compared value is relative to current value
>
> **FTM_ABSOLUTE_VALUE** Next compared value is absolute

Definition at line 57 of file ftm_oc_driver.h.

16.34.4   Function Documentation

16.34.4.1   status_t FTM_DRV_DeinitOutputCompare (uint32_t *instance,* const **ftm_output_cmp_param_t** ∗ *param*)

Disables compare match output control and clears FTM timer configuration.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*param*	Configuration of the output compare channel

Returns

> success
>
> > • STATUS_SUCCESS : Completed successfully.
> >
> > • STATUS_ERROR : Error occurred.

Definition at line 108 of file ftm_oc_driver.c.

16.34.4.2   status_t FTM_DRV_InitOutputCompare (uint32_t *instance,* const **ftm_output_cmp_param_t** ∗ *param*)

Configures the FTM to generate timed pulses(Output compare mode).

When the FTM counter matches the value of CnV, the channel output is changed based on what is specified in the compareMode argument. The signal period can be modified using param->MaxCountValue. After this function max counter value and CnV are equal. FTM_DRV_SetNextComparematchValue function can be used to change CnV value.

Parameters

in	*instance*	The FTM peripheral instance number.

in	*param*	configuration of the output compare channels

Returns

success

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 45 of file ftm_oc_driver.c.

16.34.4.3   status_t FTM_DRV_UpdateOutputCompareChannel (uint32_t *instance,* uint8_t *channel,* uint16_t *nextComparematchValue,* ftm_output_compare_update_t *update,* bool *softwareTrigger*)

Sets the next compare match value based on the current counter value.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*channel*	Configuration of the output compare channel
in	*next↩ Comparematch↩ Value*	Timer value in ticks until the next compare match event should appear
in	*update*	- FTM_RELATIVE_VALUE : nextComparemantchValue will be added to current counter value - FTM_ABSOLUTE_VALUE : nextComparemantchValue will be written in counter register as it is
in	*softwareTrigger*	This parameter will be true if software trigger sync is enabled and the user want to generate a software trigger (the value from buffer will be moved to register immediate or at next loading point depending on the sync configuration). Otherwise this parameter must be false and the next compared value will be stored in buffer until a trigger signal will be received.

Returns

success

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 150 of file ftm_oc_driver.c.

16.35   FlexTimer Pulse Width Modulation Driver (FTM_PWM)

16.35.1   Detailed Description

FlexTimer Pulse Width Modulation Peripheral Driver.

Hardware background

The FTM of the S32K1xx is based on a 16 bits counter and supports: input capture, output compare, PWM and some instances include quadrature decoder.

How to use FTM driver in your application

For all operation modes (without Quadrature Decoder mode) the user need to configure ftm_user_config_t. This structure will be used for initialization (FTM_DRV_Init). The next functions used are specific for each operation mode.

PWM mode

For this mode, the user needs to configure parameters such: number of PWM channels, frequency, dead time, fault channels and duty cycle, alignment (edge or center). All this information is included in the ftm_pwm_param_t structure.

FTM_DRV_UpdatePwmChannel can be used to update duty cycles at run time. If the type of update in the duty cycle when the duty cycle can have value between 0x0 (0%) and 0x8000 (100%). If the type of update in ticks when the firstEdge and secondEdge variables can have value between 0 and ftmPeriod which is stored in the state structure.

Example:

```
/* The state structure of instance in the PWM mode */
ftm_state_t statePwm;
#define FTM_PWM_INSTANCE        1UL
/* Fault configuration structure */
ftm_pwm_fault_param_t PWM_FaultConfig =
{
    false,
    true,
    5U,                        /* Fault filter value */
    FTM_FAULT_CONTROL_MAN_EVEN,
    {
        {
            true,              /* Fault channel state (Enabled/Disabled) */
            false,             /* Fault channel filter state (Enabled/Disabled) */
            FTM_POLARITY_HIGH, /* Channel output state on fault */
        },
        {
            false,             /* Fault Channel state (Enabled/Disabled) */
            false,             /* Fault channel filter state (Enabled/Disabled) */
            FTM_POLARITY_LOW   /* Channel output state on fault */
        },
        {
            false,             /* Fault Channel state (Enabled/Disabled) */
            false,             /* Fault channel filter state (Enabled/Disabled) */
            FTM_POLARITY_LOW   /* Channel output state on fault */
        },
        {
            false,             /* Fault Channel state (Enabled/Disabled) */
            false,             /* Fault channel filter state (Enabled/Disabled) */
            FTM_POLARITY_LOW   /* Channel output state on fault */
        }
    }
};

/* Independent channels configuration structure for PWM */
ftm_independent_ch_param_t PWM_IndependentChannelsConfig[1] =
{
    {
        0U,                        /* HwChannelId */
        FTM_POLARITY_HIGH,         /* edgeMode */
        0x2500U,                   /* uDutyCyclePercent (0-0x8000) */
        false,                     /* External Trigger */
        FTM_HIGH_TRUE_PULSE,
        false,
        FTM_MAIN_DUPLICATED,
```

```
            false
        }
};

/* PWM configuration for PWM */
ftm_pwm_param_t PWM_PwmConfig =
{
    1U,                                 /* Number of independent PWM channels */
    0U,                                 /* Number of combined PWM channels */
    FTM_MODE_EDGE_ALIGNED_PWM,          /* PWM mode */
    0U,                                 /* DeadTime Value */
    FTM_DEADTIME_DIVID_BY_4,            /* DeadTime clock divider */
    7481U,                              /* PWM frequency */
    PWM_IndependentChannelsConfig,      /* Independent PWM channels configuration structure */
    NULL,                               /* Combined PWM channels configuration structure */
    &PWM_FaultConfig                    /* PWM fault configuration structure */
};

/* Timer mode configuration for PWM */
/* Global configuration of PWM */
ftm_user_config_t  PWM_InitConfig =
{
    {
        true,                           /* Software trigger state */
        false,                          /* Hardware trigger 1 state */
        false,                          /* Hardware trigger 2 state */
        false,                          /* Hardware trigger 3 state */
        true,                           /* Maximum loading point state */
        true,                           /* Min loading point state */
        FTM_SYSTEM_CLOCK,               /* Update mode for INVCTRL register */
        FTM_SYSTEM_CLOCK,               /* Update mode for SWOCTRL register */
        FTM_SYSTEM_CLOCK,               /* Update mode for OUTMASK register */
        FTM_SYSTEM_CLOCK,               /* Update mode for CNTIN register */
        false,                          /* Auto clear trigger state for hardware trigger */
        FTM_UPDATE_NOW,                 /* Select synchronization method */
    },
    FTM_MODE_EDGE_ALIGNED_PWM,          /* PWM mode */
    FTM_CLOCK_DIVID_BY_4,               /* FTM clock pre-scaler */
    FTM_CLOCK_SOURCE_SYSTEMCLK,         /* FTM clock source */
    FTM_BDM_MODE_11,                    /* FTM debug mode */
    false,                              /* Interrupt state */
    false                               /* Initialization trigger */
};
FTM_DRV_Init(FTM_PWM_INSTANCE, &PWM_InitConfig, &statePwm);
FTM_DRV_InitPwm(FTM_PWM_INSTANCE, &PWM_PwmConfig);
/* It's recommended to use softwareTrigger = true */
/* The SECOND_EDGE value is used only when PWM is used in combined mode */
FTM_DRV_UpdatePwmChannel(FTM_PWM_INSTANCE, 0UL,
        FTM_PWM_UPDATE_IN_DUTY_CYCLE, 0x800U, 0x2000U, true);
```

PWM in Modified Combine mode

For this mode the user needs to configure parameters such: number of PWM channels, frequency, dead time, fault channels and duty cycle, alignment (edge or center). All this information is included in ftm_pwm_param_t data type. The Modified Combine PWM mode is intended to support the generation of PWM signals where the period is not modified while the signal is being generated, but the duty cycle will be varied. FTM_DRV_UpdatePwmChannel can be used to update duty cycles at run time. If the type of update in the duty cycle when the duty cycle can have value between 0x0 (0%) and 0x8000 (100%). If the type of update in ticks when the firstEdge and secondEdge variables can have value between 0 and ftmPeriod which is stored in the state structure.In this mode, an even channel (n) and adjacent odd channel (n+1) are combined to generate a PWM signal in the channel (n) output. Thus, the channel (n) match edge is fixed and the channel (n+1) match edge can be varied.

Example:

```
/* The state structure of instance in the PWM mode */
ftm_state_t statePwm;
#define FTM_PWM_INSTANCE        0UL
/* Fault configuration structure */
ftm_pwm_fault_param_t PWM_FaultConfig =
{
    false,
    true,
    5U,                                 /* Fault filter value */
    FTM_FAULT_CONTROL_MAN_EVEN,
    {
        {
            true,                       /* Fault channel state (Enabled/Disabled )*/
            false,                      /* Fault channel filter state (Enabled/Disabled) */
            FTM_POLARITY_HIGH,          /* Channel output state on fault */
        },
```

```
                    {
                        false,                    /* Fault Channel state (Enabled/Disabled) */
                        false,                    /* Fault channel filter state (Enabled/Disabled) */
                        FTM_POLARITY_LOW          /* Channel output state on fault */
                    },
                    {
                        false,                    /* Fault Channel state (Enabled/Disabled) */
                        false,                    /* Fault channel filter state (Enabled/Disabled) */
                        FTM_POLARITY_LOW          /* Channel output state on fault */
                    },
                    {
                        false,                    /* Fault Channel state (Enabled/Disabled) */
                        false,                    /* Fault channel filter state (Enabled/Disabled) */
                        FTM_POLARITY_LOW          /* Channel output state on fault */
                    }
                }
};
/* Combine channels configuration structure for PWM */
ftm_combined_ch_param_t flexTimer1_CombinedChannelsConfig[1] =
{
    {
        0U,                             /* Hardware channel for channel (n) */
        512U,                           /* First edge time */
        16384U,                         /* Second edge time */
        false,                          /* Dead time enabled/disabled */
        true,                           /* The modified combine mode enabled/disabled */
        FTM_POLARITY_HIGH,              /* Channel polarity */
        true,                           /* Output enabled/disabled for channel (n+1) */
        FTM_MAIN_DUPLICATED,            /* Polarity for channel (n+1) */
        false,                          /* External Trigger on the channel (n) */
        false,                          /* External Trigger on the channel (n+1) */
        FTM_HIGH_TRUE_PULSE,            /* The selection of the channel (n) mode */
        FTM_HIGH_TRUE_PULSE,            /* The selection of the channel (n+1) mode */
    }
};
/* PWM configuration for PWM */
ftm_pwm_param_t PWM_PwmConfig =
{
    0U,                             /* Number of independent PWM channels */
    1U,                             /* Number of combined PWM channels */
    FTM_MODE_EDGE_ALIGNED_PWM,      /* PWM mode */
    0U,                             /* DeadTime Value */
    FTM_DEADTIME_DIVID_BY_4,        /* DeadTime clock divider */
    7481U,                          /* PWM frequency */
    NULL,                           /* Independent PWM channels configuration structure */
    flexTimer1_CombinedChannelsConfig, /* Combined PWM channels configuration structure */
    &PWM_FaultConfig                /* PWM fault configuration structure */
};
/* Timer mode configuration for PWM */
/* Global configuration of PWM */
ftm_user_config_t  PWM_InitConfig =
{
    {
        true,                       /* Software trigger state */
        false,                      /* Hardware trigger 1 state */
        false,                      /* Hardware trigger 2 state */
        false,                      /* Hardware trigger 3 state */
        true,                       /* Maximum loading point state */
        true,                       /* Min loading point state */
        FTM_SYSTEM_CLOCK,           /* Update mode for INVCTRL register */
        FTM_SYSTEM_CLOCK,           /* Update mode for SWOCTRL register */
        FTM_SYSTEM_CLOCK,           /* Update mode for OUTMASK register */
        FTM_SYSTEM_CLOCK,           /* Update mode for CNTIN register */
        false,                      /* Auto clear trigger state for hardware trigger */
        FTM_UPDATE_NOW,             /* Select synchronization method */
    },
    FTM_MODE_EDGE_ALIGNED_PWM,      /* PWM mode */
    FTM_CLOCK_DIVID_BY_4,           /* FTM clock pre-scaler */
    FTM_CLOCK_SOURCE_SYSTEMCLK,     /* FTM clock source */
    FTM_BDM_MODE_11,                /* FTM debug mode */
    false,                          /* Interrupt state */
    false                           /* Initialization trigger */
};
FTM_DRV_Init(FTM_PWM_INSTANCE, &PWM_InitConfig, &statePwm);
FTM_DRV_InitPwm(FTM_PWM_INSTANCE, &PWM_PwmConfig);
/* It's recommended to use softwareTrigger = true */
/* The SECOND_EDGE value is used only when PWM is used in combined mode */
FTM_DRV_UpdatePwmChannel(FTM_PWM_INSTANCE, 0UL,
        FTM_PWM_UPDATE_IN_DUTY_CYCLE, 0x0U, 0x2000U, true);
```

Data Structures

- struct ftm_pwm_ch_fault_param_t

    FlexTimer driver PWM Fault channel parameters. More...

- struct ftm_pwm_fault_param_t

    FlexTimer driver PWM Fault parameter. More...
- struct ftm_independent_ch_param_t

    FlexTimer driver independent PWM parameter. More...
- struct ftm_combined_ch_param_t

    FlexTimer driver combined PWM parameter. More...
- struct ftm_pwm_param_t

    FlexTimer driver PWM parameters. More...

Macros

- #define FTM_MAX_DUTY_CYCLE (0x8000U)

    Maximum value for PWM duty cycle.
- #define FTM_DUTY_TO_TICKS_SHIFT (15U)

    Shift value which converts duty to ticks.

Enumerations

- enum ftm_pwm_update_option_t { FTM_PWM_UPDATE_IN_DUTY_CYCLE = 0x00U, FTM_PWM_UPDA↩
  TE_IN_TICKS = 0x01U }

    FlexTimer Configure type of PWM update in the duty cycle or in ticks.
- enum ftm_polarity_t { FTM_POLARITY_LOW = 0x00U, FTM_POLARITY_HIGH = 0x01U }

    FlexTimer PWM output pulse mode, high-true or low-true on match up.
- enum ftm_second_channel_polarity_t { FTM_MAIN_INVERTED = 0x01U, FTM_MAIN_DUPLICATED = 0x00U }

    FlexTimer PWM channel (n+1) polarity for combine mode.
- enum ftm_fault_mode_t { FTM_FAULT_CONTROL_DISABLED = 0x00U, FTM_FAULT_CONTROL_MA↩
  N_EVEN = 0x01U, FTM_FAULT_CONTROL_MAN_ALL = 0x02U, FTM_FAULT_CONTROL_AUTO_ALL = 0x03U }

    FlexTimer fault control.
- enum ftm_safe_state_polarity_t { FTM_HIGH_TRUE_PULSE = 0x02U, FTM_LOW_TRUE_PULSE = 0x03U }

    Select level of the channel (n) output at the beginning.

Functions

- status_t FTM_DRV_DeinitPwm (uint32_t instance)

    Stops all PWM channels .
- status_t FTM_DRV_InitPwm (uint32_t instance, const ftm_pwm_param_t ∗param)

    Configures the duty cycle and frequency and starts outputting the PWM on all channels configured in param.
- status_t FTM_DRV_UpdatePwmChannel (uint32_t instance, uint8_t channel, ftm_pwm_update_option_↩
  t typeOfUpdate, uint16_t firstEdge, uint16_t secondEdge, bool softwareTrigger)

    This function updates the waveform output in PWM mode (duty cycle and phase).
- status_t FTM_DRV_FastUpdatePwmChannels (uint32_t instance, uint8_t numberOfChannels, const uint8_t
  ∗channels, const uint16_t ∗duty, bool softwareTrigger)

    This function will update the duty cycle of PWM output for multiple channels.
- status_t FTM_DRV_UpdatePwmPeriod (uint32_t instance, ftm_pwm_update_option_t typeOfUpdate,
  uint32_t newValue, bool softwareTrigger)

    This function will update the new period in the frequency or in the counter value into mode register which modify the period of PWM signal on the channel output.

16.35.2   Data Structure Documentation

16.35.2.1   struct ftm_pwm_ch_fault_param_t

FlexTimer driver PWM Fault channel parameters.

Implements : ftm_pwm_ch_fault_param_t_Class

Definition at line 113 of file ftm_pwm_driver.h.

Data Fields

- bool faultChannelEnabled
- bool faultFilterEnabled
- ftm_polarity_t ftmFaultPinPolarity

Field Documentation

16.35.2.1.1   bool faultChannelEnabled

Fault channel state

Definition at line 115 of file ftm_pwm_driver.h.

16.35.2.1.2   bool faultFilterEnabled

Fault channel filter state

Definition at line 116 of file ftm_pwm_driver.h.

16.35.2.1.3   ftm_polarity_t ftmFaultPinPolarity

Channel output state on fault

Definition at line 117 of file ftm_pwm_driver.h.

16.35.2.2   struct ftm_pwm_fault_param_t

FlexTimer driver PWM Fault parameter.

Implements : ftm_pwm_fault_param_t_Class

Definition at line 125 of file ftm_pwm_driver.h.

Data Fields

- bool pwmOutputStateOnFault
- bool pwmFaultInterrupt
- uint8_t faultFilterValue
- ftm_fault_mode_t faultMode
- ftm_pwm_ch_fault_param_t ftmFaultChannelParam [FTM_FEATURE_FAULT_CHANNELS]

Field Documentation

16.35.2.2.1   uint8_t faultFilterValue

Fault filter value

Definition at line 129 of file ftm_pwm_driver.h.

16.35.2.2.2   ftm_fault_mode_t faultMode

Fault mode

Definition at line 130 of file ftm_pwm_driver.h.

16.35.2.2.3   ftm_pwm_ch_fault_param_t ftmFaultChannelParam[FTM_FEATURE_FAULT_CHANNELS]

Fault channels configuration

Definition at line 131 of file ftm_pwm_driver.h.

16.35.2.2.4   bool pwmFaultInterrupt

PWM fault interrupt state

Definition at line 128 of file ftm_pwm_driver.h.

16.35.2.2.5   bool pwmOutputStateOnFault

Output pin state on fault

Definition at line 127 of file ftm_pwm_driver.h.

16.35.2.3   struct ftm_independent_ch_param_t

FlexTimer driver independent PWM parameter.

Implements : ftm_independent_ch_param_t_Class

Definition at line 139 of file ftm_pwm_driver.h.

Data Fields

- uint8_t hwChannelId
- ftm_polarity_t polarity
- uint16_t uDutyCyclePercent
- bool enableExternalTrigger
- ftm_safe_state_polarity_t levelSelect
- bool enableSecondChannelOutput
- ftm_second_channel_polarity_t secondChannelPolarity
- bool deadTime

Field Documentation

16.35.2.3.1   bool deadTime

Enable/disable dead time for channel

Definition at line 150 of file ftm_pwm_driver.h.

16.35.2.3.2   bool enableExternalTrigger

true: enable the generation of a trigger is used for on-chip modules false: disable the generation of a trigger

Definition at line 145 of file ftm_pwm_driver.h.

16.35.2.3.3   bool enableSecondChannelOutput

Enable complementary mode on next channel

Definition at line 148 of file ftm_pwm_driver.h.

16.35.2.3.4   uint8_t hwChannelId

Physical hardware channel ID

Definition at line 141 of file ftm_pwm_driver.h.

16.35.2.3.5   ftm_safe_state_polarity_t levelSelect

The selection of the channel (n) mode

Definition at line 147 of file ftm_pwm_driver.h.

16.35.2.3.6 ftm_polarity_t polarity

PWM output polarity

Definition at line 142 of file ftm_pwm_driver.h.

16.35.2.3.7 ftm_second_channel_polarity_t secondChannelPolarity

Polarity of the channel n+1 relative to channel n

Definition at line 149 of file ftm_pwm_driver.h.

16.35.2.3.8 uint16_t uDutyCyclePercent

PWM pulse width, value should be between 0 (0%) to FTM_MAX_DUTY_CYCLE (100%)

Definition at line 143 of file ftm_pwm_driver.h.

16.35.2.4 struct ftm_combined_ch_param_t

FlexTimer driver combined PWM parameter.

Implements : ftm_combined_ch_param_t_Class

Definition at line 158 of file ftm_pwm_driver.h.

Data Fields

- uint8_t hwChannelId
- uint16_t firstEdge
- uint16_t secondEdge
- bool deadTime
- bool enableModifiedCombine
- ftm_polarity_t mainChannelPolarity
- bool enableSecondChannelOutput
- ftm_second_channel_polarity_t secondChannelPolarity
- bool enableExternalTrigger
- bool enableExternalTriggerOnNextChn
- ftm_safe_state_polarity_t levelSelect
- ftm_safe_state_polarity_t levelSelectOnNextChn

Field Documentation

16.35.2.4.1 bool deadTime

Enable/disable dead time for channel

Definition at line 165 of file ftm_pwm_driver.h.

16.35.2.4.2 bool enableExternalTrigger

The generation of the channel (n) trigger true: enable the generation of a trigger on the channel (n) false: disable the generation of a trigger on the channel (n)

Definition at line 171 of file ftm_pwm_driver.h.

16.35.2.4.3 bool enableExternalTriggerOnNextChn

The generation of the channel (n+1) trigger true: enable the generation of a trigger on the channel (n+1) false: disable the generation of a trigger on the channel (n+1)

Definition at line 174 of file ftm_pwm_driver.h.

16.35.2.4.4   bool enableModifiedCombine

Enable/disable the modified combine mode for channels (n) and (n+1)

Definition at line 166 of file ftm_pwm_driver.h.

16.35.2.4.5   bool enableSecondChannelOutput

Select if channel (n+1) output is enabled/disabled for the complementary mode

Definition at line 169 of file ftm_pwm_driver.h.

16.35.2.4.6   uint16_t firstEdge

First edge time. This time is relative to signal period. The value for this parameter is between 0 and FTM_MAX_↩
DUTY_CYCLE(0 = 0% from period and FTM_MAX_DUTY_CYCLE = 100% from period)

Definition at line 161 of file ftm_pwm_driver.h.

16.35.2.4.7   uint8_t hwChannelId

Physical hardware channel ID for channel (n)

Definition at line 160 of file ftm_pwm_driver.h.

16.35.2.4.8   ftm_safe_state_polarity_t levelSelect

The selection of the channel (n) mode

Definition at line 177 of file ftm_pwm_driver.h.

16.35.2.4.9   ftm_safe_state_polarity_t levelSelectOnNextChn

The selection of the channel (n+1) mode

Definition at line 178 of file ftm_pwm_driver.h.

16.35.2.4.10   ftm_polarity_t mainChannelPolarity

Main channel polarity. For FTM_POLARITY_HIGH first output value is 0 and for FTM_POLAIRTY first output value
is 1

Definition at line 167 of file ftm_pwm_driver.h.

16.35.2.4.11   ftm_second_channel_polarity_t secondChannelPolarity

Select channel (n+1) polarity relative to channel (n)

Definition at line 170 of file ftm_pwm_driver.h.

16.35.2.4.12   uint16_t secondEdge

Second edge time. This time is relative to signal period. The value for this parameter is between 0 and FTM_MA↩
X_DUTY_CYCLE(0 = 0% from period and FTM_MAX_DUTY_CYCLE = 100% from period)

Definition at line 163 of file ftm_pwm_driver.h.

16.35.2.5   struct ftm_pwm_param_t

FlexTimer driver PWM parameters.

Implements : ftm_pwm_param_t_Class

Definition at line 186 of file ftm_pwm_driver.h.

Data Fields

- uint8_t nNumIndependentPwmChannels
- uint8_t nNumCombinedPwmChannels
- ftm_config_mode_t mode
- uint8_t deadTimeValue
- ftm_deadtime_ps_t deadTimePrescaler
- uint32_t uFrequencyHZ
- ftm_independent_ch_param_t ∗ pwmIndependentChannelConfig
- ftm_combined_ch_param_t ∗ pwmCombinedChannelConfig
- ftm_pwm_fault_param_t ∗ faultConfig

Field Documentation

16.35.2.5.1   ftm_deadtime_ps_t deadTimePrescaler

Dead time pre-scaler value[ticks]

Definition at line 192 of file ftm_pwm_driver.h.

16.35.2.5.2   uint8_t deadTimeValue

Dead time value in [ticks]

Definition at line 191 of file ftm_pwm_driver.h.

16.35.2.5.3   ftm_pwm_fault_param_t∗ faultConfig

Configuration for PWM fault

Definition at line 196 of file ftm_pwm_driver.h.

16.35.2.5.4   ftm_config_mode_t mode

FTM mode

Definition at line 190 of file ftm_pwm_driver.h.

16.35.2.5.5   uint8_t nNumCombinedPwmChannels

Number of combined PWM channels

Definition at line 189 of file ftm_pwm_driver.h.

16.35.2.5.6   uint8_t nNumIndependentPwmChannels

Number of independent PWM channels

Definition at line 188 of file ftm_pwm_driver.h.

16.35.2.5.7   ftm_combined_ch_param_t∗ pwmCombinedChannelConfig

Configuration for combined PWM channels

Definition at line 195 of file ftm_pwm_driver.h.

16.35.2.5.8   ftm_independent_ch_param_t∗ pwmIndependentChannelConfig

Configuration for independent PWM channels

Definition at line 194 of file ftm_pwm_driver.h.

16.35.2.5.9   uint32_t uFrequencyHZ

PWM period in Hz

Definition at line 193 of file ftm_pwm_driver.h.

16.35.3   Macro Definition Documentation

16.35.3.1   #define FTM_DUTY_TO_TICKS_SHIFT (15U)

Shift value which converts duty to ticks.

Definition at line 41 of file ftm_pwm_driver.h.

16.35.3.2   #define FTM_MAX_DUTY_CYCLE (0x8000U)

Maximum value for PWM duty cycle.

Definition at line 39 of file ftm_pwm_driver.h.

16.35.4   Enumeration Type Documentation

16.35.4.1   enum **ftm_fault_mode_t**

FlexTimer fault control.

Implements : ftm_fault_mode_t_Class

Enumerator

> ***FTM_FAULT_CONTROL_DISABLED*** Fault control is disabled for all channels
> ***FTM_FAULT_CONTROL_MAN_EVEN*** Fault control is enabled for even channels only (channels 0, 2, 4, and 6), and the selected mode is the manual fault clearing
> ***FTM_FAULT_CONTROL_MAN_ALL*** Fault control is enabled for all channels, and the selected mode is the manual fault clearing
> ***FTM_FAULT_CONTROL_AUTO_ALL*** Fault control is enabled for all channels, and the selected mode is the automatic fault clearing

Definition at line 83 of file ftm_pwm_driver.h.

16.35.4.2   enum **ftm_polarity_t**

FlexTimer PWM output pulse mode, high-true or low-true on match up.

Implements : ftm_polarity_t_Class

Enumerator

> ***FTM_POLARITY_LOW*** When counter > CnV output signal is LOW
> ***FTM_POLARITY_HIGH*** When counter > CnV output signal is HIGH

Definition at line 59 of file ftm_pwm_driver.h.

16.35.4.3   enum **ftm_pwm_update_option_t**

FlexTimer Configure type of PWM update in the duty cycle or in ticks.

Implements : ftm_pwm_update_option_t_Class

Enumerator

> ***FTM_PWM_UPDATE_IN_DUTY_CYCLE*** The type of PWM update in the duty cycle/pulse or also use in frequency update
> ***FTM_PWM_UPDATE_IN_TICKS*** The type of PWM update in ticks

Definition at line 48 of file ftm_pwm_driver.h.

16.35.4.4 enum ftm_safe_state_polarity_t

Select level of the channel (n) output at the beginning.

Implements : ftm_safe_state_polarity_t_Class

Enumerator

> **FTM_HIGH_TRUE_PULSE**   Clear the channel (n) output on match, (Clear the channel (n+1) output con match in combine mode and modified combine mode)
>
> **FTM_LOW_TRUE_PULSE**   Set the channel (n) output on match, (set the channel (n+1) output on match in combine mode and modified combine mode)

Definition at line 100 of file ftm_pwm_driver.h.

16.35.4.5 enum ftm_second_channel_polarity_t

FlexTimer PWM channel (n+1) polarity for combine mode.

Implements : ftm_second_channel_polarity_t_Class

Enumerator

> **FTM_MAIN_INVERTED**   The channel (n+1) output is the inverse of the channel (n) output
>
> **FTM_MAIN_DUPLICATED**   The channel (n+1) output is the same as the channel (n) output

Definition at line 70 of file ftm_pwm_driver.h.

16.35.5 Function Documentation

16.35.5.1 status_t FTM_DRV_DeinitPwm (uint32_t *instance*)

Stops all PWM channels .

Parameters

in	*instance*	The FTM peripheral instance number.

Returns

> counter the current counter value

Definition at line 280 of file ftm_pwm_driver.c.

16.35.5.2 status_t FTM_DRV_FastUpdatePwmChannels (uint32_t *instance,* uint8_t *numberOfChannels,* const uint8_t ∗ *channels,* const uint16_t ∗ *duty,* bool *softwareTrigger*)

This function will update the duty cycle of PWM output for multiple channels.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*numberOf↩ Channels*	The number of channels which should be updated.
in	*channels*	The list of channels which should be updated.
in	*duty*	The list of duty cycles for selected channels.

in	*softwareTrigger*	If true a software trigger is generate to update PWM parameters.

Returns

> success
>
>> • STATUS_SUCCESS : Completed successfully.
>>
>> • STATUS_ERROR : Error occurred.

Definition at line 539 of file ftm_pwm_driver.c.

16.35.5.3   status_t FTM_DRV_InitPwm (uint32_t *instance,* const **ftm_pwm_param_t ∗ *param*)**

Configures the duty cycle and frequency and starts outputting the PWM on all channels configured in param.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*param*	FTM driver PWM parameter to configure PWM options.

Returns

> success
>
>> • STATUS_SUCCESS : Completed successfully.
>>
>> • STATUS_ERROR : Error occurred.

Definition at line 43 of file ftm_pwm_driver.c.

16.35.5.4   status_t FTM_DRV_UpdatePwmChannel (uint32_t *instance,* uint8_t *channel,* ftm_pwm_update_option_t
**   *typeOfUpdate,* uint16_t *firstEdge,* uint16_t *secondEdge,* bool *softwareTrigger*)**

This function updates the waveform output in PWM mode (duty cycle and phase).

Parameters

in	*instance*	The FTM peripheral instance number.
in	*channel*	The channel number. In combined mode, the code finds the channel.
in	*typeOfUpdate*	The type of PWM update in the duty cycle/pulse or in ticks.
in	*firstEdge*	Duty cycle or first edge time for PWM mode. Can take value between 0 - F↩TM_MAX_DUTY_CYCLE(0 = 0% from period and FTM_MAX_DUTY_CYCLE = 100% from period) Or value in ticks for the first of the PWM mode in which can have value between 0 and ftmPeriod is stored in the state structure.
in	*secondEdge*	Second edge time - only for combined mode. Can take value between 0 - F↩TM_MAX_DUTY_CYCLE(0 = 0% from period and FTM_MAX_DUTY_CYCLE = 100% from period). Or value in ticks for the second of the PWM mode in which can have value between 0 and ftmPeriod is stored in the state structure.
in	*softwareTrigger*	If true a software trigger is generate to update PWM parameters.

Returns

> success
>
>> • STATUS_SUCCESS : Completed successfully.
>>
>> • STATUS_ERROR : Error occurred.

Definition at line 344 of file ftm_pwm_driver.c.

16.35.5.5   status_t FTM_DRV_UpdatePwmPeriod (uint32_t *instance,* ftm_pwm_update_option_t *typeOfUpdate,* uint32_t
**   *newValue,* bool *softwareTrigger*)**

This function will update the new period in the frequency or in the counter value into mode register which modify the period of PWM signal on the channel output.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*typeOfUpdate*	The type of PWM update is a period in Hz or in ticks. • For FTM_PWM_UPDATE_IN_DUTY_CYCLE which reuse in FTM_D←↩ RV_UpdatePwmChannel function will update in Hz. • For FTM_PWM_UPDATE_IN_TICKS will update in ticks.
in	*newValue*	The frequency or the counter value which will select with modified value for PWM signal. If the type of update in the duty cycle, the newValue parameter must be value between 1U and maximum is the frequency of the FTM counter. If the type of update in ticks, the newValue parameter must be value between 1U and 0xFFFFU.
in	*softwareTrigger*	If true a software trigger is generate to update PWM parameters.

Returns

operation status

- STATUS_SUCCESS : Completed successfully.

- STATUS_ERROR : Error occurred.

Definition at line 473 of file ftm_pwm_driver.c.

16.36   FlexTimer Quadrature Decoder Driver (FTM_QD)

16.36.1   Detailed Description

FlexTimer Quadrature Decoder Peripheral Driver.

Hardware background

The FTM of the S32K1xx is based on a 16 bits counter and supports: input capture, output compare, PWM and some instances include quadrature decoder.

How to use FTM driver in your application

For all operation modes (without Quadrature Decoder mode) the user need to configure ftm_user_config_t. This structure will be used for initialization (FTM_DRV_Init). The next functions used are specific for each operation mode.

Quadrature decoder mode

For this mode the user needs to configure parameters like: maximum counter value, initial counter value, mode (Count and Direction Encoding mode), and for both input phases polarity and filtering. All this information is included in ftm_quad_decode_config_t. In this mode, the counter is clocked by the phase A and phase B. The current state of the decoder can be obtained using FTM_DRV_QuadGetState.

Hardware limitation:

In count and direction mode if initial value of the PHASE_A is HIGH the counter will be incremented.

Example:

```
/* The state structure of instance in the quadrature mode */
ftm_state_t stateQuad;
#define FTM_QUADRATURE_INSTANCE    1UL
ftm_quad_decoder_state_t quadra_state;
ftm_quad_decode_config_t quadrature_decoder_configuration =
{
    FTM_QUAD_COUNT_AND_DIR,        /* Quadrature decoder mode */
    0U,                            /* Initial counter value */
    32500U,                        /* Maximum counter value */
    {
        false,                     /* Filter state */
        0U,                        /* Filter value */
        FTM_QUAD_PHASE_NORMAL      /* Phase polarity */
    },
    {
        false,                     /* Filter state */
        0U,                        /* Filter value */
        FTM_QUAD_PHASE_NORMAL      /* Phase polarity */
    }
};
/* Timer mode configuration for Quadrature */
/* Global configuration of Quadrature */
ftm_user_config_t  Quadrature_InitConfig =
{
    {
        false,                        /* Software trigger state */
        false,                        /* Hardware trigger 1 state */
        false,                        /* Hardware trigger 2 state */
        false,                        /* Hardware trigger 3 state */
        false,                        /* Maximum loading point state */
        false,                        /* Min loading point state */
        FTM_SYSTEM_CLOCK,             /* Update mode for INVCTRL register */
        FTM_SYSTEM_CLOCK,             /* Update mode for SWOCTRL register */
        FTM_SYSTEM_CLOCK,             /* Update mode for OUTMASK register */
        FTM_SYSTEM_CLOCK,             /* Update mode for CNTIN register */
        false,                        /* Auto clear trigger state for hardware trigger */
        FTM_UPDATE_NOW,               /* Select synchronization method */
    },
    FTM_MODE_QUADRATURE_DECODER,   /* Mode of operation for FTM */
    FTM_CLOCK_DIVID_BY_2,          /* FTM clock pre-scaler */
    FTM_CLOCK_SOURCE_SYSTEMCLK,    /* FTM clock source */
    FTM_BDM_MODE_11,               /* FTM debug mode */
    false,                         /* Interrupt state */
    false                          /* Initialization trigger */
};
```

```
FTM_DRV_Init(FTM_QUADRATURE_INSTANCE, &Quadrature_InitConfig, &stateQuad);
FTM_DRV_QuadDecodeStart(FTM_QUADRATURE_INSTANCE, &quadrature_decoder_configuration);
quadra_state = FTM_DRV_QuadGetState(FTM_QUADRATURE_INSTANCE);
```

Data Structures

- struct ftm_phase_params_t

    FlexTimer quadrature decoder channel parameters. More...

- struct ftm_quad_decode_config_t

    FTM quadrature configure structure. More...

- struct ftm_quad_decoder_state_t

    FTM quadrature state(counter value and flags) More...

Functions

- status_t FTM_DRV_QuadDecodeStart (uint32_t instance, const ftm_quad_decode_config_t ∗config)

    Configures the quadrature mode and starts measurement.

- status_t FTM_DRV_QuadDecodeStop (uint32_t instance)

    De-activates the quadrature decode mode.

- ftm_quad_decoder_state_t FTM_DRV_QuadGetState (uint32_t instance)

    Return the current quadrature decoder state (counter value, overflow flag and overflow direction)

16.36.2   Data Structure Documentation

16.36.2.1   struct ftm_phase_params_t

FlexTimer quadrature decoder channel parameters.

Implements : ftm_phase_params_t_Class

Definition at line 44 of file ftm_qd_driver.h.

Data Fields

- bool phaseInputFilter
- uint8_t phaseFilterVal
- ftm_quad_phase_polarity_t phasePolarity

Field Documentation

16.36.2.1.1   uint8_t phaseFilterVal

Filter value (if input filter is enabled)

Definition at line 48 of file ftm_qd_driver.h.

16.36.2.1.2   bool phaseInputFilter

True: disable phase filter, False: enable phase filter

Definition at line 46 of file ftm_qd_driver.h.

16.36.2.1.3   ftm_quad_phase_polarity_t phasePolarity

Phase polarity

Definition at line 49 of file ftm_qd_driver.h.

16.36.2.2   struct ftm_quad_decode_config_t

FTM quadrature configure structure.

Implements : ftm_quad_decode_config_t_Class

Definition at line 57 of file ftm_qd_driver.h.

Data Fields

- ftm_quad_decode_mode_t mode
- uint16_t initialVal
- uint16_t maxVal
- ftm_phase_params_t phaseAConfig
- ftm_phase_params_t phaseBConfig

Field Documentation

16.36.2.2.1   uint16_t initialVal

Initial counter value

Definition at line 60 of file ftm_qd_driver.h.

16.36.2.2.2   uint16_t maxVal

Maximum counter value

Definition at line 61 of file ftm_qd_driver.h.

16.36.2.2.3   ftm_quad_decode_mode_t mode

FTM_QUAD_PHASE_ENCODE or FTM_QUAD_COUNT_AND_DIR

Definition at line 59 of file ftm_qd_driver.h.

16.36.2.2.4   ftm_phase_params_t phaseAConfig

Configuration for the input phase a

Definition at line 62 of file ftm_qd_driver.h.

16.36.2.2.5   ftm_phase_params_t phaseBConfig

Configuration for the input phase b

Definition at line 63 of file ftm_qd_driver.h.

16.36.2.3   struct ftm_quad_decoder_state_t

FTM quadrature state(counter value and flags)

Implements : ftm_quad_decoder_state_t_Class

Definition at line 71 of file ftm_qd_driver.h.

Data Fields

- uint16_t counter
- bool overflowFlag
- bool overflowDirection
- bool counterDirection

Field Documentation

16.36.2.3.1 uint16_t counter

Counter value

Definition at line 73 of file ftm_qd_driver.h.

16.36.2.3.2 bool counterDirection

False FTM counter is decreasing, True FTM counter is increasing

Definition at line 78 of file ftm_qd_driver.h.

16.36.2.3.3 bool overflowDirection

False if overflow occurred at minimum value, True if overflow occurred at maximum value

Definition at line 76 of file ftm_qd_driver.h.

16.36.2.3.4 bool overflowFlag

True if overflow occurred, False if overflow doesn't occurred

Definition at line 74 of file ftm_qd_driver.h.

16.36.3 Function Documentation

16.36.3.1 status_t FTM_DRV_QuadDecodeStart (uint32_t *instance,* const **ftm_quad_decode_config_t** ∗ *config*)

Configures the quadrature mode and starts measurement.

Parameters

| in | *instance* | Instance number of the FTM module. |
| in | *config* | Configuration structure(quadrature decode mode, polarity for both phases, initial and maximum value for the counter, filter configuration). |

Returns

success

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 50 of file ftm_qd_driver.c.

16.36.3.2 status_t FTM_DRV_QuadDecodeStop (uint32_t *instance*)

De-activates the quadrature decode mode.

Parameters

| in | *instance* | Instance number of the FTM module. |

Returns

success

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 109 of file ftm_qd_driver.c.

16.36.3.3 ftm_quad_decoder_state_t FTM_DRV_QuadGetState (uint32_t *instance*)

Return the current quadrature decoder state (counter value, overflow flag and overflow direction)

Parameters

in	*instance*	Instance number of the FTM module.

Returns

    The current state of quadrature decoder

Definition at line 130 of file ftm_qd_driver.c.

16.37 Flexible I/O (FlexIO)

16.37.1 Detailed Description

The FlexIO is a highly configurable module providing a wide range of functionality including:

- Emulation of a variety of serial communication protocols, such as SPI, I2C, I2S or UART, while requiring low CPU overhead and being more efficient that having multiple dedicated peripherals for each protocol.

- Flexible 16-bit timers with support for a variety of trigger, reset, enable and disable conditions

- PWM/Waveform generation

Several drivers are provided for this device, implementing a variety of communication protocols. There is also a common layer on which all the drivers are based, allowing more driver instances, either of the same type or different types, to function in parallel on the same FlexIO device. Each driver instance needs a certain number of Flex↩ IO resources (shifters and timers) and as long as there are enough free resources new driver instances can be initialized. The table below shows the driver types and the number of resources needed by each one:

Drivers	Timers	Shifters	Pins
SPI	2	2	4
I2C	2	2	2
I2S	2	2	4
UART	1	1	1

The number of timers and shifters available on a specific device can be found in the reference manual.

Modules

- FlexIO Common Driver

    Common services for FlexIO drivers.
- FlexIO I2C Driver

    I2C communication over FlexIO module (FLEXIO_I2C)
- FlexIO I2S Driver

    I2S communication over FlexIO module (FLEXIO_I2S)
- FlexIO SPI Driver

    SPI communication over FlexIO module (FLEXIO_SPI)
- FlexIO UART Driver

    UART communication over FlexIO module (FLEXIO_UART)

16.38 Ftm_common

16.38.1 Detailed Description

Data Structures

- struct ftm_state_t

    FlexTimer state structure of the driver. More...

- struct ftm_pwm_sync_t

    FlexTimer Registers sync parameters Please don't use software and hardware trigger simultaneously Implements : ftm_pwm_sync_t_Class. More...

- struct ftm_user_config_t

    Configuration structure that the user needs to set. More...

Macros

- #define FTM_RMW_SC(base, mask, value) (((base)->SC) = (((((base)->SC) & ~(mask)) | (value)))

    FTM_SC - Read and modify and write to Status And Control (RW)

- #define FTM_RMW_CNT(base, mask, value) (((base)->CNT) = (((((base)->CNT) & ~(mask)) | (value)))

    FTM_CNT - Read and modify and write to Counter (RW)

- #define FTM_RMW_MOD(base, mask, value) (((base)->MOD) = (((((base)->MOD) & ~(mask)) | (value)))

    FTM_MOD - Read and modify and write Modulo (RW)

- #define FTM_RMW_CNTIN(base, mask, value) (((base)->CNTIN) = (((((base)->CNTIN) & ~(mask)) | (value)))

    FTM_CNTIN - Read and modify and write Counter Initial Value (RW)

- #define FTM_RMW_STATUS(base, mask, value) (((base)->STATUS) = (((((base)->STATUS) & ~(mask)) | (value)))

    FTM_STATUS - Read and modify and write Capture And Compare Status (RW)

- #define FTM_RMW_MODE(base, mask, value) (((base)->MODE) = (((((base)->MODE) & ~(mask)) | (value)))

    FTM_MODE - Read and modify and write Counter Features Mode Selection (RW)

- #define FTM_RMW_CnSCV_REG(base, channel, mask, value) (((base)->CONTROLS[channel].CnSC) = (((((base)->CONTROLS[channel].CnSC) & ~(mask)) | (value)))

    FTM_CnSCV - Read and modify and write Channel (n) Status And Control (RW)

- #define FTM_RMW_DEADTIME(base, mask, value) (((base)->DEADTIME) = (((((base)->DEADTIME) & ~(mask)) | (value)))

    FTM_DEADTIME - Read and modify and write Dead-time Insertion Control (RW)

- #define FTM_RMW_EXTTRIG_REG(base, mask, value) (((base)->EXTTRIG) = (((((base)->EXTTRIG) & ~(mask)) | (value)))

    FTM_EXTTRIG - Read and modify and write External Trigger Control (RW)

- #define FTM_RMW_FLTCTRL(base, mask, value) (((base)->FLTCTRL) = (((((base)->FLTCTRL) & ~(mask)) | (value)))

    FTM_FLTCTRL - Read and modify and write Fault Control (RW)

- #define FTM_RMW_FMS(base, mask, value) (((base)->FMS) = (((((base)->FMS) & ~(mask)) | (value)))

    FTM_FMS - Read and modify and write Fault Mode Status (RW)

- #define FTM_RMW_CONF(base, mask, value) (((base)->CONF) = (((((base)->CONF) & ~(mask)) | (value)))

    FTM_CONF - Read and modify and write Configuration (RW)

- #define FTM_RMW_POL(base, mask, value) (((base)->POL) = (((((base)->POL) & ~(mask)) | (value)))

    POL - Read and modify and write Polarity (RW)

- #define FTM_RMW_FILTER(base, mask, value) (((base)->FILTER) = (((((base)->FILTER) & ~(mask)) | (value)))

    FILTER - Read and modify and write Filter (RW)

- #define FTM_RMW_SYNC(base, mask, value) (((base)->SYNC) = (((((base)->SYNC) & ~(mask)) | (value)))

SYNC - Read and modify and write Synchronization (RW)

- #define FTM_RMW_QDCTRL(base, mask, value) (((base)->QDCTRL) = ((((base)->QDCTRL) & ~(mask)) | (value)))

    QDCTRL - Read and modify and write Quadrature Decoder Control And Status (RW)

- #define FTM_RMW_PAIR0DEADTIME(base, mask, value) (((base)->PAIR0DEADTIME) = ((((base)->PA↩IR0DEADTIME) & ~(mask)) | (value)))

    FTM_PAIR0DEADTIME - Read and modify and write Dead-time Insertion Control for the pair 0 (RW)

- #define FTM_RMW_PAIR1DEADTIME(base, mask, value) (((base)->PAIR1DEADTIME) = ((((base)->PA↩IR1DEADTIME) & ~(mask)) | (value)))

    FTM_PAIR1DEADTIME - Read and modify and write Dead-time Insertion Control for the pair 1 (RW)

- #define FTM_RMW_PAIR2DEADTIME(base, mask, value) (((base)->PAIR2DEADTIME) = ((((base)->PA↩IR2DEADTIME) & ~(mask)) | (value)))

    FTM_PAIR2DEADTIME - Read and modify and write Dead-time Insertion Control for the pair 2 (RW)

- #define FTM_RMW_PAIR3DEADTIME(base, mask, value) (((base)->PAIR3DEADTIME) = ((((base)->PA↩IR3DEADTIME) & ~(mask)) | (value)))

    FTM_PAIR3DEADTIME - Read and modify and write Dead-time Insertion Control for the pair 3 (RW)

- #define CHAN0_IDX (0U)

    Channel number for CHAN1.

- #define CHAN1_IDX (1U)

    Channel number for CHAN2.

- #define CHAN2_IDX (2U)

    Channel number for CHAN3.

- #define CHAN3_IDX (3U)

    Channel number for CHAN4.

- #define CHAN4_IDX (4U)

    Channel number for CHAN5.

- #define CHAN5_IDX (5U)

    Channel number for CHAN6.

- #define CHAN6_IDX (6U)

    Channel number for CHAN7.

- #define CHAN7_IDX (7U)

Enumerations

- enum ftm_config_mode_t {
  FTM_MODE_NOT_INITIALIZED = 0x00U, FTM_MODE_INPUT_CAPTURE = 0x01U, FTM_MODE_OUT↩PUT_COMPARE = 0x02U, FTM_MODE_EDGE_ALIGNED_PWM = 0x03U,
  FTM_MODE_CEN_ALIGNED_PWM = 0x04U, FTM_MODE_QUADRATURE_DECODER = 0x05U, FTM_↩MODE_UP_TIMER = 0x06U, FTM_MODE_UP_DOWN_TIMER = 0x07U }

    FlexTimer operation mode.

- enum ftm_clock_source_t { FTM_CLOCK_SOURCE_NONE = 0x00U, FTM_CLOCK_SOURCE_SYSTEM↩CLK = 0x01U, FTM_CLOCK_SOURCE_FIXEDCLK = 0x02U, FTM_CLOCK_SOURCE_EXTERNALCLK = 0x03U }

    FlexTimer clock source selection.

- enum ftm_clock_ps_t {
  FTM_CLOCK_DIVID_BY_1 = 0x00U, FTM_CLOCK_DIVID_BY_2 = 0x01U, FTM_CLOCK_DIVID_BY_4 = 0x02U, FTM_CLOCK_DIVID_BY_8 = 0x03U,
  FTM_CLOCK_DIVID_BY_16 = 0x04U, FTM_CLOCK_DIVID_BY_32 = 0x05U, FTM_CLOCK_DIVID_BY_64 = 0x06U, FTM_CLOCK_DIVID_BY_128 = 0x07U }

    FlexTimer pre-scaler factor selection for the clock source. In quadrature decoder mode set FTM_CLOCK_DIVID_↩BY_1.

- enum ftm_interrupt_option_t {
  FTM_CHANNEL0_INT_ENABLE = 0x00000001U, FTM_CHANNEL1_INT_ENABLE = 0x00000002U, FT↩
  M_CHANNEL2_INT_ENABLE = 0x00000004U, FTM_CHANNEL3_INT_ENABLE = 0x00000008U,
  FTM_CHANNEL4_INT_ENABLE = 0x00000010U, FTM_CHANNEL5_INT_ENABLE = 0x00000020U, FT↩
  M_CHANNEL6_INT_ENABLE = 0x00000040U, FTM_CHANNEL7_INT_ENABLE = 0x00000080U,
  FTM_FAULT_INT_ENABLE = 0x00000100U, FTM_TIME_OVER_FLOW_INT_ENABLE = 0x00000200↩
  U, FTM_RELOAD_INT_ENABLE = 0x00000400U }

    List of FTM interrupts.

- enum ftm_status_flag_t {
  FTM_CHANNEL0_FLAG = 0x00000001U, FTM_CHANNEL1_FLAG = 0x00000002U, FTM_CHANNEL2_↩
  FLAG = 0x00000004U, FTM_CHANNEL3_FLAG = 0x00000008U,
  FTM_CHANNEL4_FLAG = 0x00000010U, FTM_CHANNEL5_FLAG = 0x00000020U, FTM_CHANNEL6_↩
  FLAG = 0x00000040U, FTM_CHANNEL7_FLAG = 0x00000080U,
  FTM_FAULT_FLAG = 0x00000100U, FTM_TIME_OVER_FLOW_FLAG = 0x00000200U, FTM_RELOAD↩
  _FLAG = 0x00000400U, FTM_CHANNEL_TRIGGER_FLAG = 0x00000800U }

    List of FTM flags.

- enum ftm_reg_update_t { FTM_SYSTEM_CLOCK = 0U, FTM_PWM_SYNC = 1U }

    FTM sync source.

- enum ftm_pwm_sync_mode_t { FTM_WAIT_LOADING_POINTS = 0U, FTM_UPDATE_NOW = 1U }

    FTM update register.

- enum ftm_deadtime_ps_t { FTM_DEADTIME_DIVID_BY_1 = 0x01U, FTM_DEADTIME_DIVID_BY_4 =
  0x02U, FTM_DEADTIME_DIVID_BY_16 = 0x03U }

    FlexTimer pre-scaler factor for the dead-time insertion.

- enum ftm_quad_decode_mode_t { FTM_QUAD_PHASE_ENCODE = 0x00U, FTM_QUAD_COUNT_AND↩
  _DIR = 0x01U }

    FlexTimer quadrature decode modes, phase encode or count and direction mode.

- enum ftm_quad_phase_polarity_t { FTM_QUAD_PHASE_NORMAL = 0x00U, FTM_QUAD_PHASE_INVE↩
  RT = 0x01U }

    FlexTimer quadrature phase polarities, normal or inverted polarity.

- enum ftm_bdm_mode_t { FTM_BDM_MODE_00 = 0x00U, FTM_BDM_MODE_01 = 0x01U, FTM_BDM_M↩
  ODE_10 = 0x02U, FTM_BDM_MODE_11 = 0x03U }

    Options for the FlexTimer behavior in BDM Mode.

Functions

- static void FTM_DRV_SetClockFilterPs (FTM_Type ∗const ftmBase, uint8_t filterPrescale)

    Sets the filter Pre-scaler divider.

- static uint8_t FTM_DRV_GetClockFilterPs (const FTM_Type ∗ftmBase)

    Reads the FTM filter clock divider.

- static uint16_t FTM_DRV_GetCounter (const FTM_Type ∗ftmBase)

    Returns the FTM peripheral current counter value.

- static uint16_t FTM_DRV_GetMod (const FTM_Type ∗ftmBase)

    Returns the FTM peripheral counter modulo value.

- static uint16_t FTM_DRV_GetCounterInitVal (const FTM_Type ∗ftmBase)

    Returns the FTM peripheral counter initial value.

- static void FTM_DRV_ClearChSC (FTM_Type ∗const ftmBase, uint8_t channel)

    Clears the content of Channel (n) Status And Control.

- static uint8_t FTM_DRV_GetChnEdgeLevel (const FTM_Type ∗ftmBase, uint8_t channel)

    Gets the FTM peripheral timer channel edge level.

- static void FTM_DRV_SetChnIcrstCmd (FTM_Type ∗const ftmBase, uint8_t channel, bool enable)

    Configure the feature of FTM counter reset by the selected input capture event.

- static bool FTM_DRV_IsChnIcrst (const FTM_Type ∗ftmBase, uint8_t channel)

    Returns whether the FTM FTM counter is reset.

- static void FTM_DRV_SetChnDmaCmd (FTM_Type ∗const ftmBase, uint8_t channel, bool enable)

    Enables or disables the FTM peripheral timer channel DMA.
- static bool FTM_DRV_IsChnDma (const FTM_Type ∗ftmBase, uint8_t channel)

    Returns whether the FTM peripheral timer channel DMA is enabled.
- static void FTM_DRV_SetTrigModeControlCmd (FTM_Type ∗const ftmBase, uint8_t channel, bool enable)

    Enables or disables the trigger generation on FTM channel outputs.
- static bool FTM_DRV_GetTriggerControled (const FTM_Type ∗ftmBase, uint8_t channel)

    Returns whether the trigger mode is enabled.
- static bool FTM_DRV_GetChInputState (const FTM_Type ∗ftmBase, uint8_t channel)

    Get the state of channel input.
- static bool FTM_DRV_GetChOutputValue (const FTM_Type ∗ftmBase, uint8_t channel)

    Get the value of channel output.
- static uint16_t FTM_DRV_GetChnCountVal (const FTM_Type ∗ftmBase, uint8_t channel)

    Gets the FTM peripheral timer channel counter value.
- static bool FTM_DRV_GetChnEventStatus (const FTM_Type ∗ftmBase, uint8_t channel)

    Gets the FTM peripheral timer channel event status.
- static uint32_t FTM_DRV_GetEventStatus (const FTM_Type ∗ftmBase)

    Gets the FTM peripheral timer status info for all channels.
- static void FTM_DRV_ClearChnEventStatus (FTM_Type ∗const ftmBase, uint8_t channel)

    Clears the FTM peripheral timer all channel event status.
- static void FTM_DRV_SetChnOutputMask (FTM_Type ∗const ftmBase, uint8_t channel, bool mask)

    Sets the FTM peripheral timer channel output mask.
- static void FTM_DRV_SetChnOutputInitStateCmd (FTM_Type ∗const ftmBase, uint8_t channel, bool state)

    Sets the FTM peripheral timer channel output initial state 0 or 1.
- static void FTM_DRV_DisableFaultInt (FTM_Type ∗const ftmBase)

    Disables the FTM peripheral timer fault interrupt.
- static void FTM_DRV_SetCaptureTestCmd (FTM_Type ∗const ftmBase, bool enable)

    Enables or disables the FTM peripheral timer capture test mode.
- static bool FTM_DRV_IsFtmEnable (const FTM_Type ∗ftmBase)

    Get status of the FTMEN bit in the FTM_MODE register.
- static void FTM_DRV_SetCountReinitSyncCmd (FTM_Type ∗const ftmBase, bool enable)

    Determines if the FTM counter is re-initialized when the selected trigger for synchronization is detected.
- static bool FTM_DRV_IsWriteProtectionEnabled (const FTM_Type ∗ftmBase)

    Checks whether the write protection is enabled.
- static bool FTM_DRV_IsFaultInputEnabled (const FTM_Type ∗ftmBase)

    Checks whether the logic OR of the fault inputs is enabled.
- static bool FTM_DRV_IsFaultFlagDetected (const FTM_Type ∗ftmBase, uint8_t channel)

    Checks whether a fault condition is detected at the fault input.
- static void FTM_DRV_ClearFaultFlagDetected (FTM_Type ∗const ftmBase, uint8_t channel)

    Clear a fault condition is detected at the fault input.
- static void FTM_DRV_SetDualChnInvertCmd (FTM_Type ∗const ftmBase, uint8_t chnlPairNum, bool enable)

    Enables or disables the channel invert for a channel pair.
- static void FTM_DRV_SetChnSoftwareCtrlCmd (FTM_Type ∗const ftmBase, uint8_t channel, bool enable)

    Enables or disables the channel software output control.
- static void FTM_DRV_SetChnSoftwareCtrlVal (FTM_Type ∗const ftmBase, uint8_t channel, bool enable)

    Sets the channel software output control value.
- static void FTM_DRV_SetGlobalLoadCmd (FTM_Type ∗const ftmBase)

    Set the global load mechanism.
- static void FTM_DRV_SetLoadCmd (FTM_Type ∗const ftmBase, bool enable)

    Enable the global load.
- static void FTM_DRV_SetHalfCycleCmd (FTM_Type ∗const ftmBase, bool enable)

Enable the half cycle reload.

- static void FTM_DRV_SetPwmLoadCmd (FTM_Type ∗const ftmBase, bool enable)

    Enables or disables the loading of MOD, CNTIN and CV with values of their write buffer.

- static void FTM_DRV_SetPwmLoadChnSelCmd (FTM_Type ∗const ftmBase, uint8_t channel, bool enable)

    Includes or excludes the channel in the matching process.

- static void FTM_DRV_SetInitTrigOnReloadCmd (FTM_Type ∗const ftmBase, bool enable)

    Enables or disables the FTM initialization trigger on Reload Point.

- static void FTM_DRV_SetGlobalTimeBaseOutputCmd (FTM_Type ∗const ftmBase, bool enable)

    Enables or disables the FTM global time base signal generation to other FTM's.

- static void FTM_DRV_SetGlobalTimeBaseCmd (FTM_Type ∗const ftmBase, bool enable)

    Enables or disables the FTM timer global time base.

- static void FTM_DRV_SetLoadFreq (FTM_Type ∗const ftmBase, uint8_t val)

    Sets the FTM timer TOF Frequency.

- static void FTM_DRV_SetExtPairDeadtimeValue (FTM_Type ∗const ftmBase, uint8_t channelPair, uint8_↩
t value)

    Sets the FTM extended dead-time value for the channel pair.

- static void FTM_DRV_SetPairDeadtimePrescale (FTM_Type ∗const ftmBase, uint8_t channelPair, ftm_↩
deadtime_ps_t divider)

    Sets the FTM dead time divider for the channel pair.

- static void FTM_DRV_SetPairDeadtimeCount (FTM_Type ∗const ftmBase, uint8_t channelPair, uint8_t count)

    Sets the FTM dead-time value for the channel pair.

- status_t FTM_DRV_Init (uint32_t instance, const ftm_user_config_t ∗info, ftm_state_t ∗state)

    Initializes the FTM driver.

- status_t FTM_DRV_Deinit (uint32_t instance)

    Shuts down the FTM driver.

- status_t FTM_DRV_MaskOutputChannels (uint32_t instance, uint32_t channelsMask, bool softwareTrigger)

    This function will mask the output of the channels and at match events will be ignored by the masked channels.

- status_t FTM_DRV_SetInitialCounterValue (uint32_t instance, uint16_t counterValue, bool softwareTrigger)

    This function configure the initial counter value. The counter will get this value after an overflow event.

- status_t FTM_DRV_SetHalfCycleReloadPoint (uint32_t instance, uint16_t reloadPoint, bool softwareTrigger)

    This function configure the value of the counter which will generates an reload point.

- status_t FTM_DRV_SetSoftOutChnValue (uint32_t instance, uint8_t channelsValues, bool softwareTrigger)

    *This function will force the output value of a channel to a specific value. Before using this function it's mandatory to mask the match events using FTM_DRV_MaskOutputChannels and to enable software output control using FTM_↩
DRV_SetSoftwareOutputChannelControl.*

- status_t FTM_DRV_SetSoftwareOutputChannelControl (uint32_t instance, uint8_t channelsMask, bool softwareTrigger)

    This function will configure which output channel can be software controlled.

- status_t FTM_DRV_SetAllChnSoftwareOutputControl (uint32_t instance, uint8_t channelMask, uint8_↩
t channelValueMask)

    This function will control list of channels by software to force the output to specified value.

- status_t FTM_DRV_SetInvertingControl (uint32_t instance, uint8_t channelsPairMask, bool softwareTrigger)

    This function will configure if the second channel of a pair will be inverted or not.

- status_t FTM_DRV_SetModuloCounterValue (uint32_t instance, uint16_t counterValue, bool softwareTrigger)

    This function configure the maximum counter value.

- status_t FTM_DRV_SetOutputlevel (uint32_t instance, uint8_t channel, uint8_t level)

    This function will set the channel edge or level on the selection of the channel mode.

- status_t FTM_DRV_SetSync (uint32_t instance, const ftm_pwm_sync_t ∗param)

    This function configures sync mechanism for some FTM registers (MOD, CNINT, HCR, CnV, OUTMASK, INVCTRL, SWOCTRL).

- status_t FTM_DRV_EnableInterrupts (uint32_t instance, uint32_t interruptMask)

This function will enable the generation a list of interrupts. It includes the FTM overflow interrupts, the reload point interrupt, the fault interrupt and the channel (n) interrupt.

- void FTM_DRV_DisableInterrupts (uint32_t instance, uint32_t interruptMask)

    This function is used to disable some interrupts.

- uint32_t FTM_DRV_GetEnabledInterrupts (uint32_t instance)

    This function will get the enabled FTM interrupts.

- uint32_t FTM_DRV_GetStatusFlags (uint32_t instance)

    This function will get the FTM status flags.

- void FTM_DRV_ClearStatusFlags (uint32_t instance, uint32_t flagMask)

    This function is used to clear the FTM status flags.

- uint32_t FTM_DRV_GetFrequency (uint32_t instance)

    Retrieves the frequency of the clock source feeding the FTM counter.

- uint16_t FTM_DRV_ConvertFreqToPeriodTicks (uint32_t instance, uint32_t freqencyHz)

    This function is used to covert the given frequency to period in ticks.

Variables

- FTM_Type *const g_ftmBase [FTM_INSTANCE_COUNT]

    Table of base addresses for FTM instances.

- const IRQn_Type g_ftmIrqId [FTM_INSTANCE_COUNT][FEATURE_FTM_CHANNEL_COUNT]

    Interrupt vectors for the FTM peripheral.

- const IRQn_Type g_ftmFaultIrqId [FTM_INSTANCE_COUNT]
- const IRQn_Type g_ftmOverflowIrqId [FTM_INSTANCE_COUNT]
- const IRQn_Type g_ftmReloadIrqId [FTM_INSTANCE_COUNT]
- ftm_state_t * ftmStatePtr [FTM_INSTANCE_COUNT]

    Pointer to runtime state structure.

16.38.2 Data Structure Documentation

16.38.2.1 struct ftm_state_t

FlexTimer state structure of the driver.

Implements : ftm_state_t_Class

Definition at line 402 of file ftm_common.h.

Data Fields

- ftm_clock_source_t ftmClockSource
- ftm_config_mode_t ftmMode
- uint16_t ftmPeriod
- uint32_t ftmSourceClockFrequency
- uint16_t measurementResults [FEATURE_FTM_CHANNEL_COUNT]
- void * channelsCallbacksParams [FEATURE_FTM_CHANNEL_COUNT]
- ic_callback_t channelsCallbacks [FEATURE_FTM_CHANNEL_COUNT]
- bool enableNotification [FEATURE_FTM_CHANNEL_COUNT]

Field Documentation

16.38.2.1.1 ic_callback_t channelsCallbacks[FEATURE_FTM_CHANNEL_COUNT]

The callback function for channels events

Definition at line 410 of file ftm_common.h.

16.38.2.1.2   void∗ channelsCallbacksParams[FEATURE_FTM_CHANNEL_COUNT]

The parameters of callback function for channels events

Definition at line 409 of file ftm_common.h.

16.38.2.1.3   bool enableNotification[FEATURE_FTM_CHANNEL_COUNT]

To save channels enable the notification on the callback application

Definition at line 411 of file ftm_common.h.

16.38.2.1.4   ftm_clock_source_t ftmClockSource

Clock source used by FTM counter

Definition at line 404 of file ftm_common.h.

16.38.2.1.5   ftm_config_mode_t ftmMode

Mode of operation for FTM

Definition at line 405 of file ftm_common.h.

16.38.2.1.6   uint16_t ftmPeriod

This field is used only in PWM mode to store signal period

Definition at line 406 of file ftm_common.h.

16.38.2.1.7   uint32_t ftmSourceClockFrequency

The clock frequency is used for counting

Definition at line 407 of file ftm_common.h.

16.38.2.1.8   uint16_t measurementResults[FEATURE_FTM_CHANNEL_COUNT]

This field is used only in input capture mode to store edges time stamps

Definition at line 408 of file ftm_common.h.

16.38.2.2   struct ftm_pwm_sync_t

FlexTimer Registers sync parameters Please don't use software and hardware trigger simultaneously Implements : ftm_pwm_sync_t_Class.

Definition at line 419 of file ftm_common.h.

Data Fields

- bool softwareSync
- bool hardwareSync0
- bool hardwareSync1
- bool hardwareSync2
- bool maxLoadingPoint
- bool minLoadingPoint
- ftm_reg_update_t inverterSync
- ftm_reg_update_t outRegSync
- ftm_reg_update_t maskRegSync
- ftm_reg_update_t initCounterSync
- bool autoClearTrigger
- ftm_pwm_sync_mode_t syncPoint

Field Documentation

16.38.2.2.1   bool autoClearTrigger

Available only for hardware trigger

Definition at line 437 of file ftm_common.h.

16.38.2.2.2   bool hardwareSync0

True - enable hardware 0 sync, False - disable hardware 0 sync

Definition at line 423 of file ftm_common.h.

16.38.2.2.3   bool hardwareSync1

True - enable hardware 1 sync, False - disable hardware 1 sync

Definition at line 425 of file ftm_common.h.

16.38.2.2.4   bool hardwareSync2

True - enable hardware 2 sync, False - disable hardware 2 sync

Definition at line 427 of file ftm_common.h.

16.38.2.2.5   ftm_reg_update_t initCounterSync

Configures CNTIN sync

Definition at line 436 of file ftm_common.h.

16.38.2.2.6   ftm_reg_update_t inverterSync

Configures INVCTRL sync

Definition at line 433 of file ftm_common.h.

16.38.2.2.7   ftm_reg_update_t maskRegSync

Configures OUTMASK sync

Definition at line 435 of file ftm_common.h.

16.38.2.2.8   bool maxLoadingPoint

True - enable maximum loading point, False - disable maximum loading point

Definition at line 429 of file ftm_common.h.

16.38.2.2.9   bool minLoadingPoint

True - enable minimum loading point, False - disable minimum loading point

Definition at line 431 of file ftm_common.h.

16.38.2.2.10   ftm_reg_update_t outRegSync

Configures SWOCTRL sync

Definition at line 434 of file ftm_common.h.

16.38.2.2.11   bool softwareSync

True - enable software sync, False - disable software sync

Definition at line 421 of file ftm_common.h.

16.38.2.2.12   ftm_pwm_sync_mode_t syncPoint

Configure synchronization method (waiting next loading point or immediate)

Definition at line 438 of file ftm_common.h.

16.38.2.3   struct ftm_user_config_t

Configuration structure that the user needs to set.

Implements : ftm_user_config_t_Class

Definition at line 447 of file ftm_common.h.

Data Fields

- ftm_pwm_sync_t syncMethod
- ftm_config_mode_t ftmMode
- ftm_clock_ps_t ftmPrescaler
- ftm_clock_source_t ftmClockSource
- ftm_bdm_mode_t BDMMode
- bool isTofIsrEnabled
- bool enableInitializationTrigger

Field Documentation

16.38.2.3.1   ftm_bdm_mode_t BDMMode

Select FTM behavior in BDM mode

Definition at line 455 of file ftm_common.h.

16.38.2.3.2   bool enableInitializationTrigger

true: enable the generation of initialization trigger false: disable the generation of initialization trigger

Definition at line 458 of file ftm_common.h.

16.38.2.3.3   ftm_clock_source_t ftmClockSource

Select clock source for FTM

Definition at line 454 of file ftm_common.h.

16.38.2.3.4   ftm_config_mode_t ftmMode

Mode of operation for FTM

Definition at line 451 of file ftm_common.h.

16.38.2.3.5   ftm_clock_ps_t ftmPrescaler

Register pre-scaler options available in the ftm_clock_ps_t enumeration

Definition at line 452 of file ftm_common.h.

16.38.2.3.6   bool isTofIsrEnabled

true: enable interrupt, false: write interrupt is disabled

Definition at line 456 of file ftm_common.h.

16.38.2.3.7   ftm_pwm_sync_t syncMethod

Register sync options available in the ftm_sync_method_t enumeration

Definition at line 449 of file ftm_common.h.

16.38.3 Macro Definition Documentation

16.38.3.1 #define CHAN0_IDX (0U)

Channel number for CHAN1.

Definition at line 208 of file ftm_common.h.

16.38.3.2 #define CHAN1_IDX (1U)

Channel number for CHAN2.

Definition at line 210 of file ftm_common.h.

16.38.3.3 #define CHAN2_IDX (2U)

Channel number for CHAN3.

Definition at line 212 of file ftm_common.h.

16.38.3.4 #define CHAN3_IDX (3U)

Channel number for CHAN4.

Definition at line 214 of file ftm_common.h.

16.38.3.5 #define CHAN4_IDX (4U)

Channel number for CHAN5.

Definition at line 216 of file ftm_common.h.

16.38.3.6 #define CHAN5_IDX (5U)

Channel number for CHAN6.

Definition at line 218 of file ftm_common.h.

16.38.3.7 #define CHAN6_IDX (6U)

Channel number for CHAN7.

Definition at line 220 of file ftm_common.h.

16.38.3.8 #define CHAN7_IDX (7U)

Definition at line 222 of file ftm_common.h.

16.38.3.9 #define FTM_RMW_CnSCV_REG(*base, channel, mask, value*) (((base)->CONTROLS[channel].CnSC) = (((base)->CONTROLS[channel].CnSC) & ∼(mask)) | (value)))

FTM_CnSCV - Read and modify and write Channel (n) Status And Control (RW)

Definition at line 115 of file ftm_common.h.

16.38.3.10 #define FTM_RMW_CNT(*base, mask, value*) (((base)->CNT) = (((base)->CNT) & ∼(mask)) | (value)))

FTM_CNT - Read and modify and write to Counter (RW)

Definition at line 90 of file ftm_common.h.

16.38.3.11   #define FTM_RMW_CNTIN(*base,   mask,   value* **) (((base)-$>$CNTIN) = ((((base)-$>$CNTIN) & $\sim$(mask)) $|$ (value)))**

FTM_CNTIN - Read and modify and write Counter Initial Value (RW)

Definition at line 100 of file ftm_common.h.

16.38.3.12   #define FTM_RMW_CONF(*base,   mask,   value* **) (((base)-$>$CONF) = ((((base)-$>$CONF) & $\sim$(mask)) $|$ (value)))**

FTM_CONF - Read and modify and write Configuration (RW)

Definition at line 139 of file ftm_common.h.

16.38.3.13   #define FTM_RMW_DEADTIME(*base,   mask,   value* **) (((base)-$>$DEADTIME) = ((((base)-$>$DEADTIME) & $\sim$(mask)) $|$ (value)))**

FTM_DEADTIME - Read and modify and write Dead-time Insertion Control (RW)

Definition at line 120 of file ftm_common.h.

16.38.3.14   #define FTM_RMW_EXTTRIG_REG(*base,   mask,   value* **) (((base)-$>$EXTTRIG) = ((((base)-$>$EXTTRIG) & $\sim$(mask)) $|$ (value)))**

FTM_EXTTRIG - Read and modify and write External Trigger Control (RW)

Definition at line 124 of file ftm_common.h.

16.38.3.15   #define FTM_RMW_FILTER(*base,   mask,   value* **) (((base)-$>$FILTER) = ((((base)-$>$FILTER) & $\sim$(mask)) $|$ (value)))**

FILTER - Read and modify and write Filter (RW)

Definition at line 149 of file ftm_common.h.

16.38.3.16   #define FTM_RMW_FLTCTRL(*base,   mask,   value* **) (((base)-$>$FLTCTRL) = ((((base)-$>$FLTCTRL) & $\sim$(mask)) $|$ (value)))**

FTM_FLTCTRL - Read and modify and write Fault Control (RW)

Definition at line 129 of file ftm_common.h.

16.38.3.17   #define FTM_RMW_FMS(*base,   mask,   value* **) (((base)-$>$FMS) = ((((base)-$>$FMS) & $\sim$(mask)) $|$ (value)))**

FTM_FMS - Read and modify and write Fault Mode Status (RW)

Definition at line 134 of file ftm_common.h.

16.38.3.18   #define FTM_RMW_MOD(*base,   mask,   value* **) (((base)-$>$MOD) = ((((base)-$>$MOD) & $\sim$(mask)) $|$ (value)))**

FTM_MOD - Read and modify and write Modulo (RW)

Definition at line 95 of file ftm_common.h.

16.38.3.19   #define FTM_RMW_MODE(*base,   mask,   value* **) (((base)-$>$MODE) = ((((base)-$>$MODE) & $\sim$(mask)) $|$ (value)))**

FTM_MODE - Read and modify and write Counter Features Mode Selection (RW)

Definition at line 110 of file ftm_common.h.

16.38.3.20   #define FTM_RMW_PAIR0DEADTIME(*base,   mask,   value* **) (((base)-$>$PAIR0DEADTIME) = ((((base)-$>$PAIR0DEADTIME) & $\sim$(mask)) $|$ (value)))**

FTM_PAIR0DEADTIME - Read and modify and write Dead-time Insertion Control for the pair 0 (RW)

Definition at line 164 of file ftm_common.h.

16.38.3.21 **#define FTM_RMW_PAIR1DEADTIME(** *base,* *mask,* *value* **) (((base)->PAIR1DEADTIME) = ((((base)->PAIR1DEADTIME) & ∼(mask)) | (value)))**

FTM_PAIR1DEADTIME - Read and modify and write Dead-time Insertion Control for the pair 1 (RW)

Definition at line 169 of file ftm_common.h.

16.38.3.22 **#define FTM_RMW_PAIR2DEADTIME(** *base,* *mask,* *value* **) (((base)->PAIR2DEADTIME) = ((((base)->PAIR2DEADTIME) & ∼(mask)) | (value)))**

FTM_PAIR2DEADTIME - Read and modify and write Dead-time Insertion Control for the pair 2 (RW)

Definition at line 174 of file ftm_common.h.

16.38.3.23 **#define FTM_RMW_PAIR3DEADTIME(** *base,* *mask,* *value* **) (((base)->PAIR3DEADTIME) = ((((base)->PAIR3DEADTIME) & ∼(mask)) | (value)))**

FTM_PAIR3DEADTIME - Read and modify and write Dead-time Insertion Control for the pair 3 (RW)

Channel number for CHAN0.

Definition at line 179 of file ftm_common.h.

16.38.3.24 **#define FTM_RMW_POL(** *base, mask, value* **) (((base)->POL) = ((((base)->POL) & ∼(mask)) | (value)))**

POL - Read and modify and write Polarity (RW)

Definition at line 144 of file ftm_common.h.

16.38.3.25 **#define FTM_RMW_QDCTRL(** *base, mask, value* **) (((base)->QDCTRL) = ((((base)->QDCTRL) & ∼(mask)) | (value)))**

QDCTRL - Read and modify and write Quadrature Decoder Control And Status (RW)

Definition at line 159 of file ftm_common.h.

16.38.3.26 **#define FTM_RMW_SC(** *base, mask, value* **) (((base)->SC) = ((((base)->SC) & ∼(mask)) | (value)))**

FTM_SC - Read and modify and write to Status And Control (RW)

Definition at line 85 of file ftm_common.h.

16.38.3.27 **#define FTM_RMW_STATUS(** *base, mask, value* **) (((base)->STATUS) = ((((base)->STATUS) & ∼(mask)) | (value)))**

FTM_STATUS - Read and modify and write Capture And Compare Status (RW)

Definition at line 105 of file ftm_common.h.

16.38.3.28 **#define FTM_RMW_SYNC(** *base, mask, value* **) (((base)->SYNC) = ((((base)->SYNC) & ∼(mask)) | (value)))**

SYNC - Read and modify and write Synchronization (RW)

Definition at line 154 of file ftm_common.h.

16.38.4 **Enumeration Type Documentation**

16.38.4.1 **enum ftm_bdm_mode_t**

Options for the FlexTimer behavior in BDM Mode.

Implements : ftm_bdm_mode_t_Class

Enumerator

  FTM_BDM_MODE_00 FTM counter stopped, CH(n)F bit can be set, FTM channels in functional mode, writes

to MOD,CNTIN and C(n)V registers bypass the register buffers

FTM_BDM_MODE_01 FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are forced to their safe value , writes to MOD,CNTIN and C(n)V registers bypass the register buffers

FTM_BDM_MODE_10 FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are frozen when chip enters in BDM mode, writes to MOD, CNTIN and C(n)V registers bypass the register buffers

FTM_BDM_MODE_11 FTM counter in functional mode, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers is in fully functional mode

Definition at line 381 of file ftm_common.h.

16.38.4.2   enum ftm_clock_ps_t

FlexTimer pre-scaler factor selection for the clock source. In quadrature decoder mode set FTM_CLOCK_DIVID↩
_BY_1.

Implements : ftm_clock_ps_t_Class

Enumerator

>   **FTM_CLOCK_DIVID_BY_1** Divide by 1
>
>   **FTM_CLOCK_DIVID_BY_2** Divide by 2
>
>   **FTM_CLOCK_DIVID_BY_4** Divide by 4
>
>   **FTM_CLOCK_DIVID_BY_8** Divide by 8
>
>   **FTM_CLOCK_DIVID_BY_16** Divide by 16
>
>   **FTM_CLOCK_DIVID_BY_32** Divide by 32
>
>   **FTM_CLOCK_DIVID_BY_64** Divide by 64
>
>   **FTM_CLOCK_DIVID_BY_128** Divide by 128

Definition at line 263 of file ftm_common.h.

16.38.4.3   enum ftm_clock_source_t

FlexTimer clock source selection.

Implements : ftm_clock_source_t_Class

Enumerator

>   **FTM_CLOCK_SOURCE_NONE** None use clock for FTM
>
>   **FTM_CLOCK_SOURCE_SYSTEMCLK** System clock
>
>   **FTM_CLOCK_SOURCE_FIXEDCLK** Fixed clock
>
>   **FTM_CLOCK_SOURCE_EXTERNALCLK** External clock

Definition at line 249 of file ftm_common.h.

16.38.4.4   enum ftm_config_mode_t

FlexTimer operation mode.

Implements : ftm_config_mode_t_Class

Enumerator

>   **FTM_MODE_NOT_INITIALIZED** The driver is not initialized
>
>   **FTM_MODE_INPUT_CAPTURE** Input capture
>
>   **FTM_MODE_OUTPUT_COMPARE** Output compare
>
>   **FTM_MODE_EDGE_ALIGNED_PWM** Edge aligned PWM

FTM_MODE_CEN_ALIGNED_PWM Center aligned PWM

FTM_MODE_QUADRATURE_DECODER Quadrature decoder

FTM_MODE_UP_TIMER Timer with up counter

FTM_MODE_UP_DOWN_TIMER timer with up-down counter

Definition at line 232 of file ftm_common.h.

16.38.4.5 enum ftm_deadtime_ps_t

FlexTimer pre-scaler factor for the dead-time insertion.

Implements : ftm_deadtime_ps_t_Class

Enumerator

FTM_DEADTIME_DIVID_BY_1 Divide by 1

FTM_DEADTIME_DIVID_BY_4 Divide by 4

FTM_DEADTIME_DIVID_BY_16 Divide by 16

Definition at line 345 of file ftm_common.h.

16.38.4.6 enum ftm_interrupt_option_t

List of FTM interrupts.

Implements : ftm_interrupt_option_t_Class

Enumerator

FTM_CHANNEL0_INT_ENABLE Channel 0 interrupt

FTM_CHANNEL1_INT_ENABLE Channel 1 interrupt

FTM_CHANNEL2_INT_ENABLE Channel 2 interrupt

FTM_CHANNEL3_INT_ENABLE Channel 3 interrupt

FTM_CHANNEL4_INT_ENABLE Channel 4 interrupt

FTM_CHANNEL5_INT_ENABLE Channel 5 interrupt

FTM_CHANNEL6_INT_ENABLE Channel 6 interrupt

FTM_CHANNEL7_INT_ENABLE Channel 7 interrupt

FTM_FAULT_INT_ENABLE Fault interrupt

FTM_TIME_OVER_FLOW_INT_ENABLE Time overflow interrupt

FTM_RELOAD_INT_ENABLE Reload interrupt; Available only on certain SoC's

Definition at line 280 of file ftm_common.h.

16.38.4.7 enum ftm_pwm_sync_mode_t

FTM update register.

Implements : ftm_pwm_sync_mode_t_Class

Enumerator

FTM_WAIT_LOADING_POINTS FTM register is updated at first loading point

FTM_UPDATE_NOW FTM register is updated immediately

Definition at line 334 of file ftm_common.h.

16.38.4.8 enum **ftm_quad_decode_mode_t**

FlexTimer quadrature decode modes, phase encode or count and direction mode.

Implements : ftm_quad_decode_mode_t_Class

Enumerator

> **FTM_QUAD_PHASE_ENCODE** Phase encoding mode
>
> **FTM_QUAD_COUNT_AND_DIR** Counter and direction encoding mode

Definition at line 357 of file ftm_common.h.

16.38.4.9 enum **ftm_quad_phase_polarity_t**

FlexTimer quadrature phase polarities, normal or inverted polarity.

Implements : ftm_quad_phase_polarity_t_Class

Enumerator

> **FTM_QUAD_PHASE_NORMAL** Phase input signal is not inverted before identifying the rising and falling
> edges of this signal
>
> **FTM_QUAD_PHASE_INVERT** Phase input signal is inverted before identifying the rising and falling edges of
> this signal

Definition at line 368 of file ftm_common.h.

16.38.4.10 enum **ftm_reg_update_t**

FTM sync source.

Implements : ftm_reg_update_t_Class

Enumerator

> **FTM_SYSTEM_CLOCK** Register is updated with its buffer value at all rising edges of system clock
>
> **FTM_PWM_SYNC** Register is updated with its buffer value at the FTM synchronization

Definition at line 321 of file ftm_common.h.

16.38.4.11 enum **ftm_status_flag_t**

List of FTM flags.

Implements : ftm_status_flag_t_Class

Enumerator

> **FTM_CHANNEL0_FLAG** Channel 0 Flag
> **FTM_CHANNEL1_FLAG** Channel 1 Flag
> **FTM_CHANNEL2_FLAG** Channel 2 Flag
> **FTM_CHANNEL3_FLAG** Channel 3 Flag
> **FTM_CHANNEL4_FLAG** Channel 4 Flag
> **FTM_CHANNEL5_FLAG** Channel 5 Flag
> **FTM_CHANNEL6_FLAG** Channel 6 Flag
> **FTM_CHANNEL7_FLAG** Channel 7 Flag
> **FTM_FAULT_FLAG** Fault Flag
> **FTM_TIME_OVER_FLOW_FLAG** Time overflow Flag
> **FTM_RELOAD_FLAG** Reload Flag; Available only on certain SoC's
> **FTM_CHANNEL_TRIGGER_FLAG** Channel trigger Flag

Definition at line 300 of file ftm_common.h.

16.38.5 Function Documentation

16.38.5.1 static void FTM_DRV_ClearChnEventStatus (FTM_Type ∗const *ftmBase,* uint8_t *channel*) `[inline]`, `[static]`

Clears the FTM peripheral timer all channel event status.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number

Implements : FTM_DRV_ClearChnEventStatus_Activity

Definition at line 807 of file ftm_common.h.

16.38.5.2 static void FTM_DRV_ClearChSC (FTM_Type ∗const *ftmBase,* uint8_t *channel*) `[inline]`,`[static]`

Clears the content of Channel (n) Status And Control.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number

Implements : FTM_DRV_ClearChSC_Activity

Definition at line 554 of file ftm_common.h.

16.38.5.3 static void FTM_DRV_ClearFaultFlagDetected (FTM_Type ∗const *ftmBase,* uint8_t *channel*) `[inline]`, `[static]`

Clear a fault condition is detected at the fault input.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel

Implements : FTM_DRV_ClearFaultFlagDetected_Activity

Definition at line 993 of file ftm_common.h.

16.38.5.4 void FTM_DRV_ClearStatusFlags (uint32_t *instance,* uint32_t *flagMask*)

This function is used to clear the FTM status flags.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*flagMask*	The status flags to clear. This is a logical OR of members of the enumeration ftm_status_flag_t

Definition at line 681 of file ftm_common.c.

16.38.5.5 uint16_t FTM_DRV_ConvertFreqToPeriodTicks (uint32_t *instance,* uint32_t *freqencyHz*)

This function is used to covert the given frequency to period in ticks.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*freqencyHz*	Frequency value in Hz.

Returns

> The value in ticks of the frequency

Definition at line 788 of file ftm_common.c.

16.38.5.6   status_t FTM_DRV_Deinit (uint32_t *instance*)

Shuts down the FTM driver.

Parameters

in	*instance*	The FTM peripheral instance number.

Returns

> operation status
>
> > • STATUS_SUCCESS : Completed successfully.
> >
> > • STATUS_ERROR : Error occurred.

Definition at line 196 of file ftm_common.c.

16.38.5.7   static void FTM_DRV_DisableFaultInt (FTM_Type ∗const *ftmBase*) `[inline],[static]`

Disables the FTM peripheral timer fault interrupt.

Parameters

in	*ftmBase*	The FTM base address pointer

Implements : FTM_DRV_DisableFaultInt_Activity

Definition at line 880 of file ftm_common.h.

16.38.5.8   void FTM_DRV_DisableInterrupts (uint32_t *instance,* uint32_t *interruptMask*)

This function is used to disable some interrupts.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*interruptMask*	The mask of interrupt. This is a logical OR of members of the enumeration ftm_interrupt_option_t

Definition at line 536 of file ftm_common.c.

16.38.5.9   status_t FTM_DRV_EnableInterrupts (uint32_t *instance,* uint32_t *interruptMask*)

This function will enable the generation a list of interrupts. It includes the FTM overflow interrupts, the reload point interrupt, the fault interrupt and the channel (n) interrupt.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*interruptMask*	The mask of interrupt. This is a logical OR of members of the enumeration ftm_interrupt_option_t

Returns

> operation status
>
> > • STATUS_SUCCESS : Completed successfully.

Definition at line 485 of file ftm_common.c.

16.38.5.10 **static bool FTM_DRV_GetChInputState (const FTM_Type ∗ *ftmBase,* uint8_t *channel*)** `[inline]`, `[static]`

Get the state of channel input.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number

Returns

> State of the channel inputs
>
> - true : The channel input is one
>
> - false: The channel input is zero

Implements : FTM_DRV_GetChInputState_Activity

Definition at line 720 of file ftm_common.h.

16.38.5.11   static uint16_t FTM_DRV_GetChnCountVal (const FTM_Type ∗ *ftmBase,* uint8_t *channel*) [inline], [static]

Gets the FTM peripheral timer channel counter value.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number

Returns

> Channel counter value

Implements : FTM_DRV_GetChnCountVal_Activity

Definition at line 757 of file ftm_common.h.

16.38.5.12   static uint8_t FTM_DRV_GetChnEdgeLevel (const FTM_Type ∗ *ftmBase,* uint8_t *channel*) [inline], [static]

Gets the FTM peripheral timer channel edge level.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number

Returns

> The ELSnB:ELSnA mode value, will be 00, 01, 10, 11

Implements : FTM_DRV_GetChnEdgeLevel_Activity

Definition at line 576 of file ftm_common.h.

16.38.5.13   static bool FTM_DRV_GetChnEventStatus (const FTM_Type ∗ *ftmBase,* uint8_t *channel*) [inline], [static]

Gets the FTM peripheral timer channel event status.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number

Returns

> Channel event status
>
> - true : A channel event has occurred
> - false : No channel event has occurred

Implements : FTM_DRV_GetChnEventStatus_Activity

Definition at line 777 of file ftm_common.h.

16.38.5.14   static bool FTM_DRV_GetChOutputValue (const FTM_Type ∗ *ftmBase,* uint8_t *channel*) `[inline]`, `[static]`

Get the value of channel output.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number

Returns

> Value of the channel outputs
>
> - true : The channel output is one
> - false: The channel output is zero

Implements : FTM_DRV_GetChOutputValue_Activity

Definition at line 739 of file ftm_common.h.

16.38.5.15   static uint8_t FTM_DRV_GetClockFilterPs (const FTM_Type ∗ *ftmBase*) `[inline]`,`[static]`

Reads the FTM filter clock divider.

Parameters

in	*ftmBase*	The FTM base address pointer

Returns

> The FTM filter clock pre-scale divider

Implements : FTM_DRV_GetClockFilterPs_Activity

Definition at line 499 of file ftm_common.h.

16.38.5.16   static uint16_t FTM_DRV_GetCounter (const FTM_Type ∗ *ftmBase*) `[inline]`,`[static]`

Returns the FTM peripheral current counter value.

Parameters

in	*ftmBase*	The FTM base address pointer

Returns

> The current FTM timer counter value

Implements : FTM_DRV_GetCounter_Activity

Definition at line 513 of file ftm_common.h.

16.38.5.17   static uint16_t FTM_DRV_GetCounterInitVal (const FTM_Type ∗ *ftmBase*)   `[inline],[static]`

Returns the FTM peripheral counter initial value.

Parameters

in	*ftmBase*	The FTM base address pointer

Returns

   FTM timer counter initial value

Implements : FTM_DRV_GetCounterInitVal_Activity

Definition at line 541 of file ftm_common.h.

16.38.5.18   uint32_t FTM_DRV_GetEnabledInterrupts (uint32_t *instance*)

This function will get the enabled FTM interrupts.

Parameters

in	*instance*	The FTM peripheral instance number.

Returns

   The enabled interrupts. This is the logical OR of members of the enumeration ftm_interrupt_option_t

Definition at line 585 of file ftm_common.c.

16.38.5.19   static uint32_t FTM_DRV_GetEventStatus (const FTM_Type ∗ *ftmBase*)   [inline],[static]

Gets the FTM peripheral timer status info for all channels.

Parameters

in	*ftmBase*	The FTM base address pointer

Returns

   Channel event status value

Implements : FTM_DRV_GetEventStatus_Activity

Definition at line 794 of file ftm_common.h.

16.38.5.20   uint32_t FTM_DRV_GetFrequency (uint32_t *instance*)

Retrieves the frequency of the clock source feeding the FTM counter.

Function will return a 0 if no clock source is selected and the FTM counter is disabled

Parameters

in	*instance*	The FTM peripheral instance number.

Returns

   The frequency of the clock source running the FTM counter (0 if counter is disabled)

Definition at line 733 of file ftm_common.c.

16.38.5.21   static uint16_t FTM_DRV_GetMod (const FTM_Type ∗ *ftmBase*)   [inline],[static]

Returns the FTM peripheral counter modulo value.

Parameters

in	*ftmBase*	The FTM base address pointer

Returns

FTM timer modulo value

Implements : FTM_DRV_GetMod_Activity

Definition at line 527 of file ftm_common.h.

16.38.5.22 uint32_t FTM_DRV_GetStatusFlags (uint32_t *instance*)

This function will get the FTM status flags.

Parameters

in	*instance*	The FTM peripheral instance number.

Returns

The status flags. This is the logical OR of members of the enumeration ftm_status_flag_t

Definition at line 631 of file ftm_common.c.

16.38.5.23 static bool FTM_DRV_GetTriggerControled (const FTM_Type ∗ *ftmBase,* uint8_t *channel*) `[inline]`, `[static]`

Returns whether the trigger mode is enabled.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number

Returns

State of the channel outputs

- true : Enabled a trigger generation on channel output
- false: PWM outputs without generating a pulse

Implements : FTM_DRV_GetTriggerControled_Activity

Definition at line 701 of file ftm_common.h.

16.38.5.24 status_t FTM_DRV_Init (uint32_t *instance,* const ftm_user_config_t ∗ *info,* ftm_state_t ∗ *state*)

Initializes the FTM driver.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*info*	The FTM user configuration structure, see ftm_user_config_t.
out	*state*	The FTM state structure of the driver.

Returns

operation status

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 117 of file ftm_common.c.

16.38.5.25   static bool FTM_DRV_IsChnDma (const FTM_Type ∗ *ftmBase,* uint8_t *channel*)   `[inline],[static]`

Returns whether the FTM peripheral timer channel DMA is enabled.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number

Returns

> State of the FTM peripheral timer channel DMA
>
> - true : Enabled DMA transfers
> - false: Disabled DMA transfers

Implements : FTM_DRV_IsChnDma_Activity

Definition at line 661 of file ftm_common.h.

16.38.5.26   static bool FTM_DRV_IsChnIcrst (const FTM_Type ∗ *ftmBase,* **uint8_t** *channel* **)**   `[inline],[static]`

Returns whether the FTM FTM counter is reset.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number

Returns

> State of the FTM peripheral timer channel ICRST
>
> - true : Enabled the FTM counter reset
> - false: Disabled the FTM counter reset

Implements : FTM_DRV_IsChnIcrst_Activity

Definition at line 621 of file ftm_common.h.

16.38.5.27   static bool FTM_DRV_IsFaultFlagDetected (const FTM_Type ∗ *ftmBase,* **uint8_t** *channel* **)**   `[inline],` `[static]`

Checks whether a fault condition is detected at the fault input.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel

Returns

> the fault condition status
>
> - true : A fault condition was detected at the fault input
> - false: No fault condition was detected at the fault input

Implements : FTM_DRV_IsFaultFlagDetected_Activity

Definition at line 977 of file ftm_common.h.

16.38.5.28   static bool FTM_DRV_IsFaultInputEnabled (const FTM_Type ∗ *ftmBase* **)**   `[inline],[static]`

Checks whether the logic OR of the fault inputs is enabled.

Parameters

in	*ftmBase*	The FTM base address pointer

Returns

the enabled fault inputs status

- true : The logic OR of the enabled fault inputs is 1
- false: The logic OR of the enabled fault inputs is 0

Implements : FTM_DRV_IsFaultInputEnabled_Activity

Definition at line 960 of file ftm_common.h.

16.38.5.29 static bool FTM_DRV_IsFtmEnable (const FTM_Type ∗ *ftmBase*) `[inline],[static]`

Get status of the FTMEN bit in the FTM_MODE register.

Parameters

in	*ftmBase*	The FTM base address pointer

Returns

the FTM Enable status

- true : TPM compatibility. Free running counter and synchronization compatible with TPM
- false: Free running counter and synchronization are different from TPM behaviour

Implements : FTM_DRV_IsFtmEnable_Activity

Definition at line 911 of file ftm_common.h.

16.38.5.30 static bool FTM_DRV_IsWriteProtectionEnabled (const FTM_Type ∗ *ftmBase*) `[inline],[static]`

Checks whether the write protection is enabled.

Parameters

in	*ftmBase*	The FTM base address pointer

Returns

Write-protection status

- true : If enabled
- false: If not

Implements : FTM_DRV_IsWriteProtectionEnabled_Activity

Definition at line 944 of file ftm_common.h.

16.38.5.31 status_t FTM_DRV_MaskOutputChannels (uint32_t *instance,* uint32_t *channelsMask,* bool *softwareTrigger*)

This function will mask the output of the channels and at match events will be ignored by the masked channels.

Parameters

in	*instance*	The FTM peripheral instance number.

in	*channelsMask*	The mask which will select which channels will ignore match events.
in	*softwareTrigger*	If true a software trigger is generate to update PWM parameters.

Returns

> success
>
>> • STATUS_SUCCESS : Completed successfully.
>>
>> • STATUS_ERROR : Error occurred.

Definition at line 216 of file ftm_common.c.

16.38.5.32   status_t FTM_DRV_SetAllChnSoftwareOutputControl (uint32_t *instance,* uint8_t *channelMask,* uint8_t *channelValueMask*)

This function will control list of channels by software to force the output to specified value.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*channelMask*	The mask which will configure the channels which can be software controlled.
in	*channelValue↩ Mask*	The values which will be software configured for channels.

Returns

> success
>
>> • STATUS_SUCCESS : Completed successfully.

Definition at line 321 of file ftm_common.c.

16.38.5.33   static void FTM_DRV_SetCaptureTestCmd (FTM_Type *const *ftmBase,* bool *enable*)   [inline],[static]

Enables or disables the FTM peripheral timer capture test mode.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*enable*	Capture Test Mode Enable • true : Capture test mode is enabled • false: Capture test mode is disabled

Implements : FTM_DRV_SetCaptureTestCmd_Activity

Definition at line 895 of file ftm_common.h.

16.38.5.34   static void FTM_DRV_SetChnDmaCmd (FTM_Type *const *ftmBase,* uint8_t *channel,* bool *enable*)   [inline], [static]

Enables or disables the FTM peripheral timer channel DMA.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number

in	*enable*	Enable DMA transfers for the channel
		• true : Enabled DMA transfers
		• false: Disabled DMA transfers

Implements : FTM_DRV_SetChnDmaCmd_Activity

Definition at line 640 of file ftm_common.h.

16.38.5.35   static void FTM_DRV_SetChnIcrstCmd (FTM_Type ∗const *ftmBase,* uint8_t *channel,* bool *enable*) `[inline]`, `[static]`

Configure the feature of FTM counter reset by the selected input capture event.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number
in	*enable*	Enable the FTM counter reset
		• true : FTM counter is reset
		• false: FTM counter is not reset

Implements : FTM_DRV_SetChnIcrstCmd_Activity

Definition at line 600 of file ftm_common.h.

16.38.5.36   static void FTM_DRV_SetChnOutputInitStateCmd (FTM_Type ∗const *ftmBase,* uint8_t *channel,* bool *state*) `[inline]`,`[static]`

Sets the FTM peripheral timer channel output initial state 0 or 1.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number
in	*state*	Initial state for channels output
		• true : The initialization value is 1
		• false: The initialization value is 0

Implements : FTM_DRV_SetChnOutputInitStateCmd_Activity

Definition at line 857 of file ftm_common.h.

16.38.5.37   static void FTM_DRV_SetChnOutputMask (FTM_Type ∗const *ftmBase,* uint8_t *channel,* bool *mask*) `[inline]`,`[static]`

Sets the FTM peripheral timer channel output mask.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number
in	*mask*	Value to set Output Mask
		• true : Channel output is masked
		• false: Channel output is not masked

Implements : FTM_DRV_SetChnOutputMask_Activity

Definition at line 830 of file ftm_common.h.

16.38.5.38   static void FTM_DRV_SetChnSoftwareCtrlCmd (FTM_Type ∗const *ftmBase,* uint8_t *channel,* bool *enable*) `[inline],[static]`

Enables or disables the channel software output control.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	Channel to be enabled or disabled
in	*enable*	State of channel software output control • true : To enable, channel output will be affected by software output control • false: To disable, channel output is unaffected

Implements : FTM_DRV_SetChnSoftwareCtrlCmd_Activity

Definition at line 1043 of file ftm_common.h.

16.38.5.39   static void FTM_DRV_SetChnSoftwareCtrlVal (FTM_Type ∗const *ftmBase,* uint8_t *channel,* bool *enable*) `[inline],[static]`

Sets the channel software output control value.

Parameters

in	*ftmBase*	The FTM base address pointer.
in	*channel*	Channel to be configured
in	*enable*	State of software output control value • true : to force 1 to the channel output • false: to force 0 to the channel output

Implements : FTM_DRV_SetChnSoftwareCtrlVal_Activity

Definition at line 1070 of file ftm_common.h.

16.38.5.40   static void FTM_DRV_SetClockFilterPs (FTM_Type ∗const *ftmBase,* uint8_t *filterPrescale*) `[inline],` `[static]`

Sets the filter Pre-scaler divider.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*filterPrescale*	The FTM peripheral clock pre-scale divider

Implements : FTM_DRV_SetClockFilterPs_Activity

Definition at line 484 of file ftm_common.h.

16.38.5.41   static void FTM_DRV_SetCountReinitSyncCmd (FTM_Type ∗const *ftmBase,* bool *enable*) `[inline],` `[static]`

Determines if the FTM counter is re-initialized when the selected trigger for synchronization is detected.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*enable*	FTM counter re-initialization selection • true : To update FTM counter when triggered • false: To count normally

Implements : FTM_DRV_SetCountReinitSyncCmd_Activity

Definition at line 927 of file ftm_common.h.

16.38.5.42   static void FTM_DRV_SetDualChnInvertCmd (FTM_Type ∗const *ftmBase,* uint8_t *chnlPairNum,* bool *enable*) `[inline],[static]`

Enables or disables the channel invert for a channel pair.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*chnlPairNum*	The FTM peripheral channel pair number
in	*enable*	State of channel invert for a channel pair • true : To enable channel inverting • false: To disable channel inversion

Implements : FTM_DRV_SetDualChnInvertCmd_Activity

Definition at line 1016 of file ftm_common.h.

16.38.5.43   static void FTM_DRV_SetExtPairDeadtimeValue (FTM_Type ∗const *ftmBase,* uint8_t *channelPair,* uint8_t *value*) `[inline],[static]`

Sets the FTM extended dead-time value for the channel pair.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channelPair*	The FTM peripheral channel pair (n)
in	*value*	The FTM peripheral extend pre-scale divider using the concatenation with the dead-time value

Implements : FTM_DRV_SetExtPairDeadtimeValue_Activity

Definition at line 1269 of file ftm_common.h.

16.38.5.44   static void FTM_DRV_SetGlobalLoadCmd (FTM_Type ∗const *ftmBase*) `[inline],[static]`

Set the global load mechanism.

Parameters

in	*ftmBase*	The FTM base address pointer • true : LDOK bit is set • false: No action

Implements : FTM_DRV_SetGlobalLoadCmd_Activity

Definition at line 1096 of file ftm_common.h.

16.38.5.45   static void FTM_DRV_SetGlobalTimeBaseCmd (FTM_Type ∗const *ftmBase,* bool *enable*) `[inline], [static]`

Enables or disables the FTM timer global time base.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*enable*	State of global time base
		• true : To enable
		• false: To disable

Implements : FTM_DRV_SetGlobalTimeBaseCmd_Activity

Definition at line 1240 of file ftm_common.h.

16.38.5.46   static void FTM_DRV_SetGlobalTimeBaseOutputCmd (FTM_Type ∗const *ftmBase,* bool *enable*) `[inline],` `[static]`

Enables or disables the FTM global time base signal generation to other FTM's.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*enable*	State of global time base signal
		• true : To enable
		• false: To disable

Implements : FTM_DRV_SetGlobalTimeBaseOutputCmd_Activity

Definition at line 1224 of file ftm_common.h.

16.38.5.47   static void FTM_DRV_SetHalfCycleCmd (FTM_Type ∗const *ftmBase,* bool *enable*) `[inline],[static]`

Enable the half cycle reload.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*enable*	State of the half cycle match as a reload opportunity
		• true : Half cycle reload is enabled
		• false: Half cycle reload is disabled

Implements : FTM_DRV_SetHalfCycleCmd_Activity

Definition at line 1134 of file ftm_common.h.

16.38.5.48   status_t FTM_DRV_SetHalfCycleReloadPoint (uint32_t *instance,* uint16_t *reloadPoint,* bool *softwareTrigger*)

This function configure the value of the counter which will generates an reload point.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*reloadPoint*	Counter value which generates the reload point.
in	*softwareTrigger*	If true a software trigger is generate to update parameters.

Returns

success

  • STATUS_SUCCESS : Completed successfully.
  • STATUS_ERROR : Error occurred.

Definition at line 258 of file ftm_common.c.

16.38.5.49 status_t FTM_DRV_SetInitialCounterValue (uint32_t *instance,* uint16_t *counterValue,* bool *softwareTrigger*)

This function configure the initial counter value. The counter will get this value after an overflow event.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*counterValue*	Initial counter value.
in	*softwareTrigger*	If true a software trigger is generate to update parameters.

Returns

    success

        • STATUS_SUCCESS : Completed successfully.

        • STATUS_ERROR : Error occurred.

Definition at line 237 of file ftm_common.c.

16.38.5.50 static void FTM_DRV_SetInitTrigOnReloadCmd (FTM_Type ∗const *ftmBase,* bool *enable*) `[inline],` `[static]`

Enables or disables the FTM initialization trigger on Reload Point.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*enable*	bit controls whether an initialization trigger is generated • true : Trigger is generated when a reload point is reached • false: Trigger is generated on counter wrap events

Implements : FTM_DRV_SetInitTrigOnReloadCmd_Activity

Definition at line 1208 of file ftm_common.h.

16.38.5.51 status_t FTM_DRV_SetInvertingControl (uint32_t *instance,* uint8_t *channelsPairMask,* bool *softwareTrigger*)

This function will configure if the second channel of a pair will be inverted or not.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*channelsPair↩ Mask*	The mask which will configure which channel pair will invert the second channel.
in	*softwareTrigger*	If true a software trigger is generate to update registers.

Returns

    success

        • STATUS_SUCCESS : Completed successfully.

        • STATUS_ERROR : Error occurred.

Definition at line 344 of file ftm_common.c.

16.38.5.52 static void FTM_DRV_SetLoadCmd (FTM_Type ∗const *ftmBase,* bool *enable*) `[inline],[static]`

Enable the global load.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*enable*	State of the global load mechanism • true : Global Load OK enabled • false: Global Load OK disabled

Implements : FTM_DRV_SetLoadCmd_Activity

Definition at line 1111 of file ftm_common.h.

16.38.5.53   static void FTM_DRV_SetLoadFreq (FTM_Type ∗const *ftmBase,* uint8_t *val*) `[inline],[static]`

Sets the FTM timer TOF Frequency.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*val*	Value of the TOF bit set frequency

Implements : FTM_DRV_SetLoadFreq_Activity

Definition at line 1254 of file ftm_common.h.

16.38.5.54   status_t FTM_DRV_SetModuloCounterValue (uint32_t *instance,* uint16_t *counterValue,* bool *softwareTrigger*)

This function configure the maximum counter value.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*counterValue*	Maximum counter value
in	*softwareTrigger*	If true a software trigger is generate to update parameters.

Returns

>   success
>
>   • STATUS_SUCCESS : Completed successfully.
>
>   • STATUS_ERROR : Error occurred.

Definition at line 364 of file ftm_common.c.

16.38.5.55   status_t FTM_DRV_SetOutputlevel (uint32_t *instance,* uint8_t *channel,* uint8_t *level*)

This function will set the channel edge or level on the selection of the channel mode.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*channel*	The channel number.
in	*level*	The level or edge selection for channel mode.

Returns

>   success
>
>   • STATUS_SUCCESS : Completed successfully.

Definition at line 385 of file ftm_common.c.

16.38.5.56   static void FTM_DRV_SetPairDeadtimeCount (FTM_Type ∗const *ftmBase,* uint8_t *channelPair,* uint8_t *count*) `[inline],[static]`

Sets the FTM dead-time value for the channel pair.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channelPair*	The FTM peripheral channel pair (n)
in	*count*	The FTM peripheral selects the dead-time value • 0U : no counts inserted • 1U : 1 count is inserted • 2U : 2 count is inserted • ... up to a possible 63 counts

Implements : FTM_DRV_SetPairDeadtimeCount_Activity

Definition at line 1347 of file ftm_common.h.

16.38.5.57   static void FTM_DRV_SetPairDeadtimePrescale (FTM_Type ∗const *ftmBase,* uint8_t *channelPair,* ftm_deadtime_ps_t *divider*) `[inline],[static]`

Sets the FTM dead time divider for the channel pair.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channelPair*	The FTM peripheral channel pair (n)
in	*divider*	The FTM peripheral pre-scaler divider • FTM_DEADTIME_DIVID_BY_1 : Divide by 1 • FTM_DEADTIME_DIVID_BY_4 : Divide by 4 • FTM_DEADTIME_DIVID_BY_16: Divide by 16

Implements : FTM_DRV_SetPairDeadtimePrescale_Activity

Definition at line 1308 of file ftm_common.h.

16.38.5.58   static void FTM_DRV_SetPwmLoadChnSelCmd (FTM_Type ∗const *ftmBase,* uint8_t *channel,* bool *enable*) `[inline],[static]`

Includes or excludes the channel in the matching process.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	Channel to be configured
in	*enable*	State of channel • true : means include the channel in the matching process • false: means do not include channel in the matching process

Implements : FTM_DRV_SetPwmLoadChnSelCmd_Activity

Definition at line 1181 of file ftm_common.h.

16.38.5.59   static void FTM_DRV_SetPwmLoadCmd (FTM_Type ∗const *ftmBase,* bool *enable*) `[inline],[static]`

Enables or disables the loading of MOD, CNTIN and CV with values of their write buffer.

Parameters

in	*ftmBase*	The FTM base address pointer

in	*enable*	State of loading updated values
		
		    • true : To enable
		
		    • false: To disable

Implements : FTM_DRV_SetPwmLoadCmd_Activity

Definition at line 1157 of file ftm_common.h.

16.38.5.60   status_t FTM_DRV_SetSoftOutChnValue (uint32_t *instance,* uint8_t *channelsValues,* bool *softwareTrigger*)

This function will force the output value of a channel to a specific value. Before using this function it's mandatory to mask the match events using FTM_DRV_MaskOutputChannels and to enable software output control using FTM↩_DRV_SetSoftwareOutputChannelControl.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*channelsValues*	The values which will be software configured for channels.
in	*softwareTrigger*	If true a software trigger is generate to update registers.

Returns

success

    • STATUS_SUCCESS : Completed successfully.

    • STATUS_ERROR : Error occurred.

Definition at line 281 of file ftm_common.c.

16.38.5.61   status_t FTM_DRV_SetSoftwareOutputChannelControl (uint32_t *instance,* uint8_t *channelsMask,* bool *softwareTrigger*)

This function will configure which output channel can be software controlled.

Parameters

in	*instance*	The FTM peripheral instance number.
in	*channelsMask*	The mask which will configure the channels which can be software controlled.
in	*softwareTrigger*	If true a software trigger is generate to update registers.

Returns

success

    • STATUS_SUCCESS : Completed successfully.

    • STATUS_ERROR : Error occurred.

Definition at line 301 of file ftm_common.c.

16.38.5.62   status_t FTM_DRV_SetSync (uint32_t *instance,* const **ftm_pwm_sync_t ∗ *param*)**

This function configures sync mechanism for some FTM registers (MOD, CNINT, HCR, CnV, OUTMASK, INVCTRL, SWOCTRL).

Parameters

in	*instance*	The FTM peripheral instance number.
in	*param*	The sync configuration structure.

Returns

operation status

- STATUS_SUCCESS : Completed successfully.
- STATUS_ERROR : Error occurred.

Definition at line 409 of file ftm_common.c.

16.38.5.63 **static void FTM_DRV_SetTrigModeControlCmd (FTM_Type ∗const *ftmBase,* uint8_t *channel,* bool *enable*)**
[inline],[static]

Enables or disables the trigger generation on FTM channel outputs.

Parameters

in	*ftmBase*	The FTM base address pointer
in	*channel*	The FTM peripheral channel number
in	*enable*	Trigger mode control - false : Enable PWM output without generating a pulse - true : Disable a trigger generation on channel output

Implements : FTM_DRV_SetTrigModeControlCmd_Activity

Definition at line 680 of file ftm_common.h.

16.38.6   Variable Documentation

16.38.6.1   ftm_state_t∗ ftmStatePtr[FTM_INSTANCE_COUNT]

Pointer to runtime state structure.

Definition at line 84 of file ftm_common.c.

16.38.6.2   FTM_Type∗ const g_ftmBase[FTM_INSTANCE_COUNT]

Table of base addresses for FTM instances.

Definition at line 71 of file ftm_common.c.

16.38.6.3   const IRQn_Type g_ftmFaultIrqId[FTM_INSTANCE_COUNT]

Definition at line 75 of file ftm_common.c.

16.38.6.4   const IRQn_Type g_ftmIrqId[FTM_INSTANCE_COUNT][FEATURE_FTM_CHANNEL_COUNT]

Interrupt vectors for the FTM peripheral.

Definition at line 74 of file ftm_common.c.

16.38.6.5   const IRQn_Type g_ftmOverflowIrqId[FTM_INSTANCE_COUNT]

Definition at line 76 of file ftm_common.c.

16.38.6.6   const IRQn_Type g_ftmReloadIrqId[FTM_INSTANCE_COUNT]

Definition at line 77 of file ftm_common.c.

16.39 Initialization

16.39.1 Detailed Description

Initialize transport layer (queues, status, ...).

Functions

- void ld_init (l_ifc_handle iii)

    Initialize or reinitialize the raw and cooked layers.

16.39.2 Function Documentation

16.39.2.1 void ld_init (l_ifc_handle *iii*)

Initialize or reinitialize the raw and cooked layers.

Parameters

in		*iii*	Interface name

Returns

void

Initialize or reinitialize the raw and cooked layers on the interface iii. All the transport layer buffers will be initialized.

Definition at line 52 of file lin_commontl_api.c.

16.40 Interface management

16.40.1 Detailed Description

This group contains APIs that help users manage interface(s) in LIN node.

Functions

- l_bool l_ifc_init (l_ifc_handle iii)

  Initialize the controller specified by name, i.e. sets up internal functions such as the baud rate. The default schedule set by the l_ifc_init call will be the L_NULL_SCHEDULE where no frames will be sent and received. This is the first call a user must perform, before using any other interface related LIN API functions. The function returns zero if the initialization was successful and non-zero if failed.

- void l_ifc_goto_sleep (l_ifc_handle iii)

  Request slave nodes on the cluster connected to the interface to enter bus sleep mode by issuing one go to sleep command. This API is available only for Master nodes.

- void l_ifc_wake_up (l_ifc_handle iii)

  Transmit the wake up signal.

- l_u16 l_ifc_read_status (l_ifc_handle iii)

  This function will return the status of the previous communication.

16.40.2 Function Documentation

16.40.2.1 void l_ifc_goto_sleep (l_ifc_handle *iii*)

Request slave nodes on the cluster connected to the interface to enter bus sleep mode by issuing one go to sleep command. This API is available only for Master nodes.

Note

After sending go to sleep command successfully, the master node sets go to sleep flag to 1 and goes to sleep mode. At the end of Go to sleep schedule table, at the end of frame slot of go to sleep command, in l_sch_tick() the master node actually switches its active schedule table to Null to stop all communication. To start LIN communication, the master node shall call l_ifc_wake_up() to wake up LIN cluster and l_sch_set() to activate normal schedule table.

Parameters

in	*iii*	Interface name

Returns

void

Definition at line 391 of file lin_common_api.c.

16.40.2.2 l_bool l_ifc_init (l_ifc_handle *iii*)

Initialize the controller specified by name, i.e. sets up internal functions such as the baud rate. The default schedule set by the l_ifc_init call will be the L_NULL_SCHEDULE where no frames will be sent and received. This is the first call a user must perform, before using any other interface related LIN API functions. The function returns zero if the initialization was successful and non-zero if failed.

Parameters

in	*iii*	Interface name

Returns

Operation status

- Zero: Initialization was successful.
- Non-zero: Initialization failed.

Definition at line 420 of file lin_common_api.c.

16.40.2.3 l_u16 l_ifc_read_status (l_ifc_handle *iii*)

This function will return the status of the previous communication.

Parameters

in	*iii*	Interface name

Returns

l_u16

Definition at line 484 of file lin_common_api.c.

16.40.2.4 void l_ifc_wake_up (l_ifc_handle *iii*)

Transmit the wake up signal.

Parameters

in	*iii*	Interface name

Returns

void

Definition at line 470 of file lin_common_api.c.

16.41 Interrupt vector numbers for S32K116

This module covers interrupt number allocation.

16.42   Interrupt_manager

16.42.1   Detailed Description

Typedefs

- typedef void(∗ isr_t) (void)

    Interrupt handler type.

Functions

- void DefaultISR (void)

    Default ISR.

Interrupt manager APIs

- void INT_SYS_InstallHandler (IRQn_Type irqNumber, const isr_t newHandler, isr_t ∗const oldHandler)

    Installs an interrupt handler routine for a given IRQ number.
- void INT_SYS_EnableIRQ (IRQn_Type irqNumber)

    Enables an interrupt for a given IRQ number.
- void INT_SYS_DisableIRQ (IRQn_Type irqNumber)

    Disables an interrupt for a given IRQ number.
- void INT_SYS_EnableIRQGlobal (void)

    Enables system interrupt.
- void INT_SYS_DisableIRQGlobal (void)

    Disable system interrupt.
- void INT_SYS_SetPriority (IRQn_Type irqNumber, uint8_t priority)

    Set Interrupt Priority.
- uint8_t INT_SYS_GetPriority (IRQn_Type irqNumber)

    Get Interrupt Priority.

16.42.2   Typedef Documentation

16.42.2.1   typedef void(∗ isr_t) (void)

Interrupt handler type.

Definition at line 79 of file interrupt_manager.h.

16.42.3   Function Documentation

16.42.3.1   void DefaultISR (void)

Default ISR.

16.42.3.2   void INT_SYS_DisableIRQ (IRQn_Type *irqNumber*)

Disables an interrupt for a given IRQ number.

This function disables the individual interrupt for a specified IRQ number.

Parameters

irqNumber	IRQ number

Definition at line 186 of file interrupt_manager.c.

16.42.3.3   void INT_SYS_DisableIRQGlobal (void)

Disable system interrupt.

This function disables the global interrupt by calling the core API.

Definition at line 237 of file interrupt_manager.c.

16.42.3.4   void INT_SYS_EnableIRQ (IRQn_Type *irqNumber*)

Enables an interrupt for a given IRQ number.

This function enables the individual interrupt for a specified IRQ number.

Parameters

irqNumber	IRQ number

Definition at line 158 of file interrupt_manager.c.

16.42.3.5   void INT_SYS_EnableIRQGlobal (void)

Enables system interrupt.

This function enables the global interrupt by calling the core API.

Definition at line 214 of file interrupt_manager.c.

16.42.3.6   uint8_t INT_SYS_GetPriority (IRQn_Type *irqNumber*)

Get Interrupt Priority.

The function gets the priority of an interrupt.

Parameters

irqNumber	Interrupt number.

Returns

> priority Priority of the interrupt.

Definition at line 319 of file interrupt_manager.c.

16.42.3.7   void INT_SYS_InstallHandler (IRQn_Type *irqNumber,* const **isr_t *newHandler,* **isr_t** *∗const oldHandler*)**

Installs an interrupt handler routine for a given IRQ number.

This function lets the application register/replace the interrupt handler for a specified IRQ number. See a chip-specific reference manual for details and the startup_<SoC>.s file for each chip family to find out the default interrupt handler for each device.

Note

> This method is applicable only if interrupt vector is copied in RAM.

Parameters

irqNumber	IRQ number
newHandler	New interrupt handler routine address pointer
oldHandler	Pointer to a location to store current interrupt handler

Definition at line 98 of file interrupt_manager.c.

16.42.3.8 void INT_SYS_SetPriority (IRQn_Type *irqNumber,* uint8_t *priority*)

Set Interrupt Priority.

The function sets the priority of an interrupt.

Parameters

irqNumber	Interrupt number.
priority	Priority to set.

Definition at line 255 of file interrupt_manager.c.

16.43 J2602 Specific API

J2602 protocol is LIN 2.0 based. It contains LIN 2.0's modules to support Signal management, network management, scheduler and J2602 status management. The goal of J2602 is to improve the interoperability and interchangeability of LIN devices within a network by resolving those LIN2.0 requirements that are ambiguous, conflicting, or optional. Moreover, J2602 provides additional requirements that are not present in LIN2.0. For example: fault tolerant, operation, network topology, etc. Different to LIN2.1 protocol, J2602 does not support sporadic and event trigger frames in communication.

16.44   J2602 Transport Layer specific API

16.44.1   Detailed Description

Contains Transport Layer APIs that only used for J2602 protocol.

Modules

- [Node configuration](#)

    This group contains APIs that used for node configuration purpose.

16.44   J2602 Transport Layer specific API

16.45 LIN 2.1 Specific API

16.45.1 Detailed Description

LIN 2.1 is extended from in LIN 2.0 specification through diagnostic services and few functions were removed as obsolete.

1. LIN 2.1 is compatible with LIN 2.0:

   • A LIN 2.1 master node may handle a LIN 2.0 slave node if the master node also contains all functionality of a LIN 2.0 master node, e.g. obsolete functions like Assign frame Id.

   • A LIN 2.1 slave node can be used in a cluster with a LIN 2.0 master node if the LIN 2.1 slave node is pre-configured, i.e. the LIN 2.1 slave node has a valid configuration after reset.

2. Changes between LIN 2.0 and LIN 2.1:

   • LIN2.1 enhance the capacity of LIN2.0 on event-triggered frame collision handling and diagnostic services supported. Besides, several features are added to fulfill powerful capacity of LIN network such as configuration service, assign frame ID range configuration, etc.

Functions

   • void lin_collision_resolve (l_ifc_handle iii, l_u8 pid)

     Switch to collision resolve table.

   • void lin_update_word_status_lin21 (l_ifc_handle iii, lin_lld_event_id_t event_id)

     Update node status flags.

   • void lin_update_err_signal (l_ifc_handle iii, l_u8 frm_id)

     Update error signal.

   • void lin_make_res_evnt_frame (l_ifc_handle iii, l_u8 pid)

     This function packs signals associated with event trigger frame into buffer.

   • void lin_update_rx_evnt_frame (l_ifc_handle iii, l_u8 pid)

     The function updates the receive flags associated with signals/frames in case receive an event trigger frame.

16.45.2 Function Documentation

16.45.2.1 void lin_collision_resolve (l_ifc_handle *iii,* l_u8 *pid*)

Switch to collision resolve table.

Parameters

in	*iii*	Interface name
in	*pid*	PID to process

Returns

   void

Definition at line 35 of file lin_lin21_proto.c.

16.45.2.2 void lin_make_res_evnt_frame (l_ifc_handle *iii,* l_u8 *pid*)

This function packs signals associated with event trigger frame into buffer.

Parameters

in	*iii*	Interface name
in	*pid*	PID to process

Returns

void

Definition at line 223 of file lin_lin21_proto.c.

16.45.2.3 void lin_update_err_signal (l_ifc_handle *iii,* l_u8 *frm_id*)

Update error signal.

Parameters

in	*iii*	Interface name
in	*frm_id*	Frame index

Returns

void

Definition at line 150 of file lin_lin21_proto.c.

16.45.2.4 void lin_update_rx_evnt_frame (l_ifc_handle *iii,* l_u8 *pid*)

The function updates the receive flags associated with signals/frames in case receive an event trigger frame.

Parameters

in	*iii*	Interface name
in	*pid*	PID to process

Returns

void

Definition at line 186 of file lin_lin21_proto.c.

16.45.2.5 void lin_update_word_status_lin21 (l_ifc_handle *iii,* lin_lld_event_id_t *event_id*)

Update node status flags.

Parameters

in	*iii*	Interface name
in	*event_id*	Event id

Returns

void

Definition at line 70 of file lin_lin21_proto.c.

16.46   LIN Core API

16.46.1   Detailed Description

The LIN core API handles initialization, processing and a signal based interaction between the application and the LIN core. Refer to chapter 7, LIN 2.2A specification.

- Core API layer consists of API functions as defined by the LIN2.1/J2602 specifications.

- Enabling the user to utilize the LIN2.1/J2602 communication within the user application.

- Both the static and dynamic modes for calling the API functions are supported.

- The core API layer interacts with the low level layer and can be called by such upper layers as LIN2.1 TL API, LIN TL J2602 or application for diagnostic implementation.

Modules

- Common Core API.
- J2602 Specific API
- LIN 2.1 Specific API

16.47   LIN Driver

16.47.1   Detailed Description

This section describes the programming interface of the Peripheral driver for LIN.

16.47.2   LIN Driver Overview

The LIN (Local Interconnect Network) Driver is an use-case driven High Level Peripheral Driver. The driver is built on HAL drivers and provides users important key features. NXP provides LIN Stack as a middleware software package that is developed on LIN driver. Users also can create their own LIN applications and LIN stack that are compatible with LIN Specification.
In this release package, LIN Driver is built on LPUART interface.

16.47.3   LIN Driver Device structures

The driver uses instantiations of the lin_state_t to maintain the current state of a particular LIN Hardware instance module driver.
The user is required to provide memory for the driver state structures during the initialization. The driver itself does not statically allocate memory.

16.47.4   LIN Driver Initialization

1. To initialize the LIN driver, call the LIN_DRV_Init() function and pass the instance number of the relevant LIN hardware interface instance which is LPUART instance in this release.
   For example: to use LPUART0 pass value 0 to the initialization function.

2. Pass a user configuration structure lin_user_config_t as shown here:

   ```
   /* LIN Driver configuration structure */
   typedef struct {
       uint32_t baudRate;
       bool nodeFunction;
       bool autobaudEnable;
       lin_timer_get_time_interval_t timerGetTimeIntervalCallback;
   } lin_user_config_t;
   ```

3. For LIN, typically the user configures the lin_user_config_t instantiation with a baudrate from 1000bps to 20000bps.
   -E.g. 19200 bps linUserConfig.baudRate = 19200U.

4. Node function can be MASTER or SLAVE.
   -E.g. linUserConfig.nodeFunction = MASTER

5. If users do not want to use Autobaud feature, then just configure linUserConfig.autobaudEnable = FALSE.

6. Users shall assign measurement callback function pointer that is *timerGetTimeIntervalCallback*. This function must return time period between two consecutive calls in nano seconds with accuracy at least 0.1 microsecond and if this function is called for the first time, it will start the timer to measure time. When an event (such as detecting a falling edge of a dominant signal while node is in sleep mode) occurs, LIN driver will call *timerGetTimeIntervalCallback* to start time measurement. Then on rising edge of that signal, LIN driver will call *timerGetTimeIntervalCallback* function to get time interval of that dominant signal in nano seconds. If Autobaud feature is enabled, LIN driver uses *timerGetTimeIntervalCallback* to measure two bit time length between two consecutive falling edges of the sync byte in order to evaluate Master's baudrate. Users can implement this function in their applications. -E.g. linUserConfig.timerGetTimeIntervalCallback = timerGet↩
   TimeIntervalCallback0; This is a code example to set up a FTM0 for LIN Driver:

   ```
   /* Global variables */
   uint16_t timerCounterValue[2] = {0u};
   uint16_t timerOverflowInterruptCount = 0u;
   ```

```
/* Callback function to get time interval in nano seconds */
uint32_t timerGetTimeIntervalCallback0(uint32_t *ns)
{
    timerCounterValue[1] = (uint16_t)(ftmBase->CNT);
    *ns = ((uint32_t)(timerCounterValue[1] + timerOverflowInterruptCount*65536u - timerCounterValue[0]))*10
      00 / TIMER_1US;
    timerOverflowInterruptCount = 0U;
    timerCounterValue[0] = timerCounterValue[1];
    return 0U;
}
```

7. This is a code example to set up a user LIN Driver configuration instantiation:

```
/* Device instance number as LPUART instance*/
#define LI0 (0U)

lin_state_t linState;
lin_user_config_t linUserConfig;
/* Set baudrate 19200 bps */
linUserConfig.baudRate = 19200U;
/* Node is MASTER */
linUserConfig.nodeFunction = MASTER;
/* Disable autobaud feature */
linUserConfig.autobaudEnable = FALSE;
/* Callback function to get time interval in nano seconds */
linUserConfig.timerGetTimeIntervalCallback = (lin_timer_get_time_t)
    timerGetTimeIntervalCallback0;

/* Initialize LIN Hardware interface */
LIN_DRV_Init(LI0, (lin_user_config_t *) &linUserConfig, (
    lin_state_t *) &linState);
```

8. The users are required to initialize a timer for LIN.
   E.g. a Flex Timer (FTM). FTM instance should be initialized in Output Compare mode with an interrupt(E.g. FTM0_Ch0_Ch1_IRQHandler) period of about 500 us. Users can choose a different interrupt period that is appropriate to their applications. In timer interrupt handler, users shall call LIN_DRV_TimeoutService to handle linCurrentState->timeoutCounter while sending or receiving data.

16.47.5 LIN Data Transfers

The driver implements transmit and receive functions to transfer buffers of data by blocking and non-blocking modes.

The blocking transmit and receive functions include LIN_DRV_SendFrameDataBlocking() and the LIN_DRV_↩ ReceiveFrameDataBlocking() functions.

The non-blocking (async) transmit and receive functions include the LIN_DRV_SendFrameData() and the LIN_D↩ RV_ReceiveFrameData() functions.

The LIN_DRV_ReceiveFrameData() function is recommended to be called in an interrupt event of receiving PID as implemented in LIN Stack middleware.

The LIN_DRV_ReceiveFrameData() and LIN_DRV_ReceiveFrameDataBlocking() functions should be called before data is transferring on the LIN bus. Otherwise, some data may be lost.

Master nodes can transmit frame headers in non-blocking mode using LIN_DRV_MasterSendHeader().

In all these cases, the functions are interrupt-driven.

16.47.6 Autobaud feature

AUTOBAUD is an extensive feature in LIN Driver which allows a slave node to automatically detect baudrate of LIN bus and adapt its original baudrate to bus value. Auto Baud is applied when the baudrate of the incoming data is unknown. Currently autobaud feature is supported to detect LIN bus baudrates 2400, 4800, 9600, 14400, 19200 bps.

1. If autobaud feature is enabled, at LIN driver initialization slave's baudrate is set to 19200bps. The application should use a timer interrupt in input capture mode of both rising and falling edges(E.g FTM), call LIN_DR↩ V_AutoBaudCapture(uint32_t instance) function to calculate and set Slave's baudrate like Master's baudrate.

When receiving a frame header, the slave detect LIN bus's baudrate based on the synchronization byte and adapts its baudrate accordingly. On changing baudrate, the slave set current event ID to LIN_BAUDRATE_↩ ADJUSTED and call the callback function. In that callback function users might change the frame data count timeout. Users can look at CallbackHandler() in lin.c of lin middleware for a reference.

Note: Lin driver should be initiated before initiating a timer interrupt(E.g FTM).

2. Baudrate evaluation process is executed until autobaud successfully. During run-time if LIN bus's baudrate is changed suddenly to a value other than the slave's current baudrate, users shall reset MCU to execute baudrate evaluation process.

Data Structures

- struct lin_user_config_t

  LIN hardware configuration structure Implements : lin_user_config_t_Class. More...

- struct lin_state_t

  Runtime state of the LIN driver. More...

Macros

- #define SLAVE 0U
- #define MASTER 1U
- #define MAKE_PARITY 0U
- #define CHECK_PARITY 1U

Typedefs

- typedef uint32_t(∗ lin_timer_get_time_interval_t) (uint32_t ∗nanoSeconds)

  Callback function to get time interval in nanoseconds Implements : lin_timer_get_time_interval_t_Class.

- typedef void(∗ lin_callback_t) (uint32_t instance, void ∗linState)

  LIN Driver callback function type Implements : lin_callback_t_Class.

Enumerations

- enum lin_event_id_t {
  LIN_NO_EVENT = 0x00U, LIN_WAKEUP_SIGNAL = 0x01U, LIN_BAUDRATE_ADJUSTED = 0x02U, LIN↩
  _RECV_BREAK_FIELD_OK = 0x03U,
  LIN_SYNC_OK = 0x04U, LIN_SYNC_ERROR = 0x05U, LIN_PID_OK = 0x06U, LIN_PID_ERROR = 0x07U,
  LIN_FRAME_ERROR = 0x08U, LIN_READBACK_ERROR = 0x09U, LIN_CHECKSUM_ERROR = 0x0AU,
  LIN_TX_COMPLETED = 0x0BU,
  LIN_RX_COMPLETED = 0x0CU, LIN_RX_OVERRUN = 0x0DU }

  Defines types for an enumerating event related to an Identifier. Implements : lin_event_id_t_Class.

- enum lin_node_state_t {
  LIN_NODE_STATE_UNINIT = 0x00U, LIN_NODE_STATE_SLEEP_MODE = 0x01U, LIN_NODE_STATE↩
  _IDLE = 0x02U, LIN_NODE_STATE_SEND_BREAK_FIELD = 0x03U,
  LIN_NODE_STATE_RECV_SYNC = 0x04U, LIN_NODE_STATE_SEND_PID = 0x05U, LIN_NODE_STA↩
  TE_RECV_PID = 0x06U, LIN_NODE_STATE_RECV_DATA = 0x07U,
  LIN_NODE_STATE_RECV_DATA_COMPLETED = 0x08U, LIN_NODE_STATE_SEND_DATA = 0x09U, L↩
  IN_NODE_STATE_SEND_DATA_COMPLETED = 0x0AU }

  Define type for an enumerating LIN Node state. Implements : lin_node_state_t_Class.

408 CONTENTS

LIN DRIVER

- status_t LIN_DRV_Init (uint32_t instance, lin_user_config_t ∗linUserConfig, lin_state_t ∗linCurrentState)

    Initializes an instance LIN Hardware Interface for LIN Network.

- status_t LIN_DRV_Deinit (uint32_t instance)

    Shuts down the LIN Hardware Interface by disabling interrupts and transmitter/receiver.

- lin_callback_t LIN_DRV_InstallCallback (uint32_t instance, lin_callback_t function)

    Installs callback function that is used for LIN_DRV_IRQHandler.

- status_t LIN_DRV_SendFrameDataBlocking (uint32_t instance, const uint8_t ∗txBuff, uint8_t txSize, uint32↩_t timeoutMSec)

    Sends Frame data out through the LIN Hardware Interface using blocking method. This function will calculate the checksum byte and send it with the frame data. Blocking means that the function does not return until the transmission is complete. This function checks if txSize is in range from 1 to 8. If not, it will return STATUS_ERROR. This function also returns STATUS_ERROR if node's current state is in SLEEP mode. This function checks if the isBusBusy is false, if not it will return LIN_BUS_BUSY. The function does not return until the transmission is complete. If the transmission is successful, it will return STATUS_SUCCESS. If not, it will return STATUS_TIMEOUT.

- status_t LIN_DRV_SendFrameData (uint32_t instance, const uint8_t ∗txBuff, uint8_t txSize)

    Sends frame data out through the LIN Hardware Interface using non-blocking method. This enables an a-sync method for transmitting data. Non-blocking means that the function returns immediately. The application has to get the transmit status to know when the transmit is complete. This function will calculate the checksum byte and send it with the frame data. The function will return immediately after calling this function. If txSize is equal to 0 or greater than 8 or node's current state is in SLEEP mode then the function will return STATUS_ERROR. If isBusBusy is currently true then the function will return LIN_BUS_BUSY.

- status_t LIN_DRV_GetTransmitStatus (uint32_t instance, uint8_t ∗bytesRemaining)

    Get status of an on-going non-blocking transmission While sending frame data using non-blocking method, users can use this function to get status of that transmission. The bytesRemaining shows number of bytes that still needed to transmit.

- status_t LIN_DRV_ReceiveFrameDataBlocking (uint32_t instance, uint8_t ∗rxBuff, uint8_t rxSize, uint32_t timeoutMSec)

    Receives frame data through the LIN Hardware Interface using blocking method. This function receives data from LPUART module using blocking method, the function does not return until the receive is complete. The interrupt handler LIN_LPUART_DRV_IRQHandler will check the checksum byte. If the checksum is correct, it will receive the frame data. If the checksum is incorrect, this function will return STATUS_TIMEOUT and data in rxBuff might be wrong. This function also check if rxSize is in range from 1 to 8. If not, it will return STATUS_ERROR. This function also returns STATUS_ERROR if node's current state is in SLEEP mode. This function checks if the isBusBusy is false, if not it will return LIN_BUS_BUSY.

- status_t LIN_DRV_ReceiveFrameData (uint32_t instance, uint8_t ∗rxBuff, uint8_t rxSize)

    Receives frame data through the LIN Hardware Interface using non-blocking method. This function will check the checksum byte. If the checksum is correct, it will receive it with the frame data. Non-blocking means that the function returns immediately. The application has to get the receive status to know when the reception is complete. The interrupt handler LIN_LPUART_DRV_IRQHandler will check the checksum byte. If the checksum is correct, it will receive the frame data. If the checksum is incorrect, this function will return STATUS_TIMEOUT and data in rxBuff might be wrong. This function also check if rxSize is in range from 1 to 8. If not, it will return STATUS_ERROR. This function also returns STATUS_ERROR if node's current state is in SLEEP mode. This function checks if the isBusBusy is false, if not it will return LIN_BUS_BUSY.

- status_t LIN_DRV_AbortTransferData (uint32_t instance)

    Aborts an on-going non-blocking transmission/reception. While performing a non-blocking transferring data, users can call this function to terminate immediately the transferring.

- status_t LIN_DRV_GetReceiveStatus (uint32_t instance, uint8_t ∗bytesRemaining)

    Get status of an on-going non-blocking reception. This function returns whether the data reception is complete. When performing non-blocking transmit, the user can call this function to ascertain the state of the current receive progress: in progress (STATUS_BUSY) or timeout (STATUS_TIMEOUT) or complete (STATUS_SUCCESS). In addition, if the reception is still in progress, the user can obtain the number of bytes that still needed to receive.

- status_t LIN_DRV_GoToSleepMode (uint32_t instance)

    Puts current LIN node to sleep mode This function changes current node state to LIN_NODE_STATE_SLEEP_M↩ODE.

- status_t LIN_DRV_GotoIdleState (uint32_t instance)

Generated on Thu Feb 8 2018 15:07:14 for S32SDK Documentation by Doxygen

Puts current LIN node to Idle state This function changes current node state to LIN_NODE_STATE_IDLE.

- status_t LIN_DRV_SendWakeupSignal (uint32_t instance)

  Sends a wakeup signal through the LIN Hardware Interface.

- lin_node_state_t LIN_DRV_GetCurrentNodeState (uint32_t instance)

  Get the current LIN node state.

- void LIN_DRV_TimeoutService (uint32_t instance)

  Callback function for Timer Interrupt Handler Users may use (optional, not required) LIN_DRV_TimeoutService to check if timeout has occurred during non-blocking frame data transmission and reception. User may initialize a timer (for example FTM) in Output Compare Mode with period of 500 micro seconds (recommended). In timer IRQ handler, call this function.

- void LIN_DRV_SetTimeoutCounter (uint32_t instance, uint32_t timeoutValue)

  Set Value for Timeout Counter that is used in LIN_DRV_TimeoutService.

- status_t LIN_DRV_MasterSendHeader (uint32_t instance, uint8_t id)

  Sends frame header out through the LIN Hardware Interface using a non-blocking method. This function sends LIN Break field, sync field then the ID with correct parity. This function checks if the interface is Master, if not, it will return STATUS_ERROR.This function checks if id is in range from 0 to 0x3F, if not it will return STATUS_ERROR.

- status_t LIN_DRV_EnableIRQ (uint32_t instance)

  Enables LIN hardware interrupts.

- status_t LIN_DRV_DisableIRQ (uint32_t instance)

  Disables LIN hardware interrupts.

- void LIN_DRV_IRQHandler (uint32_t instance)

  Interrupt handler for LIN Hardware Interface.

- uint8_t LIN_DRV_ProcessParity (uint8_t PID, uint8_t typeAction)

  Makes or checks parity bits. If action is checking parity, the function returns ID value if parity bits are correct or 0xFF if parity bits are incorrect. If action is making parity bits, then from input value of ID, the function returns PID. This is not a public API as it is called by other API functions.

- uint8_t LIN_DRV_MakeChecksumByte (const uint8_t ∗buffer, uint8_t sizeBuffer, uint8_t PID)

  Makes the checksum byte for a frame.

- status_t LIN_DRV_AutoBaudCapture (uint32_t instance)

  Captures time interval to capture baudrate automatically when enable autobaud feature. This function should only be used in Slave. The timer should be in input capture mode of both rising and falling edges. The timer input capture pin should be externally connected to RXD pin.

16.47.7 Data Structure Documentation

16.47.7.1 struct lin_user_config_t

LIN hardware configuration structure Implements : lin_user_config_t_Class.

Definition at line 66 of file lin_driver.h.

Data Fields

- uint32_t baudRate
- bool nodeFunction
- bool autobaudEnable
- lin_timer_get_time_interval_t timerGetTimeIntervalCallback

Field Documentation

16.47.7.1.1 bool autobaudEnable

Enable Autobaud feature

Definition at line 69 of file lin_driver.h.

16.47.7.1.2   uint32_t baudRate

baudrate of LIN Hardware Interface to configure

Definition at line 67 of file lin_driver.h.

16.47.7.1.3   bool nodeFunction

Node function as Master or Slave

Definition at line 68 of file lin_driver.h.

16.47.7.1.4   lin_timer_get_time_interval_t timerGetTimeIntervalCallback

Callback function to get time interval in nanoseconds

Definition at line 70 of file lin_driver.h.

16.47.7.2   struct lin_state_t

Runtime state of the LIN driver.

Note that the caller provides memory for the driver state structures during initialization because the driver does not statically allocate memory. Implements : lin_state_t_Class

Definition at line 125 of file lin_driver.h.

Data Fields

- const uint8_t ∗ txBuff
- uint8_t ∗ rxBuff
- uint8_t cntByte
- volatile uint8_t txSize
- volatile uint8_t rxSize
- uint8_t checkSum
- volatile bool isTxBusy
- volatile bool isRxBusy
- volatile bool isBusBusy
- volatile bool isTxBlocking
- volatile bool isRxBlocking
- lin_callback_t Callback
- uint8_t currentId
- uint8_t currentPid
- volatile lin_event_id_t currentEventId
- volatile lin_node_state_t currentNodeState
- volatile uint32_t timeoutCounter
- volatile bool timeoutCounterFlag
- volatile bool baudrateEvalEnable
- volatile uint8_t fallingEdgeInterruptCount
- uint32_t linSourceClockFreq
- semaphore_t txCompleted
- semaphore_t rxCompleted

Field Documentation

16.47.7.2.1   volatile bool baudrateEvalEnable

Baudrate Evaluation Process Enable

Definition at line 144 of file lin_driver.h.

16.47.7.2.2   lin_callback_t Callback

Callback function to invoke after receiving a byte or transmitting a byte.

Definition at line 137 of file lin_driver.h.

16.47.7.2.3   uint8_t checkSum

Checksum byte.

Definition at line 131 of file lin_driver.h.

16.47.7.2.4   uint8_t cntByte

To count number of bytes already transmitted or received.

Definition at line 128 of file lin_driver.h.

16.47.7.2.5   volatile lin_event_id_t currentEventId

Current ID Event

Definition at line 140 of file lin_driver.h.

16.47.7.2.6   uint8_t currentId

Current ID

Definition at line 138 of file lin_driver.h.

16.47.7.2.7   volatile lin_node_state_t currentNodeState

Current Node state

Definition at line 141 of file lin_driver.h.

16.47.7.2.8   uint8_t currentPid

Current PID

Definition at line 139 of file lin_driver.h.

16.47.7.2.9   volatile uint8_t fallingEdgeInterruptCount

Falling Edge count of a sync byte

Definition at line 145 of file lin_driver.h.

16.47.7.2.10   volatile bool isBusBusy

True if there are data, frame headers being transferred on bus

Definition at line 134 of file lin_driver.h.

16.47.7.2.11   volatile bool isRxBlocking

True if receive is blocking transaction.

Definition at line 136 of file lin_driver.h.

16.47.7.2.12   volatile bool isRxBusy

True if the LIN interface is receiving frame data.

Definition at line 133 of file lin_driver.h.

16.47.7.2.13 volatile bool isTxBlocking

True if transmit is blocking transaction.

Definition at line 135 of file lin_driver.h.

16.47.7.2.14 volatile bool isTxBusy

True if the LIN interface is transmitting frame data.

Definition at line 132 of file lin_driver.h.

16.47.7.2.15 uint32_t linSourceClockFreq

Frequency of the source clock for LIN

Definition at line 146 of file lin_driver.h.

16.47.7.2.16 uint8_t∗ rxBuff

The buffer of received data.

Definition at line 127 of file lin_driver.h.

16.47.7.2.17 semaphore_t rxCompleted

Used to wait for LIN interface ISR to complete reception

Definition at line 148 of file lin_driver.h.

16.47.7.2.18 volatile uint8_t rxSize

The remaining number of bytes to be received.

Definition at line 130 of file lin_driver.h.

16.47.7.2.19 volatile uint32_t timeoutCounter

Value of the timeout counter

Definition at line 142 of file lin_driver.h.

16.47.7.2.20 volatile bool timeoutCounterFlag

Timeout counter flag

Definition at line 143 of file lin_driver.h.

16.47.7.2.21 const uint8_t∗ txBuff

The buffer of data being sent.

Definition at line 126 of file lin_driver.h.

16.47.7.2.22 semaphore_t txCompleted

Used to wait for LIN interface ISR to complete transmission.

Definition at line 147 of file lin_driver.h.

16.47.7.2.23 volatile uint8_t txSize

The remaining number of bytes to be transmitted.

Definition at line 129 of file lin_driver.h.

16.47.8 Macro Definition Documentation

16.47.8.1 #define CHECK_PARITY 1U

Definition at line 53 of file lin_driver.h.

16.47.8.2 #define MAKE_PARITY 0U

Definition at line 52 of file lin_driver.h.

16.47.8.3 #define MASTER 1U

Definition at line 51 of file lin_driver.h.

16.47.8.4 #define SLAVE 0U

Definition at line 50 of file lin_driver.h.

16.47.9 Typedef Documentation

16.47.9.1 typedef void(∗ lin_callback_t) (uint32_t instance, void ∗linState)

LIN Driver callback function type Implements : lin_callback_t_Class.

Definition at line 116 of file lin_driver.h.

16.47.9.2 typedef uint32_t(∗ lin_timer_get_time_interval_t) (uint32_t ∗nanoSeconds)

Callback function to get time interval in nanoseconds Implements : lin_timer_get_time_interval_t_Class.

Definition at line 60 of file lin_driver.h.

16.47.10 Enumeration Type Documentation

16.47.10.1 enum lin_event_id_t

Defines types for an enumerating event related to an Identifier. Implements : lin_event_id_t_Class.

Enumerator

    LIN_NO_EVENT No event yet
    LIN_WAKEUP_SIGNAL Received a wakeup signal
    LIN_BAUDRATE_ADJUSTED Indicate that baudrate was adjusted to Master's baudrate
    LIN_RECV_BREAK_FIELD_OK Indicate that correct Break Field was received
    LIN_SYNC_OK Sync byte is correct
    LIN_SYNC_ERROR Sync byte is incorrect
    LIN_PID_OK PID correct
    LIN_PID_ERROR PID incorrect
    LIN_FRAME_ERROR Framing Error
    LIN_READBACK_ERROR Readback data is incorrect
    LIN_CHECKSUM_ERROR Checksum byte is incorrect
    LIN_TX_COMPLETED Sending data completed
    LIN_RX_COMPLETED Receiving data completed
    LIN_RX_OVERRUN RX overrun flag

Definition at line 77 of file lin_driver.h.

16.47.10.2   enum lin_node_state_t

Define type for an enumerating LIN Node state. Implements : lin_node_state_t_Class.

Enumerator

       LIN_NODE_STATE_UNINIT   Uninitialized state

       LIN_NODE_STATE_SLEEP_MODE   Sleep mode state

       LIN_NODE_STATE_IDLE   Idle state

       LIN_NODE_STATE_SEND_BREAK_FIELD   Send break field state

       LIN_NODE_STATE_RECV_SYNC   Receive the synchronization byte state

       LIN_NODE_STATE_SEND_PID   Send PID state

       LIN_NODE_STATE_RECV_PID   Receive PID state

       LIN_NODE_STATE_RECV_DATA   Receive data state

       LIN_NODE_STATE_RECV_DATA_COMPLETED   Receive data completed state

       LIN_NODE_STATE_SEND_DATA   Send data state

       LIN_NODE_STATE_SEND_DATA_COMPLETED   Send data completed state

Definition at line 98 of file lin_driver.h.

16.47.11   Function Documentation

16.47.11.1   status_t LIN_DRV_AbortTransferData (uint32_t *instance*)

Aborts an on-going non-blocking transmission/reception. While performing a non-blocking transferring data, users can call this function to terminate immediately the transferring.

Parameters

instance	LIN Hardware Interface instance number

Returns

      function always return STATUS_SUCCESS

Definition at line 257 of file lin_driver.c.

16.47.11.2   status_t LIN_DRV_AutoBaudCapture (uint32_t *instance*)

Captures time interval to capture baudrate automatically when enable autobaud feature. This function should only be used in Slave. The timer should be in input capture mode of both rising and falling edges. The timer input capture pin should be externally connected to RXD pin.

Parameters

instance	LIN Hardware Interface instance number

Returns

    operation status

- STATUS_SUCCESS: Operation was successful.
- STATUS_BUSY: Operation is running.
- STATUS_ERROR: Operation failed due to break char incorrect, wakeup signal incorrect or calculate baudrate failed.

Definition at line 484 of file lin_driver.c.

16.47.11.3   status_t LIN_DRV_Deinit (uint32_t *instance*)

Shuts down the LIN Hardware Interface by disabling interrupts and transmitter/receiver.

Parameters

instance	LIN Hardware Interface instance number

Returns

> operation status:
> - STATUS_SUCCESS : Operation was successful.
> - STATUS_ERROR : Operation failed due to destroy TX and RX semaphores.

Definition at line 83 of file lin_driver.c.

16.47.11.4 status_t LIN_DRV_DisableIRQ (uint32_t *instance*)

Disables LIN hardware interrupts.

Parameters

instance	LIN Hardware Interface instance number

Returns

> function always return STATUS_SUCCESS.

Definition at line 446 of file lin_driver.c.

16.47.11.5 status_t LIN_DRV_EnableIRQ (uint32_t *instance*)

Enables LIN hardware interrupts.

Parameters

instance	LIN Hardware Interface instance number.

Returns

> function always return STATUS_SUCCESS.

Definition at line 428 of file lin_driver.c.

16.47.11.6 lin_node_state_t LIN_DRV_GetCurrentNodeState (uint32_t *instance*)

Get the current LIN node state.

Parameters

instance	LIN Hardware Interface instance number

Returns

> current LIN node state

Definition at line 354 of file lin_driver.c.

16.47.11.7 status_t LIN_DRV_GetReceiveStatus (uint32_t *instance,* uint8_t ∗ *bytesRemaining*)

Get status of an on-going non-blocking reception. This function returns whether the data reception is complete. When performing non-blocking transmit, the user can call this function to ascertain the state of the current receive progress: in progress (STATUS_BUSY) or timeout (STATUS_TIMEOUT) or complete (STATUS_SUCCESS). In addition, if the reception is still in progress, the user can obtain the number of bytes that still needed to receive.

Parameters

instance	LIN Hardware Interface instance number
bytesRemaining	Number of bytes still needed to receive

Returns

> operation status:
>
>   - STATUS_SUCCESS : The reception is complete.
>
>   - STATUS_TIMEOUT : The reception isn't complete.
>
>   - STATUS_BUSY : The reception is on going

Definition at line 280 of file lin_driver.c.

16.47.11.8   status_t LIN_DRV_GetTransmitStatus (uint32_t *instance,* uint8_t ∗ *bytesRemaining*)

Get status of an on-going non-blocking transmission While sending frame data using non-blocking method, users can use this function to get status of that transmission. The bytesRemaining shows number of bytes that still needed to transmit.

Parameters

instance	LIN Hardware Interface instance number
bytesRemaining	Number of bytes still needed to transmit

Returns

> operation status:
>
>   - STATUS_SUCCESS : The transmission is successful.
>
>   - STATUS_BUSY : The transmission is sending
>
>   - STATUS_TIMEOUT : Operation failed due to timeout has occurred.

Definition at line 178 of file lin_driver.c.

16.47.11.9   status_t LIN_DRV_GotoIdleState (uint32_t *instance*)

Puts current LIN node to Idle state This function changes current node state to LIN_NODE_STATE_IDLE.

Parameters

instance	LIN Hardware Interface instance number

Returns

> function always return STATUS_SUCCESS

Definition at line 318 of file lin_driver.c.

16.47.11.10   status_t LIN_DRV_GoToSleepMode (uint32_t *instance*)

Puts current LIN node to sleep mode This function changes current node state to LIN_NODE_STATE_SLEEP_↩
MODE.

Parameters

instance	LIN Hardware Interface instance number

Returns

> function always return STATUS_SUCCESS

Definition at line 300 of file lin_driver.c.

16.47.11.11   status_t LIN_DRV_Init (uint32_t *instance,* lin_user_config_t ∗ *linUserConfig,* lin_state_t ∗ *linCurrentState*)

Initializes an instance LIN Hardware Interface for LIN Network.

The caller provides memory for the driver state structures during initialization. The user must select the LIN Hardware Interface clock source in the application to initialize the LIN Hardware Interface.

Parameters

instance	LIN Hardware Interface instance number
linUserConfig	user configuration structure of type lin_user_config_t
linCurrentState	pointer to the LIN Hardware Interface driver state structure

Returns

> operation status:
>
> - STATUS_SUCCESS : Operation was successful.
> - STATUS_ERROR : Operation failed due to semaphores initialize error.

Definition at line 62 of file lin_driver.c.

16.47.11.12   lin_callback_t LIN_DRV_InstallCallback (uint32_t *instance,* lin_callback_t *function*)

Installs callback function that is used for LIN_DRV_IRQHandler.

Note

> After a callback is installed, it bypasses part of the LIN Hardware Interface IRQHandler logic. Therefore, the callback needs to handle the indexes of txBuff and txSize.

Parameters

instance	LIN Hardware Interface instance number.
function	the LIN receive callback function.

Returns

> Former LIN callback function pointer.

Definition at line 102 of file lin_driver.c.

16.47.11.13   void LIN_DRV_IRQHandler (uint32_t *instance*)

Interrupt handler for LIN Hardware Interface.

Parameters

instance	LIN Hardware Interface instance number

Returns

> void

Definition at line 466 of file lin_driver.c.

16.47.11.14 **uint8_t LIN_DRV_MakeChecksumByte (const uint8_t** ∗ *buffer,* **uint8_t** *sizeBuffer,* **uint8_t** *PID* **)**

Makes the checksum byte for a frame.

Parameters

buffer	Pointer to Tx buffer
sizeBuffer	Number of bytes that are contained in the buffer.
PID	Protected Identifier byte.

Returns

> the checksum byte.

Definition at line 102 of file lin_common.c.

16.47.11.15 status_t LIN_DRV_MasterSendHeader (uint32_t *instance,* uint8_t *id*)

Sends frame header out through the LIN Hardware Interface using a non-blocking method. This function sends LIN Break field, sync field then the ID with correct parity. This function checks if the interface is Master, if not, it will return STATUS_ERROR.This function checks if id is in range from 0 to 0x3F, if not it will return STATUS_ERROR.

Parameters

instance	LIN Hardware Interface instance number
id	Frame Identifier

Returns

> operation status:
>
> - STATUS_SUCCESS : The transmission is successful.
> - STATUS_BUSY : Bus busy flag is true.
> - STATUS_ERROR : The interface isn't Master or id isn't in range from 0 to 0x3F or node's current state is in SLEEP mode.

Definition at line 409 of file lin_driver.c.

16.47.11.16 uint8_t LIN_DRV_ProcessParity (uint8_t *PID,* uint8_t *typeAction*)

Makes or checks parity bits. If action is checking parity, the function returns ID value if parity bits are correct or 0xFF if parity bits are incorrect. If action is making parity bits, then from input value of ID, the function returns PID. This is not a public API as it is called by other API functions.

Parameters

PID	PID byte in case of checking parity bits or ID byte in case of making parity bits.
typeAction	1 for Checking parity bits, 0 for making parity bits

Returns

> Value has 8 bit:
>
> - 0xFF : Parity bits are incorrect,
> - ID : Checking parity bits are correct.
> - PID : typeAction is making parity bits.

Definition at line 58 of file lin_common.c.

16.47.11.17 status_t LIN_DRV_ReceiveFrameData (uint32_t *instance,* uint8_t ∗ *rxBuff,* uint8_t *rxSize*)

Receives frame data through the LIN Hardware Interface using non-blocking method. This function will check the checksum byte. If the checksum is correct, it will receive it with the frame data. Non-blocking means that the function returns immediately. The application has to get the receive status to know when the reception is complete. The interrupt handler LIN_LPUART_DRV_IRQHandler will check the checksum byte. If the checksum is correct, it will

receive the frame data. If the checksum is incorrect, this function will return STATUS_TIMEOUT and data in rxBuff might be wrong. This function also check if rxSize is in range from 1 to 8. If not, it will return STATUS_ERROR. This function also returns STATUS_ERROR if node's current state is in SLEEP mode. This function checks if the isBusBusy is false, if not it will return LIN_BUS_BUSY.

Note

> If users use LIN_DRV_TimeoutService in a timer interrupt handler, then before using this function, users have to set timeout counter to an appropriate value by using LIN_DRV_SetTimeoutCounter(instance, timeout↩ Value). The timeout value should be big enough to complete the reception. Timeout in real time is (timeout↩ Value) ∗ (time period that LIN_DRV_TimeoutService is called). For example, if LIN_DRV_TimeoutService is called in an timer interrupt with period of 500 micro seconds, then timeout in real time is timeoutValue ∗ 500 micro seconds.

Parameters

instance	LIN Hardware Interface instance number
rxBuff	buffer containing 8-bit received data
rxSize	the number of bytes to receive

Returns

> operation status:
>
> - STATUS_SUCCESS : The receives frame data is successful.
> - STATUS_TIMEOUT : The checksum is incorrect.
> - STATUS_BUSY : Bus busy flag is true.
> - STATUS_ERROR : Operation failed due is equal to 0 or greater than 8 or node's current state is in SLEEP mode

Definition at line 235 of file lin_driver.c.

16.47.11.18 status_t LIN_DRV_ReceiveFrameDataBlocking (uint32_t *instance,* uint8_t ∗ *rxBuff,* uint8_t *rxSize,* uint32_t *timeoutMSec*)

Receives frame data through the LIN Hardware Interface using blocking method. This function receives data from LPUART module using blocking method, the function does not return until the receive is complete. The interrupt handler LIN_LPUART_DRV_IRQHandler will check the checksum byte. If the checksum is correct, it will receive the frame data. If the checksum is incorrect, this function will return STATUS_TIMEOUT and data in rxBuff might be wrong. This function also check if rxSize is in range from 1 to 8. If not, it will return STATUS_ERROR. This function also returns STATUS_ERROR if node's current state is in SLEEP mode. This function checks if the isBusBusy is false, if not it will return LIN_BUS_BUSY.

Parameters

instance	LIN Hardware Interface instance number
rxBuff	buffer containing 8-bit received data
rxSize	the number of bytes to receive
timeoutMSec	timeout value in milliseconds

Returns

> operation status:
>
> - STATUS_SUCCESS : The receives frame data is successful.
> - STATUS_TIMEOUT : The checksum is incorrect.
> - STATUS_BUSY : Bus busy flag is true.
> - STATUS_ERROR : Operation failed due is equal to 0 or greater than 8 or node's current state is in SLEEP mode

Definition at line 205 of file lin_driver.c.

16.47.11.19   status_t LIN_DRV_SendFrameData (uint32_t *instance,* const uint8_t ∗ *txBuff,* uint8_t *txSize*)

Sends frame data out through the LIN Hardware Interface using non-blocking method. This enables an a-sync method for transmitting data. Non-blocking means that the function returns immediately. The application has to get the transmit status to know when the transmit is complete. This function will calculate the checksum byte and send it with the frame data. The function will return immediately after calling this function. If txSize is equal to 0 or greater than 8 or node's current state is in SLEEP mode then the function will return STATUS_ERROR. If isBusBusy is currently true then the function will return LIN_BUS_BUSY.

Note

> If users use LIN_DRV_TimeoutService in a timer interrupt handler, then before using this function, users have to set timeout counter to an appropriate value by using LIN_DRV_SetTimeoutCounter(instance, timeout↩ Value). The timeout value should be big enough to complete the transmission. Timeout in real time is (timeoutValue) ∗ (time period that LIN_DRV_TimeoutService is called). For example, if LIN_DRV_Timeout↩ Service is called in an timer interrupt with period of 500 micro seconds, then timeout in real time is timeout↩ Value ∗ 500 micro seconds.

Parameters

instance	LIN Hardware Interface instance number
txBuff	source buffer containing 8-bit data chars to send
txSize	the number of bytes to send

Returns

> operation status:
>
> - STATUS_SUCCESS : The transmission is successful.
> - STATUS_BUSY : Operation failed due to isBusBusy is currently true.
> - STATUS_ERROR : Operation failed due to txSize is equal to 0 or greater than 8 or node's current state is in SLEEP mode

Definition at line 153 of file lin_driver.c.

16.47.11.20   status_t LIN_DRV_SendFrameDataBlocking (uint32_t *instance,* const uint8_t ∗ *txBuff,* uint8_t *txSize,* uint32_t *timeoutMSec*)

Sends Frame data out through the LIN Hardware Interface using blocking method. This function will calculate the checksum byte and send it with the frame data. Blocking means that the function does not return until the transmission is complete. This function checks if txSize is in range from 1 to 8. If not, it will return STATUS_ER↩ ROR. This function also returns STATUS_ERROR if node's current state is in SLEEP mode. This function checks if the isBusBusy is false, if not it will return LIN_BUS_BUSY. The function does not return until the transmission is complete. If the transmission is successful, it will return STATUS_SUCCESS. If not, it will return STATUS_TIME↩ OUT.

Parameters

instance	LIN Hardware Interface instance number
txBuff	source buffer containing 8-bit data chars to send
txSize	the number of bytes to send
timeoutMSec	timeout value in milliseconds

Returns

> operation status:
>
> - STATUS_SUCCESS : The transmission is successful.
> - STATUS_TIMEOUT : The transmission isn't successful.

Definition at line 128 of file lin_driver.c.

16.47.11.21 status_t LIN_DRV_SendWakeupSignal (uint32_t *instance*)

Sends a wakeup signal through the LIN Hardware Interface.

Parameters

instance	LIN Hardware Interface instance number

Returns

operation status:

- STATUS_SUCCESS : Bus busy flag is false.
- STATUS_BUSY : Bus busy flag is true.

Definition at line 336 of file lin_driver.c.

16.47.11.22   void LIN_DRV_SetTimeoutCounter (uint32_t *instance,* uint32_t *timeoutValue*)

Set Value for Timeout Counter that is used in LIN_DRV_TimeoutService.

Parameters

instance	LIN Hardware Interface instance number
timeoutValue	Timeout Value to be set

Returns

void

Definition at line 389 of file lin_driver.c.

16.47.11.23   void LIN_DRV_TimeoutService (uint32_t *instance*)

Callback function for Timer Interrupt Handler Users may use (optional, not required) LIN_DRV_TimeoutService to check if timeout has occurred during non-blocking frame data transmission and reception. User may initialize a timer (for example FTM) in Output Compare Mode with period of 500 micro seconds (recommended). In timer IRQ handler, call this function.

Parameters

instance	LIN Hardware Interface instance number

Returns

void

Definition at line 374 of file lin_driver.c.

16.48 LIN Stack

16.48.1 Detailed Description

This section covers the functionality of the LIN Stack middleware layer in S32 SDK.

Introduction

LIN Stack Package Components

LIN Stack is a Middleware package that supports the LIN 2.1 and above, `LIN2.1` and `J2602` specifications. In LIN Stack, LIN 2.1 covers all LIN 2.1, LIN 2.2 and LIN 2.2A specifications, as the changes following LIN 2.1 are only spelling corrections and clarifications.

- 1. **LIN Stack:**

  The layered architecture of the LIN Stack is shown on `Figure 1`. Such architecture aims maximum reusability of common code base for `LIN2.1` and `J2602` specifications for S32 Freescale automotive MCU portfolio.

  The core API layer of `LIN2.1`/ `J2602` handles initialization, processing and signal based interaction between applications and LIN Core.

  The `LIN2.1` TL (Transport Layer) provides methods for diagnostic services.

  The low level layer offers methods for handling signal transmission between user applications and hardware such as interface initialization and deinitialization, frame header sending, response receiving, etc. The low level layer is builded on top of `LIN Driver` which is builded on top of LPUART HAL layer in the current release.

  Figure 1. LIN Stack Architecture diagram

- 2. **Node Configuration Tool:**

  To generate configuration files, users can use the Node Configuration Tool that is LIN Stack PE↩ X component which allows to parse existed LDF files and reflect their contents to LIN Stack component GUI, to create new LDF files, to configure LIN cluster definitions and Node definitions. Using LIN Stack PEX component, users can easily generate the node configuration files (lin_cfg.h and lin_cfg.c) that are needed for LIN Stack to work properly.

  `Figure 2`. Shows the diagram of configuration data flow.

  Figure 2. Configuration data

  The LDF files describe complete LIN cluster definition including Master/slave mode definition, signals, frames, schedules, timing, etc.

Modules

- Diagnostic services

  Diagnostic services defines methods to implement diagnostic data transfer between a master node connected with a diagnostic tester and the slave nodes.

- LIN Core API

  The LIN core API handles initialization, processing and a signal based interaction between the application and the LIN core. Refer to chapter 7, LIN 2.2A specification.

- Low level API

  Low level layer consists of functions that call LIN driver API.

- Transport layer API

  Transport layer stands between the application layer and the core API layer.

16.49   LPI2C Driver

16.49.1   Detailed Description

Low Power Inter-Integrated Circuit (LPI2C) Peripheral Driver.

Low Power Inter-Integrated Circuit Driver.

The LPI2C driver allows communication on an I2C bus using the LPI2C module in the S32144K processor.

Features

- Interrupt based

- Master or slave operation

- Provides blocking and non-blocking transmit and receive functions

- 7-bit or 10-bit addressing

- Configurable baud rate

- Provides support for all operating modes supported by the hardware

    - Standard-mode (Sm): bidirectional data transfers up to 100 kbit/s

    - Fast-mode (Fm): bidirectional data transfers up to 400 kbit/s

Functionality

In order to use the LPI2C driver it must be first initialized in either master of slave mode, using functions LPI2↩C_DRV_MasterInit() or LPI2C_DRV_SlaveInit(). Once initialized, it cannot be initialized again for the same LPI2C module instance until it is de-initialized, using LPI2C_DRV_MasterDeinit() or LPI2C_DRV_SlaveDeinit(). Different LPI2C module instances can function independently of each other.

Master Mode

Master Mode provides functions for transmitting or receiving data to/from any I2C slave. Slave address and baud rate are provided at initialization time through the master configuration structure, but they can be changed at runtime by using LPI2C_DRV_MasterSetBaudRate() or LPI2C_DRV_MasterSetSlaveAddr(). Note that due to module limitation not any baud rate can be achieved. The driver will set a baud rate as close as possible to the requested baud rate, but there may still be substantial differences, for example if requesting a high baud rate while using a low-frequency protocol clock for the LPI2C module. The application should call LPI2C_DRV_MasterGetBaudRate() after LPI2C_DRV_MasterSetBaudRate() to check what baud rate was actually set.

To send or receive data to/from the currently configured slave address, use functions LPI2C_DRV_MasterSend↩Data() or LPI2C_DRV_MasterReceiveData() (or their blocking counterparts). Parameter `sendStop` can be used to chain multiple transfers with repeated START condition between them, for example when sending a command and then immediately receiving a response. The application should ensure that any send or receive transfer with `sendStop` set to `false` is followed by another transfer, otherwise the LPI2C master will hold the SCL line low indefinitely and block the I2C bus. The last transfer from a chain should always have `sendStop` set to `true`.

Blocking operations will return only when the transfer is completed, either successfully or with error. Non-blocking operations will initiate the transfer and return STATUS_SUCCESS, but the module is still busy with the transfer and another transfer can't be initiated until the current transfer is complete. The application can check the status of the current transfer by calling LPI2C_DRV_MasterGetTransferStatus(). If the transfer is completed, the functions will return either STATUS_SUCCESS or an error code, depending on the outcome of the last transfer.

The driver supports any operating mode supported by the module. The operating mode is set together with the baud rate, by LPI2C_DRV_MasterSetBaudRate(). For High-Speed mode a second baud rate is required, for high-speed communication. Note that due to module limitation (common prescaler setting for normal and fast baud rate) there is a limit on the maximum difference between the two baud rates. LPI2C_DRV_MasterGetBaudRate() can be used to check the baud rate setting for both modes.

Slave Mode

Slave Mode provides functions for transmitting or receiving data to/from any I2C master. There are two slave operating modes, selected by the field `slaveListening` in the slave configuration structure:

- Slave always listening: the slave interrupt is enabled at initialization time and the slave always listens to the line for a master addressing it. Any events are reported to the application through the callback function provided at initialization time. The callback can use LPI2C_DRV_SlaveSetRxBuffer() or LPI2C_DRV_Slave↩SetTxBuffer() to provide the appropriate buffers for transmit or receive, as needed.

- On-demand operation: the slave is commanded to transmit or receive data through the call of LPI2C_↩DRV_SlaveSendData() and LPI2C_DRV_SlaveReceiveData() (or their blocking counterparts). The actual moment of the transfer depends on the I2C master. The use of callbacks optional in this case, for example to treat events like LPI2C_SLAVE_EVENT_TX_EMPTY or LPI2C_SLAVE_EVENT_RX_FULL. Outside the commanded receive / transmit operations the LPI2C interrupts are disabled and the module will not react to master transfer requests.

Important Notes

- Before using the LPI2C driver in master mode the protocol clock of the module must be configured. Refer to SCG HAL and PCC HAL for clock configuration.

- Before using the LPI2C driver the pins must be routed to the LPI2C module. Refer to PORT HAL for pin routing configuration.

- The driver enables the interrupts for the corresponding LPI2C module, but any interrupt priority setting must be done by the application.

- Fast+, high-speed and ultra-fast mode aren't supported.

- Aborting a master reception is not currently supported due to hardware behavior (the module will continue a started reception even if the FIFO is reset).

- In listening mode, the init function must be called before the master starts the transfer. In non-listening mode, the init function and the appropriate send/receive function must be called before the master starts the transfer.

Data Structures

- struct lpi2c_master_user_config_t

    Defines the example structure. More...
- struct lpi2c_slave_user_config_t

    Slave configuration structure. More...
- struct lpi2c_baud_rate_params_t

    Baud rate structure. More...
- struct lpi2c_master_state_t

    Master internal context structure. More...
- struct lpi2c_slave_state_t

    Slave internal context structure. More...

Enumerations

- enum lpi2c_mode_t { LPI2C_STANDARD_MODE = 0x0U, LPI2C_FAST_MODE = 0x1U }

    I2C operating modes Implements : lpi2c_mode_t_Class.
- enum lpi2c_transfer_type_t { LPI2C_USING_DMA = 0, LPI2C_USING_INTERRUPTS = 1 }

    Type of LPI2C transfer (based on interrupts or DMA). Implements : lpi2c_transfer_type_t_Class.

LPI2C Driver

- status_t LPI2C_DRV_MasterInit (uint32_t instance, const lpi2c_master_user_config_t ∗userConfigPtr, lpi2c_master_state_t ∗master)

    Initialize the LPI2C master mode driver.

- status_t LPI2C_DRV_MasterDeinit (uint32_t instance)

    De-initialize the LPI2C master mode driver.

- void LPI2C_DRV_MasterGetBaudRate (uint32_t instance, lpi2c_baud_rate_params_t ∗baudRate)

    Get the currently configured baud rate.

- void LPI2C_DRV_MasterSetBaudRate (uint32_t instance, const lpi2c_mode_t operatingMode, const lpi2c←_baud_rate_params_t baudRate)

    Set the baud rate for any subsequent I2C communication.

- void LPI2C_DRV_MasterSetSlaveAddr (uint32_t instance, const uint16_t address, const bool is10bitAddr)

    Set the slave address for any subsequent I2C communication.

- status_t LPI2C_DRV_MasterSendData (uint32_t instance, const uint8_t ∗txBuff, uint32_t txSize, bool send←Stop)

    Perform a non-blocking send transaction on the I2C bus.

- status_t LPI2C_DRV_MasterSendDataBlocking (uint32_t instance, const uint8_t ∗txBuff, uint32_t txSize, bool sendStop, uint32_t timeout)

    Perform a blocking send transaction on the I2C bus.

- status_t LPI2C_DRV_MasterAbortTransferData (uint32_t instance)

    Abort a non-blocking I2C Master transmission or reception.

- status_t LPI2C_DRV_MasterReceiveData (uint32_t instance, uint8_t ∗rxBuff, uint32_t rxSize, bool sendStop)

    Perform a non-blocking receive transaction on the I2C bus.

- status_t LPI2C_DRV_MasterReceiveDataBlocking (uint32_t instance, uint8_t ∗rxBuff, uint32_t rxSize, bool sendStop, uint32_t timeout)

    Perform a blocking receive transaction on the I2C bus.

- status_t LPI2C_DRV_MasterGetTransferStatus (uint32_t instance, uint32_t ∗bytesRemaining)

    Return the current status of the I2C master transfer.

- void LPI2C_DRV_MasterIRQHandler (uint32_t instance)

    Handle master operation when I2C interrupt occurs.

- status_t LPI2C_DRV_SlaveInit (uint32_t instance, const lpi2c_slave_user_config_t ∗userConfigPtr, lpi2c_←slave_state_t ∗slave)

    Initialize the I2C slave mode driver.

- status_t LPI2C_DRV_SlaveDeinit (uint32_t instance)

    De-initialize the I2C slave mode driver.

- status_t LPI2C_DRV_SlaveSetTxBuffer (uint32_t instance, const uint8_t ∗txBuff, uint32_t txSize)

    Provide a buffer for transmitting data.

- status_t LPI2C_DRV_SlaveSetRxBuffer (uint32_t instance, uint8_t ∗rxBuff, uint32_t rxSize)

    Provide a buffer for receiving data.

- status_t LPI2C_DRV_SlaveSendData (uint32_t instance, const uint8_t ∗txBuff, uint32_t txSize)

    Perform a non-blocking send transaction on the I2C bus.

- status_t LPI2C_DRV_SlaveSendDataBlocking (uint32_t instance, const uint8_t ∗txBuff, uint32_t txSize, uint32_t timeout)

    Perform a blocking send transaction on the I2C bus.

- status_t LPI2C_DRV_SlaveReceiveData (uint32_t instance, uint8_t ∗rxBuff, uint32_t rxSize)

    Perform a non-blocking receive transaction on the I2C bus.

- status_t LPI2C_DRV_SlaveReceiveDataBlocking (uint32_t instance, uint8_t ∗rxBuff, uint32_t rxSize, uint32←_t timeout)

    Perform a blocking receive transaction on the I2C bus.

- status_t LPI2C_DRV_SlaveGetTransferStatus (uint32_t instance, uint32_t ∗bytesRemaining)

    Return the current status of the I2C slave transfer.

- status_t LPI2C_DRV_SlaveAbortTransferData (uint32_t instance)

    Abort a non-blocking I2C Master transmission or reception.

- void LPI2C_DRV_SlaveIRQHandler (uint32_t instance)

    Handle slave operation when I2C interrupt occurs.

16.49.2 Data Structure Documentation

16.49.2.1 struct lpi2c_master_user_config_t

Defines the example structure.

This structure is used as an example.

Master configuration structure

This structure is used to provide configuration parameters for the LPI2C master at initialization time. Implements : lpi2c_master_user_config_t_Class

Definition at line 114 of file lpi2c_driver.h.

Data Fields

- uint16_t slaveAddress
- bool is10bitAddr
- lpi2c_mode_t operatingMode
- uint32_t baudRate
- lpi2c_transfer_type_t transferType
- uint8_t dmaChannel
- i2c_master_callback_t masterCallback
- void ∗ callbackParam

Field Documentation

16.49.2.1.1 uint32_t baudRate

The baud rate in hertz to use with current slave device

Definition at line 119 of file lpi2c_driver.h.

16.49.2.1.2 void∗ callbackParam

Parameter for the master callback function

Definition at line 130 of file lpi2c_driver.h.

16.49.2.1.3 uint8_t dmaChannel

Channel number for DMA channel. If DMA mode isn't used this field will be ignored.

Definition at line 125 of file lpi2c_driver.h.

16.49.2.1.4 bool is10bitAddr

Selects 7-bit or 10-bit slave address

Definition at line 117 of file lpi2c_driver.h.

16.49.2.1.5 i2c_master_callback_t masterCallback

Master callback function. Note that this function will be called from the interrupt service routine at the end of a transfer, so its execution time should be as small as possible. It can be NULL if you want to check manually the status of the transfer.

Definition at line 126 of file lpi2c_driver.h.

16.49.2.1.6   lpi2c_mode_t operatingMode

I2C Operating mode

Definition at line 118 of file lpi2c_driver.h.

16.49.2.1.7   uint16_t slaveAddress

Slave address, 7-bit or 10-bit

Definition at line 116 of file lpi2c_driver.h.

16.49.2.1.8   lpi2c_transfer_type_t transferType

Type of LPI2C transfer

Definition at line 124 of file lpi2c_driver.h.

16.49.2.2   struct lpi2c_slave_user_config_t

Slave configuration structure.

This structure is used to provide configuration parameters for the LPI2C slave at initialization time. Implements :
lpi2c_slave_user_config_t_Class

Definition at line 139 of file lpi2c_driver.h.

Data Fields

- uint16_t slaveAddress
- bool is10bitAddr
- lpi2c_mode_t operatingMode
- bool slaveListening
- lpi2c_transfer_type_t transferType
- uint8_t dmaChannel
- i2c_slave_callback_t slaveCallback
- void ∗ callbackParam

Field Documentation

16.49.2.2.1   void∗ callbackParam

Parameter for the slave callback function

Definition at line 152 of file lpi2c_driver.h.

16.49.2.2.2   uint8_t dmaChannel

Channel number for DMA rx channel. If DMA mode isn't used this field will be ignored.

Definition at line 146 of file lpi2c_driver.h.

16.49.2.2.3   bool is10bitAddr

Selects 7-bit or 10-bit slave address

Definition at line 142 of file lpi2c_driver.h.

16.49.2.2.4   lpi2c_mode_t operatingMode

I2C Operating mode

Definition at line 143 of file lpi2c_driver.h.

16.49.2.2.5   uint16_t slaveAddress

Slave address, 7-bit or 10-bit

Definition at line 141 of file lpi2c_driver.h.

16.49.2.2.6   i2c_slave_callback_t slaveCallback

Slave callback function. Note that this function will be called from the interrupt service routine, so its execution time should be as small as possible. It can be NULL if the slave is not in listening mode (slaveListening = false)

Definition at line 147 of file lpi2c_driver.h.

16.49.2.2.7   bool slaveListening

Slave mode (always listening or on demand only)

Definition at line 144 of file lpi2c_driver.h.

16.49.2.2.8   lpi2c_transfer_type_t transferType

Type of LPI2C transfer

Definition at line 145 of file lpi2c_driver.h.

16.49.2.3   struct lpi2c_baud_rate_params_t

Baud rate structure.

This structure is used for setting or getting the baud rate. Implements : lpi2c_baud_rate_params_t_Class

Definition at line 161 of file lpi2c_driver.h.

Data Fields

- uint32_t baudRate

Field Documentation

16.49.2.3.1   uint32_t baudRate

Definition at line 163 of file lpi2c_driver.h.

16.49.2.4   struct lpi2c_master_state_t

Master internal context structure.

This structure is used by the master-mode driver for its internal logic. It must be provided by the application through the LPI2C_DRV_MasterInit() function, then it cannot be freed until the driver is de-initialized using LPI2C_DRV_↵ MasterDeinit(). The application should make no assumptions about the content of this structure.

Definition at line 203 of file lpi2c_driver.h.

16.49.2.5   struct lpi2c_slave_state_t

Slave internal context structure.

This structure is used by the slave-mode driver for its internal logic. It must be provided by the application through the LPI2C_DRV_SlaveInit() function, then it cannot be freed until the driver is de-initialized using LPI2C_DRV_↵ SlaveDeinit(). The application should make no assumptions about the content of this structure.

Definition at line 238 of file lpi2c_driver.h.

16.49.3   Enumeration Type Documentation

16.49.3.1   enum lpi2c_mode_t

I2C operating modes Implements : lpi2c_mode_t_Class.

Enumerator

> **_LPI2C_STANDARD_MODE_**   Standard-mode (Sm), bidirectional data transfers up to 100 kbit/s
>
> **_LPI2C_FAST_MODE_**   Fast-mode (Fm), bidirectional data transfers up to 400 kbit/s

Definition at line 74 of file lpi2c_driver.h.

16.49.3.2   enum lpi2c_transfer_type_t

Type of LPI2C transfer (based on interrupts or DMA). Implements : lpi2c_transfer_type_t_Class.

Enumerator

> **_LPI2C_USING_DMA_**   The driver will use DMA to perform I2C transfer
>
> **_LPI2C_USING_INTERRUPTS_**   The driver will use interrupts to perform I2C transfer

Definition at line 92 of file lpi2c_driver.h.

16.49.4   Function Documentation

16.49.4.1   status_t LPI2C_DRV_MasterAbortTransferData (uint32_t _instance_)

Abort a non-blocking I2C Master transmission or reception.

Parameters

instance	LPI2C peripheral instance number

Returns

> Error or success status returned by API

Definition at line 1477 of file lpi2c_driver.c.

16.49.4.2   status_t LPI2C_DRV_MasterDeinit (uint32_t _instance_)

De-initialize the LPI2C master mode driver.

This function de-initializes the LPI2C driver in master mode. The driver can't be used again until reinitialized. The context structure is no longer needed by the driver and can be freed after calling this function.

Parameters

instance	LPI2C peripheral instance number

Returns

> Error or success status returned by API

Definition at line 1098 of file lpi2c_driver.c.

16.49.4.3   void LPI2C_DRV_MasterGetBaudRate (uint32_t _instance,_ lpi2c_baud_rate_params_t ∗ _baudRate_)

Get the currently configured baud rate.

This function returns the currently configured baud rate.

Parameters

instance	LPI2C peripheral instance number
baudRate	structure that contains the current baud rate in hertz and the baud rate in hertz for High-speed mode (unused in other modes, can be NULL)

Definition at line 1131 of file lpi2c_driver.c.

16.49.4.4 status_t LPI2C_DRV_MasterGetTransferStatus (uint32_t *instance,* uint32_t ∗ *bytesRemaining*)

Return the current status of the I2C master transfer.

This function can be called during a non-blocking transmission to check the status of the transfer.

Parameters

instance	LPI2C peripheral instance number
bytesRemaining	the number of remaining bytes in the active I2C transfer

Returns

Error or success status returned by API

Definition at line 1646 of file lpi2c_driver.c.

16.49.4.5 status_t LPI2C_DRV_MasterInit (uint32_t *instance,* const lpi2c_master_user_config_t ∗ *userConfigPtr,* lpi2c_master_state_t ∗ *master*)

Initialize the LPI2C master mode driver.

This function initializes the LPI2C driver in master mode.

Parameters

instance	LPI2C peripheral instance number
userConfigPtr	Pointer to the LPI2C master user configuration structure. The function reads configuration data from this structure and initializes the driver accordingly. The application may free this structure after the function returns.
master	Pointer to the LPI2C master driver context structure. The driver uses this memory area for its internal logic. The application must make no assumptions about the content of this structure, and must not free this memory until the driver is de-initialized using LPI2C_DRV_Master↩ Deinit().

Returns

Error or success status returned by API

Definition at line 1016 of file lpi2c_driver.c.

16.49.4.6 void LPI2C_DRV_MasterIRQHandler (uint32_t *instance*)

Handle master operation when I2C interrupt occurs.

This is the interrupt service routine for the LPI2C master mode driver. It handles the rest of the transfer started by one of the send/receive functions.

Parameters

instance	LPI2C peripheral instance number

Definition at line 1688 of file lpi2c_driver.c.

16.49.4.7 status_t LPI2C_DRV_MasterReceiveData (uint32_t *instance,* uint8_t ∗ *rxBuff,* uint32_t *rxSize,* bool *sendStop*)

Perform a non-blocking receive transaction on the I2C bus.

This function starts the reception of a block of data from the currently configured slave address and returns immediately. The rest of the reception is handled by the interrupt service routine. Use LPI2C_DRV_MasterGetReceive↩
Status() to check the progress of the reception.

Parameters

instance	LPI2C peripheral instance number
rxBuff	pointer to the buffer where to store received data
rxSize	length in bytes of the data to be transferred
sendStop	specifies whether or not to generate stop condition after the reception

Returns

> Error or success status returned by API

Definition at line 1512 of file lpi2c_driver.c.

16.49.4.8   status_t LPI2C_DRV_MasterReceiveDataBlocking (uint32_t *instance,* uint8_t ∗ *rxBuff,* uint32_t *rxSize,* bool
        ***sendStop,* uint32_t *timeout*)**

Perform a blocking receive transaction on the I2C bus.

This function receives a block of data from the currently configured slave address, and only returns when the transmission is complete.

Parameters

instance	LPI2C peripheral instance number
rxBuff	pointer to the buffer where to store received data
rxSize	length in bytes of the data to be transferred
sendStop	specifies whether or not to generate stop condition after the reception
timeout	timeout for the transfer in milliseconds

Returns

> Error or success status returned by API

Definition at line 1605 of file lpi2c_driver.c.

16.49.4.9   status_t LPI2C_DRV_MasterSendData (uint32_t *instance,* const uint8_t ∗ *txBuff,* uint32_t *txSize,* bool *sendStop*)

Perform a non-blocking send transaction on the I2C bus.

This function starts the transmission of a block of data to the currently configured slave address and returns immediately. The rest of the transmission is handled by the interrupt service routine. Use LPI2C_DRV_MasterGetSend↩
Status() to check the progress of the transmission.

Parameters

instance	LPI2C peripheral instance number
txBuff	pointer to the data to be transferred
txSize	length in bytes of the data to be transferred
sendStop	specifies whether or not to generate stop condition after the transmission

Returns

> Error or success status returned by API

Definition at line 1362 of file lpi2c_driver.c.

16.49.4.10   status_t LPI2C_DRV_MasterSendDataBlocking (uint32_t *instance,* const uint8_t ∗ *txBuff,* uint32_t *txSize,* bool
         ***sendStop,* uint32_t *timeout*)**

Perform a blocking send transaction on the I2C bus.

This function sends a block of data to the currently configured slave address, and only returns when the transmission is complete.

Parameters

instance	LPI2C peripheral instance number
txBuff	pointer to the data to be transferred
txSize	length in bytes of the data to be transferred
sendStop	specifies whether or not to generate stop condition after the transmission
timeout	timeout for the transfer in milliseconds

Returns

> Error or success status returned by API

Definition at line 1440 of file lpi2c_driver.c.

16.49.4.11   void LPI2C_DRV_MasterSetBaudRate (uint32_t *instance,* const **lpi2c_mode_t *operatingMode,* const **lpi2c_baud_rate_params_t** *baudRate*)**

Set the baud rate for any subsequent I2C communication.

This function sets the baud rate (SCL frequency) for the I2C master. It can also change the operating mode. If the operating mode is High-Speed, a second baud rate must be provided for high-speed communication. Note that due to module limitation not any baud rate can be achieved. The driver will set a baud rate as close as possible to the requested baud rate, but there may still be substantial differences, for example if requesting a high baud rate while using a low-frequency protocol clock for the LPI2C module. The application should call LPI2C_DRV_MasterGet↩ BaudRate() after LPI2C_DRV_MasterSetBaudRate() to check what baud rate was actually set.

Parameters

instance	LPI2C peripheral instance number
operatingMode	I2C operating mode
baudRate	structure that contains the baud rate in hertz to use by current slave device and also the baud rate in hertz for High-speed mode (unused in other modes)

Definition at line 1183 of file lpi2c_driver.c.

16.49.4.12   void LPI2C_DRV_MasterSetSlaveAddr (uint32_t *instance,* const uint16_t *address,* const bool *is10bitAddr*)

Set the slave address for any subsequent I2C communication.

This function sets the slave address which will be used for any future transfer initiated by the LPI2C master.

Parameters

instance	LPI2C peripheral instance number
address	slave address, 7-bit or 10-bit
is10bitAddr	specifies if provided address is 10-bit

Definition at line 1341 of file lpi2c_driver.c.

16.49.4.13   status_t LPI2C_DRV_SlaveAbortTransferData (uint32_t *instance*)

Abort a non-blocking I2C Master transmission or reception.

Parameters

instance	LPI2C peripheral instance number

Returns

> Error or success status returned by API

Definition at line 2282 of file lpi2c_driver.c.

16.49.4.14   status_t LPI2C_DRV_SlaveDeinit (uint32_t *instance*)

De-initialize the I2C slave mode driver.

This function de-initializes the LPI2C driver in slave mode. The driver can't be used again until reinitialized. The context structure is no longer needed by the driver and can be freed after calling this function.

Parameters

instance	LPI2C peripheral instance number

Returns

> Error or success status returned by API

Definition at line 1923 of file lpi2c_driver.c.

16.49.4.15   status_t LPI2C_DRV_SlaveGetTransferStatus (uint32_t *instance,* uint32_t ∗ *bytesRemaining*)

Return the current status of the I2C slave transfer.

This function can be called during a non-blocking transmission to check the status of the transfer.

Parameters

instance	LPI2C peripheral instance number
bytesRemaining	the number of remaining bytes in the active I2C transfer

Returns

> Error or success status returned by API

Definition at line 2243 of file lpi2c_driver.c.

16.49.4.16   status_t LPI2C_DRV_SlaveInit (uint32_t *instance,* const lpi2c_slave_user_config_t ∗ *userConfigPtr,* lpi2c_slave_state_t ∗ *slave*)

Initialize the I2C slave mode driver.

Parameters

instance	LPI2C peripheral instance number
userConfigPtr	Pointer to the LPI2C slave user configuration structure. The function reads configuration data from this structure and initializes the driver accordingly. The application may free this structure after the function returns.
slave	Pointer to the LPI2C slave driver context structure. The driver uses this memory area for its internal logic. The application must make no assumptions about the content of this structure, and must not free this memory until the driver is de-initialized using LPI2C_DRV_Slave↩ Deinit().

Returns

> Error or success status returned by API

Definition at line 1804 of file lpi2c_driver.c.

16.49.4.17   void LPI2C_DRV_SlaveIRQHandler (uint32_t *instance*)

Handle slave operation when I2C interrupt occurs.

This is the interrupt service routine for the LPI2C slave mode driver. It handles any transfer initiated by an I2C master and notifies the application via the provided callback when relevant events occur.

Parameters

instance	LPI2C peripheral instance number

Definition at line 2311 of file lpi2c_driver.c.

16.49.4.18   status_t LPI2C_DRV_SlaveReceiveData (uint32_t *instance,* uint8_t ∗ *rxBuff,* uint32_t *rxSize*)

Perform a non-blocking receive transaction on the I2C bus.

Performs a non-blocking receive transaction on the I2C bus when the slave is not in listening mode (initialized with slaveListening = false). It starts the reception and returns immediately. The rest of the reception is handled by the interrupt service routine. Use LPI2C_DRV_SlaveGetReceiveStatus() to check the progress of the reception.

Parameters

instance	LPI2C peripheral instance number
rxBuff	pointer to the buffer where to store received data
rxSize	length in bytes of the data to be transferred

Returns

> Error or success status returned by API

Definition at line 2134 of file lpi2c_driver.c.

16.49.4.19   status_t LPI2C_DRV_SlaveReceiveDataBlocking (uint32_t *instance,* uint8_t ∗ *rxBuff,* uint32_t *rxSize,* uint32_t *timeout*)

Perform a blocking receive transaction on the I2C bus.

Performs a blocking receive transaction on the I2C bus when the slave is not in listening mode (initialized with slaveListening = false). It sets up the reception and then waits for the transfer to complete before returning.

Parameters

instance	LPI2C peripheral instance number
rxBuff	pointer to the buffer where to store received data
rxSize	length in bytes of the data to be transferred
timeout	timeout for the transfer in milliseconds

Returns

> Error or success status returned by API

Definition at line 2204 of file lpi2c_driver.c.

16.49.4.20   status_t LPI2C_DRV_SlaveSendData (uint32_t *instance,* const uint8_t ∗ *txBuff,* uint32_t *txSize*)

Perform a non-blocking send transaction on the I2C bus.

Performs a non-blocking send transaction on the I2C bus when the slave is not in listening mode (initialized with slaveListening = false). It starts the transmission and returns immediately. The rest of the transmission is handled by the interrupt service routine. Use LPI2C_DRV_SlaveGetTransmitStatus() to check the progress of the transmission.

Parameters

instance	LPI2C peripheral instance number
txBuff	pointer to the data to be transferred

txSize	length in bytes of the data to be transferred

Returns

> Error or success status returned by API

Definition at line 2016 of file lpi2c_driver.c.

16.49.4.21   status_t LPI2C_DRV_SlaveSendDataBlocking (uint32_t *instance,* const uint8_t ∗ *txBuff,* uint32_t *txSize,* uint32_t *timeout*)

Perform a blocking send transaction on the I2C bus.

Performs a blocking send transaction on the I2C bus when the slave is not in listening mode (initialized with slave←↩ Listening = false). It sets up the transmission and then waits for the transfer to complete before returning.

Parameters

instance	LPI2C peripheral instance number
txBuff	pointer to the data to be transferred
txSize	length in bytes of the data to be transferred
timeout	timeout for the transfer in milliseconds

Returns

> Error or success status returned by API

Definition at line 2099 of file lpi2c_driver.c.

16.49.4.22   status_t LPI2C_DRV_SlaveSetRxBuffer (uint32_t *instance,* uint8_t ∗ *rxBuff,* uint32_t *rxSize*)

Provide a buffer for receiving data.

This function provides a buffer in which the LPI2C slave-mode driver can store received data. It can be called for example from the user callback provided at initialization time, when the driver reports events LPI2C_SLAVE_EV←↩ ENT_RX_REQ or LPI2C_SLAVE_EVENT_RX_FULL.

Parameters

instance	LPI2C peripheral instance number
rxBuff	pointer to the data to be transferred
rxSize	length in bytes of the data to be transferred

Returns

> Error or success status returned by API

Definition at line 1989 of file lpi2c_driver.c.

16.49.4.23   status_t LPI2C_DRV_SlaveSetTxBuffer (uint32_t *instance,* const uint8_t ∗ *txBuff,* uint32_t *txSize*)

Provide a buffer for transmitting data.

This function provides a buffer from which the LPI2C slave-mode driver can transmit data. It can be called for example from the user callback provided at initialization time, when the driver reports events LPI2C_SLAVE_EV←↩ ENT_TX_REQ or LPI2C_SLAVE_EVENT_TX_EMPTY.

Parameters

instance	LPI2C peripheral instance number
txBuff	pointer to the data to be transferred
txSize	length in bytes of the data to be transferred

Returns

Error or success status returned by API

Definition at line 1962 of file lpi2c_driver.c.

16.50   LPIT Driver

16.50.1   Detailed Description

Low Power Interrupt Timer Peripheral Driver.

Hardware background

Each LPIT timer channel can be configured to run in one of 4 modes:
32-bit Periodic Counter: In this mode the counter will load and then decrement down to zero. It will then set the timer interrupt flag and assert the output pre-trigger.
Dual 16-bit Periodic Counter: In this mode, the counter will load and then the lower 16-bits will decrement down to zero, which will assert the output pre-trigger. The upper 16-bits will then decrement down to zero, which will negate the output pre-trigger and set the timer interrupt flag.
32-bit Trigger Accumulator: In this mode, the counter will load on the first trigger rising edge and then decrement down to zero on each trigger rising edge. It will then set the timer interrupt flag and assert the output pre-trigger.
32-bit Trigger Input Capture: In this mode, the counter will load with 0xFFFF_FFFF and then decrement down to zero. If a trigger rising edge is detected, it will store the inverse of the current counter value in the load value register, set the timer interrupt flag and assert the output pre-trigger.
In these modes, the timer channel operation is further controlled by Trigger Control bits (TSOT, TSOI, TROT) which control the load, reload, start and restart of the timer channels.

Driver consideration

The Driver uses structures for configuration. Each structure contains members that are specific to its respective functionality. There are **lpit_user_config_t** and **lpit_user_channel_config_t**.

Interrupt handling

Each LPIT timer channel has a corresponding interrupt handler. The LPIT Driver does not define interrupt handler internally. These interrupt handler methods can be defined by the user application. There are two ways to add an LPIT interrupt handler:

1. Using the weak symbols defined by start-up code. if the methods LPIT**x**_Handler(void) (x denotes instance number) are not defined, the linker use a default ISR. An error will be generated if methods with the same name are defined multiple times. This method works regardless of the placement of the interrupt vector table (Flash or RAM).

2. Using the Interrupt Manager's INT_SYS_InstallHandler() method. This can be used to dynamically change the ISR at run-time. This method works only if the interrupt vector table is located in RAM.

Clocking configuration

The LPIT Driver does not handle clock setup (from PCC) configuration. This is handled by the Clock Manager. The driver assumes that clock configurations have been made, so it is the user's responsibility to set up clocking and pin configurations correctly.

Basic operations

1. Pre-Initialization information of LPIT module

   - Before using the LPIT driver, the protocol clock of the module must be configured by the application using PCC module.

   - Configures Trigger MUX Control (TRGMUX) if want to use external trigger for LPIT module.

   - Configures different peripherals if want to use them in LPIT interrupt routine.

   - Provides configuration data structure to LPIT initialization API.

2. To initialize the LPIT module, just call the LPIT_DRV_Init() function with the user configuration data structure. This function configures LPIT module operation when MCU enters DEBUG and DOZE (Low power mode) modes and enables LPIT module. This function must be called firstly.

In the following code, LPIT module is initialized to continue to run when MCU enters both Debug and DOZE modes.

```
#define BOARD_LPIT_INSTANCE 0U
/* LPIT module configuration stucture */
lpit_user_config_t lpitconfig =
{
    .enableRunInDebug = true,
    .enableRunInDoze  = true
};
/* Initializes the LPIT module. */
LPIT_DRV_Init(BOARD_LPIT_INSTANCE, &lpitconfig);
```

3. After calling the LPIT_DRV_Init() function, call LPIT_DRV_InitChannel() function with user channel configuration structure to initialize timer channel.
   This function configures timer channel chaining, timer channel mode, timer channel period, interrupt generation, trigger source, trigger select, reload on trigger, stop on interrupt and start on trigger. In the following code, timer channel is initialized with the channel chaining is disabled, interrupt generation is enabled, operation mode is 32 bit periodic counter mode, trigger source is external, reload on trigger is disabled, stop on interrupt is disabled, start on trigger is disabled and timer period is 1 second. Note that:

   • Trigger select is not effective if trigger source is external.

   • Timer channel period must be suitable for operation mode.

   • The timer channel 0 can not be chained.
     ```
     /* Channel 0 configuration structure */
     lpit_user_channel_config_t chnlconfig =
     {
         .timerMode = LPIT_PERIODIC_COUNTER,
         .periodUnits = LPIT_PERIOD_UNITS_MICROSECONDS,
         .period = 1000000U,
         .triggerSource = LPIT_TRIGGER_SOURCE_INTERNAL,
         .triggerSelect = 1U,
         .enableReloadOnTrigger = false,
         .enableStopOnInterrupt = false,
         .enableStartOnTrigger = false,
         .chainChannel = false,
         .isInterruptEnabled = true
     };
     /* Initializes the channel 0 */
     LPIT_DRV_InitChannel(BOARD_LPIT_INSTANCE, 0, &chnlconfig);
     ```

4. To reconfigure timer channel period , just call LPIT_DRV_SetTimerPeriodByUs() or LPIT_DRV_SetTimer↩ PeriodByCount() with corresponding new period. In the following code, the timer channel period is reconfigured with new period in count unit.

```
/* Reconfigures timer channel period with new period of 10000 count*/
LPIT_DRV_SetTimerPeriodByCount(BOARD_LPIT_INSTANCE, 0, 10000);
```

5. To start timer channel counting, just call LPIT_DRV_StartTimerChannels() with timer channels starting mask. In the following code, the timer channel 0 is started with the mask of 0x1U.

```
/* Starts channel 0 counting*/
LPIT_DRV_StartTimerChannels(BOARD_LPIT_INSTANCE, 0x1U);
```

6. To stop timer channel counting, just call LPIT_DRV_StopTimerChannels() with timer channels stopping mask. In the following code, the timer channel 0 is stopped with the mask of 0x1U.

```
/* Stops channel 0 counting*/
LPIT_DRV_StopTimerChannels(BOARD_LPIT_INSTANCE, 0x1U);
```

7. To disable LPIT module, just call LPIT_DRV_Deinit().

```
/* Disables LPIT module*/
LPIT_DRV_Deinit(BOARD_LPIT_INSTANCE);
```

API

Some of the features exposed by the API are targeted specifically for timer channel mode. For example, set/get timer period in dual 16 mode function makes sense if timer channel mode is dual 16 mode, so therefor it is restricted for use in other modes.

For any invalid configuration the functions will either return an error code or trigger DEV_ASSERT (if enabled). For more details, please refer to each function description.

Data Structures

- struct lpit_user_config_t

  LPIT configuration structure. More...
- struct lpit_user_channel_config_t

  Structure to configure the channel timer. More...

Macros

- #define MAX_PERIOD_COUNT (0xFFFFFFFFU)

  Max period in count of all operation mode except for dual 16 bit periodic counter mode.
- #define MAX_PERIOD_COUNT_IN_DUAL_16BIT_MODE (0x1FFFEU)

  Max period in count of dual 16 bit periodic counter mode.
- #define MAX_PERIOD_COUNT_16_BIT (0xFFFFU)

  Max count of 16 bit.

Enumerations

- enum lpit_timer_modes_t { LPIT_PERIODIC_COUNTER = 0x00U, LPIT_DUAL_PERIODIC_COUNTER = 0x01U, LPIT_TRIGGER_ACCUMULATOR = 0x02U, LPIT_INPUT_CAPTURE = 0x03U }

  Mode options available for the LPIT timer Implements : lpit_timer_modes_t_Class.
- enum lpit_trigger_source_t { LPIT_TRIGGER_SOURCE_EXTERNAL = 0x00U, LPIT_TRIGGER_SOURCE_INTERNAL = 0x01U }

  Trigger source options.
- enum lpit_period_units_t { LPIT_PERIOD_UNITS_COUNTS = 0x00U, LPIT_PERIOD_UNITS_MICROSECONDS = 0x01U }

  Unit options for LPIT period.

Initialization and De-initialization

- void LPIT_DRV_Init (uint32_t instance, const lpit_user_config_t ∗userConfig)

  Initializes the LPIT module.
- void LPIT_DRV_Deinit (uint32_t instance)

  De-Initializes the LPIT module.
- status_t LPIT_DRV_InitChannel (uint32_t instance, uint32_t channel, const lpit_user_channel_config_t ∗userChannelConfig)

  Initializes the LPIT channel.

Timer Start and Stop

- void LPIT_DRV_StartTimerChannels (uint32_t instance, uint32_t mask)

  Starts the timer channel counting.
- void LPIT_DRV_StopTimerChannels (uint32_t instance, uint32_t mask)

  Stops the timer channel counting.

Timer Period

- status_t LPIT_DRV_SetTimerPeriodByUs (uint32_t instance, uint32_t channel, uint32_t periodUs)

    Sets the timer channel period in microseconds.
- status_t LPIT_DRV_SetTimerPeriodInDual16ModeByUs (uint32_t instance, uint32_t channel, uint16_↩
  t periodHigh, uint16_t periodLow)

    Sets the timer channel period in microseconds.
- uint64_t LPIT_DRV_GetTimerPeriodByUs (uint32_t instance, uint32_t channel)

    Gets the timer channel period in microseconds.
- uint64_t LPIT_DRV_GetCurrentTimerUs (uint32_t instance, uint32_t channel)

    Gets the current timer channel counting value in microseconds.
- void LPIT_DRV_SetTimerPeriodByCount (uint32_t instance, uint32_t channel, uint32_t count)

    Sets the timer channel period in count unit.
- void LPIT_DRV_SetTimerPeriodInDual16ModeByCount (uint32_t instance, uint32_t channel, uint16_↩
  t periodHigh, uint16_t periodLow)

    Sets the timer channel period in count unit.
- uint32_t LPIT_DRV_GetTimerPeriodByCount (uint32_t instance, uint32_t channel)

    Gets the current timer channel period in count unit.
- uint32_t LPIT_DRV_GetCurrentTimerCount (uint32_t instance, uint32_t channel)

    Gets the current timer channel counting value in count.

Interrupt

- uint32_t LPIT_DRV_GetInterruptFlagTimerChannels (uint32_t instance, uint32_t mask)

    Gets the current interrupt flag of timer channels.
- void LPIT_DRV_ClearInterruptFlagTimerChannels (uint32_t instance, uint32_t mask)

    Clears the interrupt flag of timer channels.

16.50.2 Data Structure Documentation

16.50.2.1 struct lpit_user_config_t

LPIT configuration structure.

This structure holds the configuration settings for the LPIT peripheral to enable or disable LPIT module in DEBUG and DOZE mode Implements : lpit_user_config_t_Class

Definition at line 107 of file lpit_driver.h.

Data Fields

- bool enableRunInDebug
- bool enableRunInDoze

Field Documentation

16.50.2.1.1 bool enableRunInDebug

True: Timer channels continue to run in debug mode False: Timer channels stop in debug mode

Definition at line 109 of file lpit_driver.h.

16.50.2.1.2 bool enableRunInDoze

True: Timer channels continue to run in doze mode False: Timer channels stop in doze mode

Definition at line 111 of file lpit_driver.h.

16.50.2.2   struct lpit_user_channel_config_t

Structure to configure the channel timer.

This structure holds the configuration settings for the LPIT timer channel Implements : lpit_user_channel_config↩
_t_Class

Definition at line 120 of file lpit_driver.h.

Data Fields

- lpit_timer_modes_t timerMode
- lpit_period_units_t periodUnits
- uint32_t period
- lpit_trigger_source_t triggerSource
- uint32_t triggerSelect
- bool enableReloadOnTrigger
- bool enableStopOnInterrupt
- bool enableStartOnTrigger
- bool chainChannel
- bool isInterruptEnabled

Field Documentation

16.50.2.2.1   bool chainChannel

Channel chaining enable

Definition at line 136 of file lpit_driver.h.

16.50.2.2.2   bool enableReloadOnTrigger

True: Timer channel will reload on selected trigger False: Timer channel will not reload on selected trigger

Definition at line 128 of file lpit_driver.h.

16.50.2.2.3   bool enableStartOnTrigger

True: Timer channel starts to decrement when rising edge on selected trigger is detected. False: Timer starts to
decrement immediately based on restart condition

Definition at line 132 of file lpit_driver.h.

16.50.2.2.4   bool enableStopOnInterrupt

True: Timer will stop after timeout False: Timer channel does not stop after timeout

Definition at line 130 of file lpit_driver.h.

16.50.2.2.5   bool isInterruptEnabled

Timer channel interrupt generation enable

Definition at line 137 of file lpit_driver.h.

16.50.2.2.6   uint32_t period

Period of timer channel

Definition at line 124 of file lpit_driver.h.

16.50.2.2.7   lpit_period_units_t periodUnits

Timer period value units

Definition at line 123 of file lpit_driver.h.

16.50.2.2.8 lpit_timer_modes_t timerMode

Operation mode of timer channel

Definition at line 122 of file lpit_driver.h.

16.50.2.2.9 uint32_t triggerSelect

Selects one trigger from the internal trigger sources this field makes sense if trigger source is internal

Definition at line 126 of file lpit_driver.h.

16.50.2.2.10 lpit_trigger_source_t triggerSource

Selects between internal and external trigger sources

Definition at line 125 of file lpit_driver.h.

16.50.3 Macro Definition Documentation

16.50.3.1 #define MAX_PERIOD_COUNT (0xFFFFFFFFU)

Max period in count of all operation mode except for dual 16 bit periodic counter mode.

Definition at line 57 of file lpit_driver.h.

16.50.3.2 #define MAX_PERIOD_COUNT_16_BIT (0xFFFFU)

Max count of 16 bit.

Definition at line 61 of file lpit_driver.h.

16.50.3.3 #define MAX_PERIOD_COUNT_IN_DUAL_16BIT_MODE (0x1FFFEU)

Max period in count of dual 16 bit periodic counter mode.

Definition at line 59 of file lpit_driver.h.

16.50.4 Enumeration Type Documentation

16.50.4.1 enum lpit_period_units_t

Unit options for LPIT period.

This is used to determine unit of timer period Implements : lpit_period_units_t_Class

Enumerator

> **LPIT_PERIOD_UNITS_COUNTS** Period value unit is count
> **LPIT_PERIOD_UNITS_MICROSECONDS** Period value unit is microsecond

Definition at line 94 of file lpit_driver.h.

16.50.4.2 enum lpit_timer_modes_t

Mode options available for the LPIT timer Implements : lpit_timer_modes_t_Class.

Enumerator

> **LPIT_PERIODIC_COUNTER** 32-bit Periodic Counter
> **LPIT_DUAL_PERIODIC_COUNTER** Dual 16-bit Periodic Counter
> **LPIT_TRIGGER_ACCUMULATOR** 32-bit Trigger Accumulator

LPIT_INPUT_CAPTURE   32-bit Trigger Input Capture

Definition at line 67 of file lpit_driver.h.

16.50.4.3   enum **lpit_trigger_source_t**

Trigger source options.

This is used for both internal and external trigger sources. The actual trigger options available is SoC specific, user should refer to the reference manual. Implements : lpit_trigger_source_t_Class

Enumerator

LPIT_TRIGGER_SOURCE_EXTERNAL   Use external trigger

LPIT_TRIGGER_SOURCE_INTERNAL   Use internal trigger

Definition at line 82 of file lpit_driver.h.

16.50.5   Function Documentation

16.50.5.1   void LPIT_DRV_ClearInterruptFlagTimerChannels (uint32_t *instance,* uint32_t *mask*)

Clears the interrupt flag of timer channels.

This function clears the interrupt flag of timer channels after their interrupt event occurred.

Parameters

in	*instance*	LPIT module instance number
in	*mask*	The interrupt flag clearing mask that decides which channels will be cleared interrupt flag • For example: – with mask = 0x01u then the interrupt flag of channel 0 only will be cleared – with mask = 0x02u then the interrupt flag of channel 1 only will be cleared – with mask = 0x03u then the interrupt flags of channel 0 and channel 1 will be cleared

Definition at line 646 of file lpit_driver.c.

16.50.5.2   void LPIT_DRV_Deinit (uint32_t *instance*)

De-Initializes the LPIT module.

This function disables LPIT module. In order to use the LPIT module again, LPIT_DRV_Init must be called.

Parameters

in	*instance*	LPIT module instance number

Definition at line 125 of file lpit_driver.c.

16.50.5.3   uint32_t LPIT_DRV_GetCurrentTimerCount (uint32_t *instance,* uint32_t *channel*)

Gets the current timer channel counting value in count.

This function returns the real-time timer channel counting value, the value in a range from 0 to timer channel period. Need to make sure the running time does not exceed the timer channel period.

Parameters

in	*instance*	LPIT module instance number
in	*channel*	Timer channel number

Returns

> Current timer channel counting value in count

Definition at line 592 of file lpit_driver.c.

16.50.5.4 uint64_t LPIT_DRV_GetCurrentTimerUs (uint32_t *instance,* uint32_t *channel*)

Gets the current timer channel counting value in microseconds.

This function returns an absolute time stamp in microseconds. One common use of this function is to measure the running time of a part of code. Call this function at both the beginning and end of code. The time difference between these two time stamps is the running time. The return counting value here makes sense if the operation mode of timer channel is 32 bit periodic counter or dual 16 bit periodic counter or 32-bit trigger input capture. Need to make sure the running time will not exceed the timer channel period.

Parameters

in	*instance*	LPIT module instance number
in	*channel*	Timer channel number

Returns

> Current timer channel counting value in microseconds

Definition at line 457 of file lpit_driver.c.

16.50.5.5 uint32_t LPIT_DRV_GetInterruptFlagTimerChannels (uint32_t *instance,* uint32_t *mask*)

Gets the current interrupt flag of timer channels.

This function gets the current interrupt flag of timer channels. In compare modes, the flag sets to 1 at the end of the timer period. In capture modes, the flag sets to 1 when the trigger asserts.

Parameters

in	*instance*	LPIT module instance number.

in	*mask*	The interrupt flag getting mask that decides which channels will be got interrupt flag.
		
		 • For example:
		
		   – with mask = 0x01u then the interrupt flag of channel 0 only will be got
		
		   – with mask = 0x02u then the interrupt flag of channel 1 only will be got
		
		   – with mask = 0x03u then the interrupt flags of channel 0 and channel 1 will be got

Returns

> Current the interrupt flag of timer channels

Definition at line 625 of file lpit_driver.c.

16.50.5.6   uint32_t LPIT_DRV_GetTimerPeriodByCount (uint32_t *instance,* uint32_t *channel*)

Gets the current timer channel period in count unit.

This function returns current period of timer channel given as argument.

Parameters

in	*instance*	LPIT module instance number
in	*channel*	Timer channel number

Returns

    Timer channel period in count unit

Definition at line 558 of file lpit_driver.c.

16.50.5.7 uint64_t LPIT_DRV_GetTimerPeriodByUs (uint32_t *instance,* uint32_t *channel*)

Gets the timer channel period in microseconds.

This function gets the timer channel period in microseconds. The returned period here makes sense if the operation mode of timer channel is 32 bit periodic counter or dual 16 bit periodic counter.

Parameters

in	*instance*	LPIT module instance number
in	*channel*	Timer channel number

Returns

    Timer channel period in microseconds

Definition at line 398 of file lpit_driver.c.

16.50.5.8 void LPIT_DRV_Init (uint32_t *instance,* const lpit_user_config_t ∗ *userConfig*)

Initializes the LPIT module.

This function resets LPIT module, enables the LPIT module, configures LPIT module operation in Debug and DOZE mode. The LPIT configuration structure shall be passed as arguments. This configuration structure affects all timer channels. This function should be called before calling any other LPIT driver function.

This is an example demonstrating how to define a LPIT configuration structure:

```
1 lpit_user_config_t lpitInit =
2 {
3     .enableRunInDebug = false,
4     .enableRunInDoze = true
5 };
```

Parameters

in	*instance*	LPIT module instance number.
in	*userConfig*	Pointer to LPIT configuration structure.

Definition at line 90 of file lpit_driver.c.

16.50.5.9 status_t LPIT_DRV_InitChannel (uint32_t *instance,* uint32_t *channel,* const lpit_user_channel_config_t ∗ *userChannelConfig*)

Initializes the LPIT channel.

This function initializes the LPIT timers by using a channel, this function configures timer channel chaining, timer channel mode, timer channel period, interrupt generation, trigger source, trigger select, reload on trigger, stop on interrupt and start on trigger. The timer channel number and its configuration structure shall be passed as arguments. Timer channels do not start counting by default after calling this function. The function LPIT_DRV_↩
StartTimerChannels must be called to start the timer channel counting. In order to re-configures the period, call the LPIT_DRV_SetTimerPeriodByUs or LPIT_DRV_SetTimerPeriodByCount.

This is an example demonstrating how to define a LPIT channel configuration structure:

```
1 lpit_user_channel_config_t lpitTestInit =
2 {
3  .timerMode = LPIT_PERIODIC_COUNTER,
4  .periodUnits = LPTT_PERIOD_UNITS_MICROSECONDS,
5  .period = 1000000U,
6  .triggerSource = LPIT_TRIGGER_SOURCE_INTERNAL,
7  .triggerSelect = 1U,
8  .enableReloadOnTrigger = false,
9  .enableStopOnInterrupt = false,
10  .enableStartOnTrigger = false,
11  .chainChannel = false,
12  .isInterruptEnabled = true
13 };
```

Parameters

in	*instance*	LPIT module instance number
in	*channel*	Timer channel number
in	*userChannel↩ Config*	Pointer to LPIT channel configuration structure

Returns

Operation status

- STATUS_SUCCESS: Operation was successful.

- STATUS_ERROR: The channel 0 is chained.

- STATUS_ERROR: The input period is invalid.

Definition at line 152 of file lpit_driver.c.

16.50.5.10 void LPIT_DRV_SetTimerPeriodByCount (uint32_t *instance,* uint32_t *channel,* uint32_t *count*)

Sets the timer channel period in count unit.

This function sets the timer channel period in count unit. The counter period of a running timer channel can be modified by first setting a new load value, the value will be loaded after the timer channel expires. To abort the current cycle and start a timer channel period with the new value, the timer channel must be disabled and enabled again.

Parameters

in	*instance*	LPIT module instance number
in	*channel*	Timer channel number
in	*count*	Timer channel period in count unit

Definition at line 504 of file lpit_driver.c.

16.50.5.11 status_t LPIT_DRV_SetTimerPeriodByUs (uint32_t *instance,* uint32_t *channel,* uint32_t *periodUs*)

Sets the timer channel period in microseconds.

This function sets the timer channel period in microseconds when timer channel mode is 32 bit periodic or dual 16 bit counter mode. The period range depends on the frequency of the LPIT functional clock and operation mode of timer channel. If the required period is out of range, use the suitable mode if applicable. This function is only valid for one single channel.

Parameters

in	*instance*	LPIT module instance number
in	*channel*	Timer channel number

in	*periodUs*	Timer channel period in microseconds

Returns

> Operation status
>
> - STATUS_SUCCESS: Input period of timer channel is valid.
> - STATUS_ERROR: Input period of timer channel is invalid.

Definition at line 274 of file lpit_driver.c.

16.50.5.12   void LPIT_DRV_SetTimerPeriodInDual16ModeByCount (uint32_t *instance,* uint32_t *channel,* uint16_t *periodHigh,* uint16_t *periodLow*)

Sets the timer channel period in count unit.

This function sets the timer channel period in count unit when timer channel mode is dual 16 periodic counter mode. The counter period of a running timer channel can be modified by first setting a new load value, the value will be loaded after the timer channel expires. To abort the current cycle and start a timer channel period with the new value, the timer channel must be disabled and enabled again.

Parameters

in	*instance*	LPIT module instance number
in	*channel*	Timer channel number
in	*periodHigh*	Period of higher 16 bit in count unit
in	*periodLow*	Period of lower 16 bit in count unit

Definition at line 532 of file lpit_driver.c.

16.50.5.13   status_t LPIT_DRV_SetTimerPeriodInDual16ModeByUs (uint32_t *instance,* uint32_t *channel,* uint16_t *periodHigh,* uint16_t *periodLow*)

Sets the timer channel period in microseconds.

This function sets the timer channel period in microseconds when timer channel mode is dual 16 bit periodic counter mode. The period range depends on the frequency of the LPIT functional clock and operation mode of timer channel. If the required period is out of range, use the suitable mode if applicable. This function is only valid for one single channel.

Parameters

in	*instance*	LPIT module instance number
in	*channel*	Timer channel number
in	*periodHigh*	Period of higher 16 bit in microseconds
in	*periodLow*	Period of lower 16 bit in microseconds

Returns

> Operation status
>
> - STATUS_SUCCESS: Input period of timer channel is valid.
> - STATUS_ERROR: Input period of timer channel is invalid.

Definition at line 345 of file lpit_driver.c.

16.50.5.14   void LPIT_DRV_StartTimerChannels (uint32_t *instance,* uint32_t *mask*)

Starts the timer channel counting.

This function allows starting timer channels simultaneously . After calling this function, timer channels are going operate depend on mode and control bits which controls timer channel start, reload and restart.

Parameters

in	*instance*	LPIT module instance number

in	*mask*	Timer channels starting mask that decides which channels will be started
		
		• For example:
		
		    – with mask = 0x01U then channel 0 will be started
		
		    – with mask = 0x02U then channel 1 will be started
		
		    – with mask = 0x03U then channel 0 and channel 1 will be started

Definition at line 224 of file lpit_driver.c.

16.50.5.15   void LPIT_DRV_StopTimerChannels (uint32_t *instance,* uint32_t *mask*)

Stops the timer channel counting.

This function allows stop timer channels simultaneously from counting. Timer channels reload their periods respectively after the next time they call the LPIT_DRV_StartTimerChannels. Note that: In 32-bit Trigger Accumulator mode, the counter will load on the first trigger rising edge.

Parameters

in	*instance*	LPIT module instance number

in	*mask*	Timer channels stopping mask that decides which channels will be stopped
		
		• For example:
		
		    – with mask = 0x01U then channel 0 will be stopped
		
		    – with mask = 0x02U then channel 1 will be stopped
		
		    – with mask = 0x03U then channel 0 and channel 1 will be stopped

Definition at line 248 of file lpit_driver.c.

16.51 LPSPI Driver

16.51.1 Detailed Description

Low Power Serial Peripheral Interface Peripheral Driver.

Data Structures

- struct lpspi_master_config_t

    Data structure containing information about a device on the SPI bus. More...

- struct lpspi_state_t

    Runtime state structure for the LPSPI master driver. More...

- struct lpspi_slave_config_t

    User configuration structure for the SPI slave driver. Implements : lpspi_slave_config_t_Class. More...

Enumerations

- enum lpspi_which_pcs_t { LPSPI_PCS0 = 0U, LPSPI_PCS1 = 1U, LPSPI_PCS2 = 2U, LPSPI_PCS3 = 3U }

    LPSPI Peripheral Chip Select (PCS) configuration (which PCS to configure). Implements : lpspi_which_pcs_t_Class.

- enum lpspi_signal_polarity_t { LPSPI_ACTIVE_HIGH = 1U, LPSPI_ACTIVE_LOW = 0U }

    LPSPI Signal (PCS and Host Request) Polarity configuration. Implements : lpspi_signal_polarity_t_Class.

- enum lpspi_clock_phase_t { LPSPI_CLOCK_PHASE_1ST_EDGE = 0U, LPSPI_CLOCK_PHASE_2ND_E↩DGE = 1U }

    LPSPI clock phase configuration. Implements : lpspi_clock_phase_t_Class.

- enum lpspi_sck_polarity_t { LPSPI_SCK_ACTIVE_HIGH = 0U, LPSPI_SCK_ACTIVE_LOW = 1U }

    LPSPI Clock Signal (SCK) Polarity configuration. Implements : lpspi_sck_polarity_t_Class.

- enum lpspi_transfer_type { LPSPI_USING_DMA = 0, LPSPI_USING_INTERRUPTS }

    Type of LPSPI transfer (based on interrupts or DMA). Implements : lpspi_transfer_type_Class.

- enum transfer_status_t { LPSPI_TRANSFER_OK = 0U, LPSPI_TRANSMIT_FAIL, LPSPI_RECEIVE_FAIL }

    Type of error reported by LPSPI.

Functions

- void LPSPI_DRV_IRQHandler (uint32_t instance)

    The function LPSPI_DRV_IRQHandler passes IRQ control to either the master or slave driver.

- void LPSPI_DRV_FillupTxBuffer (uint32_t instance)

    The function LPSPI_DRV_FillupTxBuffer writes data in TX hardware buffer depending on driver state and number of bytes remained to send.

- void LPSPI_DRV_ReadRXBuffer (uint32_t instance)

    The function LPSPI_DRV_ReadRXBuffer reads data from RX hardware buffer and writes this data in RX software buffer.

- void LPSPI_DRV_DisableTEIEInterrupts (uint32_t instance)

    Disable the TEIE interrupts at the end of a transfer. Disable the interrupts and clear the status for transmit/receive errors.

- void LPSPI0_IRQHandler (void)

    This function is the implementation of LPSPI0 handler named in startup code.

- void LPSPI1_IRQHandler (void)

    This function is the implementation of LPSPI1 handler named in startup code.

- void LPSPI2_IRQHandler (void)

    This function is the implementation of LPSPI2 handler named in startup code.

Variables

- LPSPI_Type * g_lpspiBase [LPSPI_INSTANCE_COUNT]

  Table of base pointers for SPI instances.

- IRQn_Type g_lpspiIrqId [LPSPI_INSTANCE_COUNT]

  Table to save LPSPI IRQ enumeration numbers defined in the CMSIS header file.

- lpspi_state_t * g_lpspiStatePtr [LPSPI_INSTANCE_COUNT]

Initialization and shutdown

- status_t LPSPI_DRV_MasterInit (uint32_t instance, lpspi_state_t ∗lpspiState, const lpspi_master_config_t ∗spiConfig)

  Initializes a LPSPI instance for interrupt driven master mode operation.

- status_t LPSPI_DRV_MasterDeinit (uint32_t instance)

  Shuts down a LPSPI instance.

- status_t LPSPI_DRV_MasterSetDelay (uint32_t instance, uint32_t delayBetwenTransfers, uint32_t delayS←
  CKtoPCS, uint32_t delayPCStoSCK)

  Configures the LPSPI master mode bus timing delay options.

Bus configuration

- status_t LPSPI_DRV_MasterConfigureBus (uint32_t instance, const lpspi_master_config_t ∗spiConfig, uint32_t ∗calculatedBaudRate)

  Configures the LPSPI port physical parameters to access a device on the bus when the LSPI instance is configured for interrupt operation.

- status_t LPSPI_DRV_SetPcs (uint32_t instance, lpspi_which_pcs_t whichPcs, lpspi_signal_polarity_t polari-
  ty)

  Select the chip to communicate with.

Blocking transfers

- status_t LPSPI_DRV_MasterTransferBlocking (uint32_t instance, const uint8_t ∗sendBuffer, uint8_←
  t ∗receiveBuffer, uint16_t transferByteCount, uint32_t timeout)

  Performs an interrupt driven blocking SPI master mode transfer.

Non-blocking transfers

- status_t LPSPI_DRV_MasterTransfer (uint32_t instance, const uint8_t ∗ sendBuffer, uint8_t ∗receiveBuffer, uint16_t transferByteCount)

  Performs an interrupt driven non-blocking SPI master mode transfer.

- status_t LPSPI_DRV_MasterGetTransferStatus (uint32_t instance, uint32_t ∗bytesRemained)

  Returns whether the previous interrupt driven transfer is completed.

- status_t LPSPI_DRV_MasterAbortTransfer (uint32_t instance)

  Terminates an interrupt driven asynchronous transfer early.

- void LPSPI_DRV_MasterIRQHandler (uint32_t instance)

  Interrupt handler for LPSPI master mode. This handler uses the buffers stored in the lpspi_master_state_t structs to transfer data.

Initialization and shutdown

- status_t LPSPI_DRV_SlaveInit (uint32_t instance, lpspi_state_t *lpspiState, const lpspi_slave_config_↩
t *slaveConfig)

    Initializes a LPSPI instance for a slave mode operation, using interrupt mechanism.

- status_t LPSPI_DRV_SlaveDeinit (uint32_t instance)

    Shuts down an LPSPI instance interrupt mechanism.

Blocking transfers

- status_t LPSPI_DRV_SlaveTransferBlocking (uint32_t instance, const uint8_t *sendBuffer, uint8_t *receive↩
Buffer, uint16_t transferByteCount, uint32_t timeout)

    Transfers data on LPSPI bus using interrupt and a blocking call.

Non-blocking transfers

- void LPSPI_DRV_SlaveIRQHandler (uint32_t instance)

    Interrupt handler for LPSPI slave mode. This handler uses the buffers stored in the lpspi_master_state_t structs to transfer data.

- status_t LPSPI_DRV_SlaveTransfer (uint32_t instance, const uint8_t *sendBuffer, uint8_t *receiveBuffer, uint16_t transferByteCount)

    Starts the transfer data on LPSPI bus using an interrupt and a non-blocking call.

- status_t LPSPI_DRV_SlaveAbortTransfer (uint32_t instance)

    Aborts the transfer that started by a non-blocking call transfer function.

- status_t LPSPI_DRV_SlaveGetTransferStatus (uint32_t instance, uint32_t *bytesRemained)

    Returns whether the previous transfer is finished.

16.51.2   Data Structure Documentation

16.51.2.1   struct lpspi_master_config_t

Data structure containing information about a device on the SPI bus.

The user must populate these members to set up the LPSPI master and properly communicate with the SPI device. Implements : lpspi_master_config_t_Class

Definition at line 52 of file lpspi_master_driver.h.

Data Fields

- uint32_t bitsPerSec
- lpspi_which_pcs_t whichPcs
- lpspi_signal_polarity_t pcsPolarity
- bool isPcsContinuous
- uint16_t bitcount
- uint32_t lpspiSrcClk
- lpspi_clock_phase_t clkPhase
- lpspi_sck_polarity_t clkPolarity
- bool lsbFirst
- lpspi_transfer_type transferType
- uint8_t rxDMAChannel
- uint8_t txDMAChannel
- spi_callback_t callback
- void * callbackParam

Field Documentation

16.51.2.1.1   uint16_t bitcount

Number of bits/frame, minimum is 8-bits

Definition at line 58 of file lpspi_master_driver.h.

16.51.2.1.2   uint32_t bitsPerSec

Baud rate in bits per second

Definition at line 54 of file lpspi_master_driver.h.

16.51.2.1.3   spi_callback_t callback

Select the callback to transfer complete

Definition at line 66 of file lpspi_master_driver.h.

16.51.2.1.4   void∗ callbackParam

Select additional callback parameters if it's necessary

Definition at line 67 of file lpspi_master_driver.h.

16.51.2.1.5   lpspi_clock_phase_t clkPhase

Selects which phase of clock to capture data

Definition at line 60 of file lpspi_master_driver.h.

16.51.2.1.6   lpspi_sck_polarity_t clkPolarity

Selects clock polarity

Definition at line 61 of file lpspi_master_driver.h.

16.51.2.1.7   bool isPcsContinuous

Keeps PCS asserted until transfer complete

Definition at line 57 of file lpspi_master_driver.h.

16.51.2.1.8   uint32_t lpspiSrcClk

Module source clock

Definition at line 59 of file lpspi_master_driver.h.

16.51.2.1.9   bool lsbFirst

Option to transmit LSB first

Definition at line 62 of file lpspi_master_driver.h.

16.51.2.1.10   lpspi_signal_polarity_t pcsPolarity

PCS polarity

Definition at line 56 of file lpspi_master_driver.h.

16.51.2.1.11   uint8_t rxDMAChannel

Channel number for DMA rx channel. If DMA mode isn't used this field will be ignored.

Definition at line 64 of file lpspi_master_driver.h.

16.51.2.1.12 lpspi_transfer_type transferType

Type of LPSPI transfer

Definition at line 63 of file lpspi_master_driver.h.

16.51.2.1.13 uint8_t txDMAChannel

Channel number for DMA tx channel. If DMA mode isn't used this field will be ignored.

Definition at line 65 of file lpspi_master_driver.h.

16.51.2.1.14 lpspi_which_pcs_t whichPcs

Selects which PCS to use

Definition at line 55 of file lpspi_master_driver.h.

16.51.2.2 struct lpspi_state_t

Runtime state structure for the LPSPI master driver.

This structure holds data that is used by the LPSPI peripheral driver to communicate between the transfer function and the interrupt handler. The interrupt handler also uses this information to keep track of its progress. The user must pass the memory for this run-time state structure. The LPSPI master driver populates the members. Implements : lpspi_state_t_Class

Definition at line 127 of file lpspi_shared_function.h.

Data Fields

- uint16_t bitsPerFrame
- uint16_t bytesPerFrame
- bool isPcsContinuous
- bool isBlocking
- uint32_t lpspiSrcClk
- volatile bool isTransferInProgress
- const uint8_t ∗ txBuff
- uint8_t ∗ rxBuff
- volatile uint16_t txCount
- volatile uint16_t rxCount
- volatile uint16_t txFrameCnt
- volatile uint16_t rxFrameCnt
- volatile bool lsb
- uint8_t fifoSize
- uint8_t rxDMAChannel
- uint8_t txDMAChannel
- lpspi_transfer_type transferType
- semaphore_t lpspiSemaphore
- transfer_status_t status
- spi_callback_t callback
- void ∗ callbackParam
- uint32_t dummy

Field Documentation

16.51.2.2.1 uint16_t bitsPerFrame

Number of bits per frame: 8- to 4096-bits; needed for TCR programming

Definition at line 129 of file lpspi_shared_function.h.

16.51.2.2.2   uint16_t bytesPerFrame

Number of bytes per frame: 1- to 512-bytes

Definition at line 131 of file lpspi_shared_function.h.

16.51.2.2.3   spi_callback_t callback

Select the callback to transfer complete

Definition at line 150 of file lpspi_shared_function.h.

16.51.2.2.4   void∗ callbackParam

Select additional callback parameters if it's necessary

Definition at line 151 of file lpspi_shared_function.h.

16.51.2.2.5   uint32_t dummy

This field is used for the cases when TX is NULL and LPSPI is in DMA mode

Definition at line 152 of file lpspi_shared_function.h.

16.51.2.2.6   uint8_t fifoSize

RX/TX fifo size

Definition at line 144 of file lpspi_shared_function.h.

16.51.2.2.7   bool isBlocking

Save the transfer type

Definition at line 134 of file lpspi_shared_function.h.

16.51.2.2.8   bool isPcsContinuous

Option to keep chip select asserted until transfer complete; needed for TCR programming

Definition at line 132 of file lpspi_shared_function.h.

16.51.2.2.9   volatile bool isTransferInProgress

True if there is an active transfer

Definition at line 136 of file lpspi_shared_function.h.

16.51.2.2.10   semaphore_t lpspiSemaphore

The semaphore used for blocking transfers

Definition at line 148 of file lpspi_shared_function.h.

16.51.2.2.11   uint32_t lpspiSrcClk

Module source clock

Definition at line 135 of file lpspi_shared_function.h.

16.51.2.2.12   volatile bool lsb

True if first bit is LSB and false if first bit is MSB

Definition at line 143 of file lpspi_shared_function.h.

16.51.2.2.13 uint8_t∗ rxBuff

The buffer into which received bytes are placed

Definition at line 138 of file lpspi_shared_function.h.

16.51.2.2.14 volatile uint16_t rxCount

Number of bytes remaining to receive

Definition at line 140 of file lpspi_shared_function.h.

16.51.2.2.15 uint8_t rxDMAChannel

Channel number for DMA rx channel

Definition at line 145 of file lpspi_shared_function.h.

16.51.2.2.16 volatile uint16_t rxFrameCnt

Number of bytes from current frame which were already received

Definition at line 142 of file lpspi_shared_function.h.

16.51.2.2.17 transfer_status_t status

The status of the current

Definition at line 149 of file lpspi_shared_function.h.

16.51.2.2.18 lpspi_transfer_type transferType

Type of LPSPI transfer

Definition at line 147 of file lpspi_shared_function.h.

16.51.2.2.19 const uint8_t∗ txBuff

The buffer from which transmitted bytes are taken

Definition at line 137 of file lpspi_shared_function.h.

16.51.2.2.20 volatile uint16_t txCount

Number of bytes remaining to send

Definition at line 139 of file lpspi_shared_function.h.

16.51.2.2.21 uint8_t txDMAChannel

Channel number for DMA tx channel

Definition at line 146 of file lpspi_shared_function.h.

16.51.2.2.22 volatile uint16_t txFrameCnt

Number of bytes from current frame which were already sent

Definition at line 141 of file lpspi_shared_function.h.

16.51.2.3 struct lpspi_slave_config_t

User configuration structure for the SPI slave driver. Implements : lpspi_slave_config_t_Class.

Definition at line 50 of file lpspi_slave_driver.h.

Data Fields

- lpspi_signal_polarity_t pcsPolarity
- uint16_t bitcount
- lpspi_clock_phase_t clkPhase
- lpspi_which_pcs_t whichPcs
- lpspi_sck_polarity_t clkPolarity
- bool lsbFirst
- lpspi_transfer_type transferType
- uint8_t rxDMAChannel
- uint8_t txDMAChannel
- spi_callback_t callback
- void ∗ callbackParam

Field Documentation

16.51.2.3.1   uint16_t bitcount

Number of bits/frame, minimum is 8-bits

Definition at line 53 of file lpspi_slave_driver.h.

16.51.2.3.2   spi_callback_t callback

Select the callback to transfer complete

Definition at line 61 of file lpspi_slave_driver.h.

16.51.2.3.3   void∗ callbackParam

Select additional callback parameters if it's necessary

Definition at line 62 of file lpspi_slave_driver.h.

16.51.2.3.4   lpspi_clock_phase_t clkPhase

Selects which phase of clock to capture data

Definition at line 54 of file lpspi_slave_driver.h.

16.51.2.3.5   lpspi_sck_polarity_t clkPolarity

Selects clock polarity

Definition at line 56 of file lpspi_slave_driver.h.

16.51.2.3.6   bool lsbFirst

Option to transmit LSB first

Definition at line 57 of file lpspi_slave_driver.h.

16.51.2.3.7   lpspi_signal_polarity_t pcsPolarity

PCS polarity

Definition at line 52 of file lpspi_slave_driver.h.

16.51.2.3.8   uint8_t rxDMAChannel

Channel number for DMA rx channel. If DMA mode isn't used this field will be ignored.

Definition at line 59 of file lpspi_slave_driver.h.

16.51.2.3.9 lpspi_transfer_type transferType

Type of LPSPI transfer

Definition at line 58 of file lpspi_slave_driver.h.

16.51.2.3.10 uint8_t txDMAChannel

Channel number for DMA tx channel. If DMA mode isn't used this field will be ignored.

Definition at line 60 of file lpspi_slave_driver.h.

16.51.2.3.11 lpspi_which_pcs_t whichPcs

Definition at line 55 of file lpspi_slave_driver.h.

16.51.3 Enumeration Type Documentation

16.51.3.1 enum lpspi_clock_phase_t

LPSPI clock phase configuration. Implements : lpspi_clock_phase_t_Class.

Enumerator

> ***LPSPI_CLOCK_PHASE_1ST_EDGE*** Data captured on SCK 1st edge, changed on 2nd.
> ***LPSPI_CLOCK_PHASE_2ND_EDGE*** Data changed on SCK 1st edge, captured on 2nd.

Definition at line 83 of file lpspi_shared_function.h.

16.51.3.2 enum lpspi_sck_polarity_t

LPSPI Clock Signal (SCK) Polarity configuration. Implements : lpspi_sck_polarity_t_Class.

Enumerator

> ***LPSPI_SCK_ACTIVE_HIGH*** Signal is Active High (idles low).
> ***LPSPI_SCK_ACTIVE_LOW*** Signal is Active Low (idles high).

Definition at line 92 of file lpspi_shared_function.h.

16.51.3.3 enum lpspi_signal_polarity_t

LPSPI Signal (PCS and Host Request) Polarity configuration. Implements : lpspi_signal_polarity_t_Class.

Enumerator

> ***LPSPI_ACTIVE_HIGH*** Signal is Active High (idles low).
> ***LPSPI_ACTIVE_LOW*** Signal is Active Low (idles high).

Definition at line 74 of file lpspi_shared_function.h.

16.51.3.4 enum lpspi_transfer_type

Type of LPSPI transfer (based on interrupts or DMA). Implements : lpspi_transfer_type_Class.

Enumerator

> ***LPSPI_USING_DMA*** The driver will use DMA to perform SPI transfer
> ***LPSPI_USING_INTERRUPTS*** The driver will use interrupts to perform SPI transfer

Definition at line 102 of file lpspi_shared_function.h.

16.51.3.5   enum lpspi_which_pcs_t

LPSPI Peripheral Chip Select (PCS) configuration (which PCS to configure). Implements : lpspi_which_pcs_t_↩
Class.

Enumerator

>   ***LPSPI_PCS0***   PCS[0]
>   ***LPSPI_PCS1***   PCS[1]
>   ***LPSPI_PCS2***   PCS[2]
>   ***LPSPI_PCS3***   PCS[3]

Definition at line 63 of file lpspi_shared_function.h.

16.51.3.6   enum transfer_status_t

Type of error reported by LPSPI.

Enumerator

>   ***LPSPI_TRANSFER_OK***   Transfer OK
>   ***LPSPI_TRANSMIT_FAIL***   Error during transmission
>   ***LPSPI_RECEIVE_FAIL***   Error during reception

Definition at line 110 of file lpspi_shared_function.h.

16.51.4   Function Documentation

16.51.4.1   void LPSPI0_IRQHandler (void)

This function is the implementation of LPSPI0 handler named in startup code.

It passes the instance to the shared LPSPI IRQ handler.

Definition at line 109 of file lpspi_irq.c.

16.51.4.2   void LPSPI1_IRQHandler (void)

This function is the implementation of LPSPI1 handler named in startup code.

It passes the instance to the shared LPSPI IRQ handler.

Definition at line 119 of file lpspi_irq.c.

16.51.4.3   void LPSPI2_IRQHandler (void)

This function is the implementation of LPSPI2 handler named in startup code.

It passes the instance to the shared LPSPI IRQ handler.

Definition at line 129 of file lpspi_irq.c.

16.51.4.4   void LPSPI_DRV_DisableTEIEInterrupts (uint32_t *instance*)

Disable the TEIE interrupts at the end of a transfer. Disable the interrupts and clear the status for transmit/receive
errors.

Definition at line 278 of file lpspi_shared_function.c.

16.51.4.5   void LPSPI_DRV_FillupTxBuffer (uint32_t *instance*)

The function LPSPI_DRV_FillupTxBuffer writes data in TX hardware buffer depending on driver state and number
of bytes remained to send.

The function LPSPI_DRV_FillupTxBuffer writes data in TX hardware buffer depending on driver state and number of bytes remained to send.

Definition at line 126 of file lpspi_shared_function.c.

16.51.4.6   void LPSPI_DRV_IRQHandler (uint32_t *instance*)

The function LPSPI_DRV_IRQHandler passes IRQ control to either the master or slave driver.

The address of the IRQ handlers are checked to make sure they are non-zero before they are called. If the IRQ handler's address is zero, it means that driver was not present in the link (because the IRQ handlers are marked as weak). This would actually be a program error, because it means the master/slave config for the IRQ was set incorrectly.

Definition at line 103 of file lpspi_shared_function.c.

16.51.4.7   status_t LPSPI_DRV_MasterAbortTransfer (uint32_t *instance*)

Terminates an interrupt driven asynchronous transfer early.

During an a-sync (non-blocking) transfer, the user has the option to terminate the transfer early if the transfer is still in progress.

Parameters

instance	The instance number of the LPSPI peripheral.

Returns

> STATUS_SUCCESS The transfer was successful, or LPSPI_STATUS_NO_TRANSFER_IN_PROGRESS No transfer is currently in progress.

Definition at line 570 of file lpspi_master_driver.c.

16.51.4.8   status_t LPSPI_DRV_MasterConfigureBus (uint32_t *instance,* const lpspi_master_config_t ∗ *spiConfig,* uint32_t ∗ *calculatedBaudRate*)

Configures the LPSPI port physical parameters to access a device on the bus when the LSPI instance is configured for interrupt operation.

In this function, the term "spiConfig" is used to indicate the SPI device for which the LPSPI master is communicating. This is an optional function as the spiConfig parameters are normally configured in the initialization function or the transfer functions, where these various functions would call the configure bus function. This is an example to set up the lpspi_master_config_t structure to call the LPSPI_DRV_MasterConfigureBus function by passing in these parameters:

```
1 lpspi_master_config_t spiConfig1;    You can also declare spiConfig2, spiConfig3, etc
2 spiConfig1.bitsPerSec = 500000;
3 spiConfig1.whichPcs = LPSPI_PCS0;
4 spiConfig1.pcsPolarity = LPSPI_ACTIVE_LOW;
5 spiConfig1.isPcsContinuous = false;
6 spiConfig1.bitCount = 16;
7 spiConfig1.clkPhase = LPSPI_CLOCK_PHASE_1ST_EDGE;
8 spiConfig1.clkPolarity = LPSPI_ACTIVE_HIGH;
9 spiConfig1.lsbFirst= false;
10 spiConfig.transferType = LPSPI_USING_INTERRUPTS;
```

Parameters

instance	The instance number of the LPSPI peripheral.
spiConfig	Pointer to the spiConfig structure. This structure contains the settings for the SPI bus configuration. The SPI device parameters are the desired baud rate (in bits-per-sec), bits-per-frame, chip select attributes, clock attributes, and data shift direction.

| calculated↩ | The calculated baud rate passed back to the user to determine if the calculated baud rate is |
| BaudRate | close enough to meet the needs. The baud rate never exceeds the desired baud rate. |

Returns

> STATUS_SUCCESS The transfer has completed successfully, or STATUS_ERROR if driver is error and need-
> s to clean error.

Definition at line 315 of file lpspi_master_driver.c.

16.51.4.9   status_t LPSPI_DRV_MasterDeinit (uint32_t *instance*)

Shuts down a LPSPI instance.

This function resets the LPSPI peripheral, gates its clock, and disables the interrupt to the core. It first checks to
see if a transfer is in progress and if so returns an error status.

Parameters

instance	The instance number of the LPSPI peripheral.

Returns

> STATUS_SUCCESS The transfer has completed successfully, or STATUS_BUSY The transfer is still in
> progress. STATUS_ERROR if driver is error and needs to clean error.

Definition at line 193 of file lpspi_master_driver.c.

16.51.4.10   status_t LPSPI_DRV_MasterGetTransferStatus (uint32_t *instance,* uint32_t ∗ *bytesRemained*)

Returns whether the previous interrupt driven transfer is completed.

When performing an a-sync (non-blocking) transfer, the user can call this function to ascertain the state of the
current transfer: in progress (or busy) or complete (success). In addition, if the transfer is still in progress, the user
can get the number of words that have been transferred up to now.

Parameters

instance	The instance number of the LPSPI peripheral.
bytesRemained	Pointer to a value that is filled in with the number of bytes that must be received.

Returns

> STATUS_SUCCESS The transfer has completed successfully, or STATUS_BUSY The transfer is still in
> progress. framesTransferred is filled with the number of words that have been transferred so far.

Definition at line 541 of file lpspi_master_driver.c.

16.51.4.11   status_t LPSPI_DRV_MasterInit (uint32_t *instance,* lpspi_state_t ∗ *lpspiState,* const lpspi_master_config_t
         ∗ *spiConfig*)

Initializes a LPSPI instance for interrupt driven master mode operation.

This function uses an interrupt-driven method for transferring data. In this function, the term "spiConfig" is used
to indicate the SPI device for which the LPSPI master is communicating. This function initializes the run-time
state structure to track the ongoing transfers, un-gates the clock to the LPSPI module, resets the LPSPI module,
configures the IRQ state structure, enables the module-level interrupt to the core, and enables the LPSPI module.
This is an example to set up the lpspi_master_state_t and call the LPSPI_DRV_MasterInit function by passing in
these parameters:

```
1 lpspi_master_state_t lpspiMasterState;  <- the user  allocates memory for this structure
2 lpspi_master_config_t spiConfig;  Can declare more configs for use in transfer functions
```

```
3 spiConfig.bitsPerSec = 500000;
4 spiConfig.whichPcs = LPSPI_PCS0;
5 spiConfig.pcsPolarity = LPSPI_ACTIVE_LOW;
6 spiConfig.isPcsContinuous = false;
7 spiConfig.bitCount = 16;
8 spiConfig.clkPhase = LPSPI_CLOCK_PHASE_1ST_EDGE;
9 spiConfig.clkPolarity = LPSPI_ACTIVE_HIGH;
10 spiConfig.lsbFirst= false;
11 spiConfig.transferType = LPSPI_USING_INTERRUPTS;
12 LPSPI_DRV_MasterInit(masterInstance, &lpspiMasterState, &spiConfig);
```

Parameters

instance	The instance number of the LPSPI peripheral.
lpspiState	The pointer to the LPSPI master driver state structure. The user passes the memory for this run-time state structure. The LPSPI master driver populates the members. This run-time state structure keeps track of the transfer in progress.
spiConfig	The data structure containing information about a device on the SPI bus

Returns

> An error code or STATUS_SUCCESS.

Definition at line 140 of file lpspi_master_driver.c.

16.51.4.12   void LPSPI_DRV_MasterIRQHandler (uint32_t *instance*)

Interrupt handler for LPSPI master mode. This handler uses the buffers stored in the lpspi_master_state_t structs to transfer data.

Parameters

instance	The instance number of the LPSPI peripheral.

Interrupt handler for LPSPI master mode. This handler uses the buffers stored in the lpspi_master_state_t structs to transfer data.

Definition at line 847 of file lpspi_master_driver.c.

16.51.4.13   status_t LPSPI_DRV_MasterSetDelay (uint32_t *instance,* uint32_t *delayBetwenTransfers,* uint32_t *delaySCKtoPCS,* uint32_t *delayPCStoSCK*)

Configures the LPSPI master mode bus timing delay options.

This function involves the LPSPI module's delay options to "fine tune" some of the signal timings and match the timing needs of a slower peripheral device. This is an optional function that can be called after the LPSPI module has been initialized for master mode. The timings are adjusted in terms of cycles of the baud rate clock. The bus timing delays that can be adjusted are listed below:

SCK to PCS Delay: Adjustable delay option between the last edge of SCK to the de-assertion of the PCS signal.

PCS to SCK Delay: Adjustable delay option between the assertion of the PCS signal to the first SCK edge.

Delay between Transfers: Adjustable delay option between the de-assertion of the PCS signal for a frame to the assertion of the PCS signal for the next frame.

Parameters

instance	The instance number of the LPSPI peripheral.
delayBetwen↩ Transfers	Minimum delay between 2 transfers in microseconds
delaySCKtoP↩ CS	Minimum delay between SCK and PCS

delayPCStoS↩ *CK*	Minimum delay between PCS and SCK

Returns

> STATUS_SUCCESS The transfer has completed successfully, or STATUS_ERROR if driver is error and need-
> s to clean error.

Definition at line 240 of file lpspi_master_driver.c.

16.51.4.14   status_t LPSPI_DRV_MasterTransfer (uint32_t *instance,* const uint8_t ∗ *sendBuffer,* uint8_t ∗ *receiveBuffer,* uint16_t *transferByteCount*)

Performs an interrupt driven non-blocking SPI master mode transfer.

This function simultaneously sends and receives data on the SPI bus, as SPI is naturally a full-duplex bus. The
function returns immediately after initiating the transfer. The user needs to check whether the transfer is complete
using the LPSPI_DRV_MasterGetTransferStatus function. This function allows the user to optionally pass in a SPI
configuration structure which allows the user to change the SPI bus attributes in conjunction with initiating a SPI
transfer. The difference between passing in the SPI configuration structure here as opposed to the configure bus
function is that the configure bus function returns the calculated baud rate where this function does not. The user
can also call the configure bus function prior to the transfer in which case the user would simply pass in a NULL to
the transfer function's device structure parameter.

Parameters

instance	The instance number of the LPSPI peripheral.
spiConfig	Pointer to the SPI configuration structure. This structure contains the settings for the SPI bus configuration in this transfer. You may pass NULL for this parameter, in which case the current bus configuration is used unmodified. The device can be configured separately by calling the LPSPI_DRV_MasterConfigureBus function.
sendBuffer	The pointer to the data buffer of the data to send. You may pass NULL for this parameter and bytes with a value of 0 (zero) is sent.
receiveBuffer	Pointer to the buffer where the received bytes are stored. If you pass NULL for this parameter, the received bytes are ignored.
transferByte↩ *Count*	The number of bytes to send and receive.

Returns

> STATUS_SUCCESS The transfer was successful, or STATUS_BUSY Cannot perform transfer because a
> transfer is already in progress

Definition at line 503 of file lpspi_master_driver.c.

16.51.4.15   status_t LPSPI_DRV_MasterTransferBlocking (uint32_t *instance,* const uint8_t ∗ *sendBuffer,* uint8_t ∗ *receiveBuffer,* uint16_t *transferByteCount,* uint32_t *timeout*)

Performs an interrupt driven blocking SPI master mode transfer.

This function simultaneously sends and receives data on the SPI bus, as SPI is naturally a full-duplex bus. The
function does not return until the transfer is complete. This function allows the user to optionally pass in a SPI
configuration structure which allows the user to change the SPI bus attributes in conjunction with initiating a SPI
transfer. The difference between passing in the SPI configuration structure here as opposed to the configure bus
function is that the configure bus function returns the calculated baud rate where this function does not. The user
can also call the configure bus function prior to the transfer in which case the user would simply pass in a NULL to
the transfer function's device structure parameter.

Parameters

instance	The instance number of the LPSPI peripheral.
sendBuffer	The pointer to the data buffer of the data to send. You may pass NULL for this parameter and bytes with a value of 0 (zero) is sent.
receiveBuffer	Pointer to the buffer where the received bytes are stored. If you pass NULL for this parameter, the received bytes are ignored.
transferByte↩ Count	The number of bytes to send and receive.
timeout	A timeout for the transfer in milliseconds. If the transfer takes longer than this amount of time, the transfer is aborted and a STATUS_TIMEOUT error returned.

Returns

> STATUS_SUCCESS The transfer was successful, or STATUS_BUSY Cannot perform transfer because a transfer is already in progress, or STATUS_TIMEOUT The transfer timed out and was aborted.

Definition at line 424 of file lpspi_master_driver.c.

16.51.4.16 void LPSPI_DRV_ReadRXBuffer (uint32_t *instance*)

The function LPSPI_DRV_ReadRXBuffer reads data from RX hardware buffer and writes this data in RX software buffer.

The function LPSPI_DRV_ReadRXBuffer reads data from RX hardware buffer and writes this data in RX software buffer.

Definition at line 217 of file lpspi_shared_function.c.

16.51.4.17 status_t LPSPI_DRV_SetPcs (uint32_t *instance,* lpspi_which_pcs_t *whichPcs,* lpspi_signal_polarity_t *polarity*)

Select the chip to communicate with.

The main purpose of this function is to set the PCS and the appropriate polarity.

Parameters

instance	LPSPI instance
whichPcs	selected chip
polarity	chip select line polarity

Returns

> STATUS_SUCCESS The transfer has completed successfully, or STATUS_ERROR if driver is error and needs to clean error.

Definition at line 592 of file lpspi_master_driver.c.

16.51.4.18 status_t LPSPI_DRV_SlaveAbortTransfer (uint32_t *instance*)

Aborts the transfer that started by a non-blocking call transfer function.

This function stops the transfer which was started by the calling the SPI_DRV_SlaveTransfer() function.

Parameters

instance	The instance number of SPI peripheral

Returns

> STATUS_SUCCESS if everything is OK.

Definition at line 444 of file lpspi_slave_driver.c.

16.51.4.19   status_t LPSPI_DRV_SlaveDeinit (uint32_t *instance*)

Shuts down an LPSPI instance interrupt mechanism.

Disables the LPSPI module, gates its clock, and changes the LPSPI slave driver state to NonInit for the LPSPI slave module which is initialized with interrupt mechanism. After de-initialization, the user can re-initialize the LPSPI slave module with other mechanisms.

Parameters

instance	The instance number of the LPSPI peripheral.

Returns

> STATUS_SUCCESS if driver starts to send/receive data successfully. STATUS_ERROR if driver is error and needs to clean error. STATUS_BUSY if a transfer is in progress

Definition at line 175 of file lpspi_slave_driver.c.

16.51.4.20   status_t LPSPI_DRV_SlaveGetTransferStatus (uint32_t *instance,* uint32_t ∗ *bytesRemained*)

Returns whether the previous transfer is finished.

When performing an a-sync transfer, the user can call this function to ascertain the state of the current transfer: in progress (or busy) or complete (success). In addition, if the transfer is still in progress, the user can get the number of words that have been transferred up to now.

Parameters

instance	The instance number of the LPSPI peripheral.
bytesRemained	Pointer to value that is filled in with the number of frames that have been sent in the active transfer. A frame is defined as the number of bits per frame.

Returns

> STATUS_SUCCESS The transfer has completed successfully, or STATUS_BUSY The transfer is still in progress. STATUS_ERROR if driver is error and needs to clean error.

Definition at line 482 of file lpspi_slave_driver.c.

16.51.4.21   status_t LPSPI_DRV_SlaveInit (uint32_t *instance,* lpspi_state_t ∗ *lpspiState,* const lpspi_slave_config_t ∗ *slaveConfig*)

Initializes a LPSPI instance for a slave mode operation, using interrupt mechanism.

This function un-gates the clock to the LPSPI module, initializes the LPSPI for slave mode. After it is initialized, the LPSPI module is configured in slave mode and the user can start transmitting and receiving data by calling send, receive, and transfer functions. This function indicates LPSPI slave uses an interrupt mechanism.

Parameters

instance	The instance number of the LPSPI peripheral.
lpspiState	The pointer to the LPSPI slave driver state structure.
slaveConfig	The configuration structure lpspi_slave_user_config_t which configures the data bus format.

Returns

> An error code or STATUS_SUCCESS.

Definition at line 103 of file lpspi_slave_driver.c.

16.51.4.22   void LPSPI_DRV_SlaveIRQHandler (uint32_t *instance*)

Interrupt handler for LPSPI slave mode. This handler uses the buffers stored in the lpspi_master_state_t structs to transfer data.

Parameters

instance	The instance number of the LPSPI peripheral.

Definition at line 374 of file lpspi_slave_driver.c.

16.51.4.23   status_t LPSPI_DRV_SlaveTransfer (uint32_t *instance,* const uint8_t * *sendBuffer,* uint8_t * *receiveBuffer,* uint16_t *transferByteCount*)

Starts the transfer data on LPSPI bus using an interrupt and a non-blocking call.

Parameters

instance	The instance number of LPSPI peripheral
sendBuffer	The pointer to data that user wants to transmit.
receiveBuffer	The pointer to data that user wants to store received data.
transferByte↵ Count	The number of bytes to send and receive.

Returns

> STATUS_SUCCESS if driver starts to send/receive data successfully. STATUS_ERROR if driver is error and needs to clean error. STATUS_BUSY if a transfer is in progress

Definition at line 243 of file lpspi_slave_driver.c.

16.51.4.24   status_t LPSPI_DRV_SlaveTransferBlocking (uint32_t *instance,* const uint8_t * *sendBuffer,* uint8_t * *receiveBuffer,* uint16_t *transferByteCount,* uint32_t *timeout*)

Transfers data on LPSPI bus using interrupt and a blocking call.

This function checks the driver status and mechanism, and transmits/receives data through the LPSPI bus. If the sendBuffer is NULL, the transmit process is ignored. If the receiveBuffer is NULL, the receive process is ignored. If both the receiveBuffer and the sendBuffer are available, the transmit and the receive progress is processed. If only the receiveBuffer is available, the receive is processed. Otherwise, the transmit is processed. This function only returns when the processes are completed. This function uses an interrupt mechanism.

Parameters

instance	The instance number of LPSPI peripheral
sendBuffer	The pointer to data that user wants to transmit.
receiveBuffer	The pointer to data that user wants to store received data.
transferByte↵ Count	The number of bytes to send and receive.
timeout	The maximum number of milliseconds that function waits before timed out reached.

Returns

> STATUS_SUCCESS if driver starts to send/receive data successfully. STATUS_ERROR if driver is error and needs to clean error. STATUS_BUSY if a transfer is in progress STATUS_TIMEOUT if time out reached while transferring is in progress.

Definition at line 204 of file lpspi_slave_driver.c.

16.51.5   Variable Documentation

16.51.5.1   LPSPI_Type∗ g_lpspiBase[LPSPI_INSTANCE_COUNT]

Table of base pointers for SPI instances.

Definition at line 81 of file lpspi_shared_function.c.

16.51.5.2   IRQn_Type g_lpspilrqld[LPSPI_INSTANCE_COUNT]

Table to save LPSPI IRQ enumeration numbers defined in the CMSIS header file.

Definition at line 84 of file lpspi_shared_function.c.

16.51.5.3   lpspi_state_t∗ g_lpspiStatePtr[LPSPI_INSTANCE_COUNT]

Definition at line 87 of file lpspi_shared_function.c.

16.52 LPTMR Driver

16.52.1 Detailed Description

Low Power Timer Peripheral Driver.

The LPTMR is a configurable general-purpose 16-bit counter that has two operational modes: Timer and Pulse-↩ Counter.

Depending on the configured operational mode, the counter in the LPTMR can be incremented using a clock input (Timer mode) or an event counter (external events like button presses or internal events from different trigger sources).

Timer Mode

In Timer mode, the LPTMR increments the internal counter from a selectable clock source. An optional 16-bit prescaler can be configured.

Pulse-Counter Mode

In Pulse-Counter Mode, the LPTMR counter increments from a selectable trigger source, input pin, which can be an external event (like a button press) or internal events (like triggers from TRGMUX).

An optional 16-bit glitch-filter can be configured to reject events that have a duration below a set period.

Initialization prerequisites

Before configuring the LPTMR, the peripheral clock must be enabled from the PCC module.

The peripheral clock must not be confused with the counter clock, which is selectable within the LPTMR.

Driver configuration

The LPTMR driver allows configuring the LPTMR for Pulse-Counter Mode or Timer Mode via the general configuration structure.

Configurable options:

- work mode (timer or pulse-counter)

- enable interrupts and DMA requests

- free running mode (overflow mode of the counter)

- compare value (interrupt generation on counter value)

- compare value measurement units (counter ticks or microseconds)

- input clock selection

- prescaler/glitch filter configuration

- enable bypass prescaler

- pin select (for pulse-counter mode)

- input pin and polarity (for pulse-counter mode)

```
/* LPTMR initialization of config structure */
lptmr_config_t config = {
  .workMode = LPTMR_WORKMODE_TIMER,
  .dmaRequest = false,
  .interruptEnable = false,
  .freeRun = false,
  .compareValue = 1000U,
  .counterUnits = LPTMR_COUNTER_UNITS_TICKS,
  .clockSelect = LPTMR_CLOCKSOURCE_SIRCDIV2,
  .prescaler = LPTMR_PRESCALE_2,
  .bypassPrescaler = false,
```

```
      .pinSelect = LPTMR_PINSELECT_TRGMUX,
      .pinPolarity = LPTMR_PINPOLARITY_RISING,
};

/* Initialize the LPTMR and start the counter in a separate operation */
status = LPTMR_DRV_Init(0, &config, false);
/* Start timer counting */
LPTMR_DRV_StartCounter(0);
```

API

Some of the features exposed by the API are targeted specifically for Timer Mode or Pulse-Counter Mode. For example, configuring the Compare Value in microseconds makes sense only for Timer Mode, so therefor it is restricted for use in Pulse-Counter mode.

For any invalid configuration the functions will either return an error code or trigger DEV_ASSERT (if enabled). For more details, please refer to each function description.

Data Structures

- struct lptmr_config_t

    Defines the configuration structure for LPTMR. More...

Enumerations

- enum lptmr_pinselect_t { LPTMR_PINSELECT_TRGMUX = 0x00u, LPTMR_PINSELECT_ALT2 = 0x02u, LPTMR_PINSELECT_ALT3 = 0x03u }

    Pulse Counter Input selection Implements : lptmr_pinselect_t_Class.

- enum lptmr_pinpolarity_t { LPTMR_PINPOLARITY_RISING = 0u, LPTMR_PINPOLARITY_FALLING = 1u }

    Pulse Counter input polarity Implements : lptmr_pinpolarity_t_Class.

- enum lptmr_workmode_t { LPTMR_WORKMODE_TIMER = 0u, LPTMR_WORKMODE_PULSECOUNTER = 1u }

    Work Mode Implements : lptmr_workmode_t_Class.

- enum lptmr_prescaler_t {
    LPTMR_PRESCALE_2 = 0x00u, LPTMR_PRESCALE_4_GLITCHFILTER_2 = 0x01u, LPTMR_PRESCA↩
    LE_8_GLITCHFILTER_4 = 0x02u, LPTMR_PRESCALE_16_GLITCHFILTER_8 = 0x03u,
    LPTMR_PRESCALE_32_GLITCHFILTER_16 = 0x04u, LPTMR_PRESCALE_64_GLITCHFILTER_32 =
    0x05u, LPTMR_PRESCALE_128_GLITCHFILTER_64 = 0x06u, LPTMR_PRESCALE_256_GLITCHFILT↩
    ER_128 = 0x07u,
    LPTMR_PRESCALE_512_GLITCHFILTER_256 = 0x08u, LPTMR_PRESCALE_1024_GLITCHFILTER_512
    = 0x09u, LPTMR_PRESCALE_2048_GLITCHFILTER_1024 = 0x0Au, LPTMR_PRESCALE_4096_GLITC↩
    HFILTER_2048 = 0x0Bu,
    LPTMR_PRESCALE_8192_GLITCHFILTER_4096 = 0x0Cu, LPTMR_PRESCALE_16384_GLITCHFILTE↩
    R_8192 = 0x0Du, LPTMR_PRESCALE_32768_GLITCHFILTER_16384 = 0x0Eu, LPTMR_PRESCALE_↩
    65536_GLITCHFILTER_32768 = 0x0Fu }

    Prescaler Selection Implements : lptmr_prescaler_t_Class.

- enum lptmr_clocksource_t { LPTMR_CLOCKSOURCE_SIRCDIV2 = 0x00u, LPTMR_CLOCKSOURCE_1↩
    KHZ_LPO = 0x01u, LPTMR_CLOCKSOURCE_RTC = 0x02u, LPTMR_CLOCKSOURCE_PCC = 0x03u }

    Clock Source selection Implements : lptmr_clocksource_t_Class.

- enum lptmr_counter_units_t { LPTMR_COUNTER_UNITS_TICKS = 0x00U, LPTMR_COUNTER_UNITS_↩
    MICROSECONDS = 0x01U }

    Defines the LPTMR counter units available for configuring or reading the timer compare value.

LPTMR Driver Functions

- void LPTMR_DRV_InitConfigStruct (lptmr_config_t *const config)

    Initialize a configuration structure with default values.

- void LPTMR_DRV_Init (const uint32_t instance, const lptmr_config_t ∗const config, const bool startCounter)

  Initialize a LPTMR instance with values from an input configuration structure.
- void LPTMR_DRV_SetConfig (const uint32_t instance, const lptmr_config_t ∗const config)

  Configure a LPTMR instance.
- void LPTMR_DRV_GetConfig (const uint32_t instance, lptmr_config_t ∗const config)

  Get the current configuration of a LPTMR instance.
- void LPTMR_DRV_Deinit (const uint32_t instance)

  De-initialize a LPTMR instance.
- status_t LPTMR_DRV_SetCompareValueByCount (const uint32_t instance, const uint16_t compareValue↩ByCount)

  Set the compare value in counter tick units, for a LPTMR instance.
- void LPTMR_DRV_GetCompareValueByCount (const uint32_t instance, uint16_t ∗const compareValueBy↩Count)

  Get the compare value in counter tick units, of a LPTMR instance.
- status_t LPTMR_DRV_SetCompareValueByUs (const uint32_t instance, const uint32_t compareValueUs)

  Set the compare value for Timer Mode in microseconds, for a LPTMR instance.
- void LPTMR_DRV_GetCompareValueByUs (const uint32_t instance, uint32_t ∗const compareValueUs)

  Get the compare value in microseconds, of a LPTMR instance.
- bool LPTMR_DRV_GetCompareFlag (const uint32_t instance)

  Get the current state of the Compare Flag of a LPTMR instance.
- void LPTMR_DRV_ClearCompareFlag (const uint32_t instance)

  Clear the Compare Flag of a LPTMR instance.
- bool LPTMR_DRV_IsRunning (const uint32_t instance)

  Get the run state of a LPTMR instance.
- void LPTMR_DRV_SetInterrupt (const uint32_t instance, const bool enableInterrupt)

  Enable/disable the LPTMR interrupt.
- uint16_t LPTMR_DRV_GetCounterValueByCount (const uint32_t instance)

  Get the current counter value in counter tick units.
- void LPTMR_DRV_StartCounter (const uint32_t instance)

  Enable the LPTMR / Start the counter.
- void LPTMR_DRV_StopCounter (const uint32_t instance)

  Disable the LPTMR / Stop the counter.
- void LPTMR_DRV_SetPinConfiguration (const uint32_t instance, const lptmr_pinselect_t pinSelect, const lptmr_pinpolarity_t pinPolarity)

  Set the Input Pin configuration for Pulse Counter mode.

16.52.2 Data Structure Documentation

16.52.2.1 struct lptmr_config_t

Defines the configuration structure for LPTMR.

Implements : lptmr_config_t_Class

Definition at line 113 of file lptmr_driver.h.

Data Fields

- bool dmaRequest
- bool interruptEnable
- bool freeRun
- lptmr_workmode_t workMode
- lptmr_clocksource_t clockSelect
- lptmr_prescaler_t prescaler

- bool [bypassPrescaler](#)
- uint32_t [compareValue](#)
- [lptmr_counter_units_t counterUnits](#)
- [lptmr_pinselect_t pinSelect](#)
- [lptmr_pinpolarity_t pinPolarity](#)

Field Documentation

16.52.2.1.1   bool bypassPrescaler

Enable/Disable prescaler bypass

Definition at line 123 of file lptmr_driver.h.

16.52.2.1.2   lptmr_clocksource_t clockSelect

Clock selection for Timer/Glitch filter

Definition at line 121 of file lptmr_driver.h.

16.52.2.1.3   uint32_t compareValue

Compare value

Definition at line 124 of file lptmr_driver.h.

16.52.2.1.4   lptmr_counter_units_t counterUnits

Compare value units

Definition at line 125 of file lptmr_driver.h.

16.52.2.1.5   bool dmaRequest

Enable/Disable DMA requests

Definition at line 116 of file lptmr_driver.h.

16.52.2.1.6   bool freeRun

Enable/Disable Free Running Mode

Definition at line 118 of file lptmr_driver.h.

16.52.2.1.7   bool interruptEnable

Enable/Disable Interrupt

Definition at line 117 of file lptmr_driver.h.

16.52.2.1.8   lptmr_pinpolarity_t pinPolarity

Pin Polarity for Pulse-Counter

Definition at line 128 of file lptmr_driver.h.

16.52.2.1.9   lptmr_pinselect_t pinSelect

Pin selection for Pulse-Counter

Definition at line 127 of file lptmr_driver.h.

16.52.2.1.10   lptmr_prescaler_t prescaler

Prescaler Selection

Definition at line 122 of file lptmr_driver.h.

16.52.2.1.11 lptmr_workmode_t workMode

Time/Pulse Counter Mode

Definition at line 119 of file lptmr_driver.h.

16.52.3 Enumeration Type Documentation

16.52.3.1 enum lptmr_clocksource_t

Clock Source selection Implements : lptmr_clocksource_t_Class.

Enumerator

>*LPTMR_CLOCKSOURCE_SIRCDIV2* SIRC clock
>
>*LPTMR_CLOCKSOURCE_1KHZ_LPO* 1kHz LPO clock
>
>*LPTMR_CLOCKSOURCE_RTC* RTC clock
>
>*LPTMR_CLOCKSOURCE_PCC* PCC configured clock

Definition at line 90 of file lptmr_driver.h.

16.52.3.2 enum lptmr_counter_units_t

Defines the LPTMR counter units available for configuring or reading the timer compare value.

Implements : lptmr_counter_units_t_Class

Enumerator

>*LPTMR_COUNTER_UNITS_TICKS*
>
>*LPTMR_COUNTER_UNITS_MICROSECONDS*

Definition at line 102 of file lptmr_driver.h.

16.52.3.3 enum lptmr_pinpolarity_t

Pulse Counter input polarity Implements : lptmr_pinpolarity_t_Class.

Enumerator

>*LPTMR_PINPOLARITY_RISING* Count pulse on rising edge
>
>*LPTMR_PINPOLARITY_FALLING* Count pulse on falling edge

Definition at line 52 of file lptmr_driver.h.

16.52.3.4 enum lptmr_pinselect_t

Pulse Counter Input selection Implements : lptmr_pinselect_t_Class.

Enumerator

>*LPTMR_PINSELECT_TRGMUX* Count pulses from TRGMUX trigger
>
>*LPTMR_PINSELECT_ALT2* Count pulses from pin alternative 2
>
>*LPTMR_PINSELECT_ALT3* Count pulses from pin alternative 3

Definition at line 40 of file lptmr_driver.h.

16.52.3.5   enum lptmr_prescaler_t

Prescaler Selection Implements : lptmr_prescaler_t_Class.

Enumerator

> *LPTMR_PRESCALE_2*   Timer mode: prescaler 2, Glitch filter mode: invalid
>
> *LPTMR_PRESCALE_4_GLITCHFILTER_2*   Timer mode: prescaler 4, Glitch filter mode: 2 clocks
>
> *LPTMR_PRESCALE_8_GLITCHFILTER_4*   Timer mode: prescaler 8, Glitch filter mode: 4 clocks
>
> *LPTMR_PRESCALE_16_GLITCHFILTER_8*   Timer mode: prescaler 16, Glitch filter mode: 8 clocks
>
> *LPTMR_PRESCALE_32_GLITCHFILTER_16*   Timer mode: prescaler 32, Glitch filter mode: 16 clocks
>
> *LPTMR_PRESCALE_64_GLITCHFILTER_32*   Timer mode: prescaler 64, Glitch filter mode: 32 clocks
>
> *LPTMR_PRESCALE_128_GLITCHFILTER_64*   Timer mode: prescaler 128, Glitch filter mode: 64 clocks
>
> *LPTMR_PRESCALE_256_GLITCHFILTER_128*   Timer mode: prescaler 256, Glitch filter mode: 128 clocks
>
> *LPTMR_PRESCALE_512_GLITCHFILTER_256*   Timer mode: prescaler 512, Glitch filter mode: 256 clocks
>
> *LPTMR_PRESCALE_1024_GLITCHFILTER_512*   Timer mode: prescaler 1024, Glitch filter mode: 512 clocks
>
> *LPTMR_PRESCALE_2048_GLITCHFILTER_1024*   Timer mode: prescaler 2048, Glitch filter mode: 1024 clocks
>
> *LPTMR_PRESCALE_4096_GLITCHFILTER_2048*   Timer mode: prescaler 4096, Glitch filter mode: 2048 clocks
>
> *LPTMR_PRESCALE_8192_GLITCHFILTER_4096*   Timer mode: prescaler 8192, Glitch filter mode: 4096 clocks
>
> *LPTMR_PRESCALE_16384_GLITCHFILTER_8192*   Timer mode: prescaler 16384, Glitch filter mode: 8192 clocks
>
> *LPTMR_PRESCALE_32768_GLITCHFILTER_16384*   Timer mode: prescaler 32768, Glitch filter mode↩ : 16384 clocks
>
> *LPTMR_PRESCALE_65536_GLITCHFILTER_32768*   Timer mode: prescaler 65536, Glitch filter mode↩ : 32768 clocks

Definition at line 68 of file lptmr_driver.h.

16.52.3.6   enum lptmr_workmode_t

Work Mode Implements : lptmr_workmode_t_Class.

Enumerator

> *LPTMR_WORKMODE_TIMER*   Timer
>
> *LPTMR_WORKMODE_PULSECOUNTER*   Pulse counter

Definition at line 60 of file lptmr_driver.h.

16.52.4   Function Documentation

16.52.4.1   void LPTMR_DRV_ClearCompareFlag (const uint32_t *instance*)

Clear the Compare Flag of a LPTMR instance.

Parameters

in	*instance*	- LPTMR instance number

16.52.4.2   void LPTMR_DRV_Deinit (const uint32_t *instance*)

De-initialize a LPTMR instance.

Parameters

in	*instance*	- LPTMR instance number

16.52.4.3   bool LPTMR_DRV_GetCompareFlag (const uint32_t *instance*)

Get the current state of the Compare Flag of a LPTMR instance.

Parameters

in	*instance*	- LPTMR instance number

Returns

> The state of the Compare Flag

16.52.4.4   void LPTMR_DRV_GetCompareValueByCount (const uint32_t *instance,* uint16_t ∗const *compareValueByCount*)

Get the compare value in counter tick units, of a LPTMR instance.

Parameters

in	*instance*	- LPTMR instance number
out	*compareValue↩ ByCount*	- Pointer to current compare value, in counter ticks

16.52.4.5   void LPTMR_DRV_GetCompareValueByUs (const uint32_t *instance,* uint32_t ∗const *compareValueUs*)

Get the compare value in microseconds, of a LPTMR instance.

Parameters

in	*instance*	- LPTMR instance number
out	*compareValue↩ Us*	- Pointer to current compare value, in microseconds

16.52.4.6   void LPTMR_DRV_GetConfig (const uint32_t *instance,* lptmr_config_t ∗const *config*)

Get the current configuration of a LPTMR instance.

Parameters

in	*instance*	- LPTMR instance number
out	*config*	- Pointer to the output configuration structure

16.52.4.7   uint16_t LPTMR_DRV_GetCounterValueByCount (const uint32_t *instance*)

Get the current counter value in counter tick units.

Parameters

in	*instance*	- LPTMR instance number

Returns

> The current counter value

16.52.4.8   void LPTMR_DRV_Init (const uint32_t *instance,* const lptmr_config_t ∗const *config,* const bool *startCounter*)

Initialize a LPTMR instance with values from an input configuration structure.

When (counterUnits == LPTMR_COUNTER_UNITS_MICROSECONDS) the function will automatically configure the timer for the input compareValue in microseconds. The input parameters for 'prescaler' and 'bypassPrescaler' will be ignored - their values will be adapted by the function, to best fit the input compareValue (in microseconds) for the operating clock frequency.

LPTMR_COUNTER_UNITS_MICROSECONDS may only be used for LPTMR_WORKMODE_TIMER mode. Otherwise the function shall not convert 'compareValue' in ticks and this is likely to cause erroneous behavior.

When (counterUnits == LPTMR_COUNTER_UNITS_TICKS) the function will use the 'prescaler' and 'bypass↩ Prescaler' provided in the input configuration structure.

Parameters

in	*instance*	- LPTMR instance number
in	*config*	- Pointer to the input configuration structure
in	*startCounter*	- Flag for starting the counter immediately after configuration

16.52.4.9   void LPTMR_DRV_InitConfigStruct (lptmr_config_t ∗const *config*)

Initialize a configuration structure with default values.

Parameters

out	*config*	- Pointer to the configuration structure to be initialized

16.52.4.10   bool LPTMR_DRV_IsRunning (const uint32_t *instance*)

Get the run state of a LPTMR instance.

Parameters

in	*instance*	- LPTMR instance number

Returns

> The run state of the LPTMR instance:
> - true: Timer/Counter started
> - false: Timer/Counter stopped

16.52.4.11   status_t LPTMR_DRV_SetCompareValueByCount (const uint32_t *instance,* const uint16_t *compareValueByCount*)

Set the compare value in counter tick units, for a LPTMR instance.

Parameters

| in | *instance* | - LPTMR instance number |
| in | *compareValue↩ ByCount* | - The compare value in counter ticks, to be written |

Returns

Operation status:

- STATUS_SUCCESS: completed successfully

- STATUS_ERROR: cannot reconfigure compare value (TCF not set)

- STATUS_TIMEOUT: compare value greater then current counter value

16.52.4.12   status_t LPTMR_DRV_SetCompareValueByUs (const uint32_t *instance,* const uint32_t *compareValueUs*)

Set the compare value for Timer Mode in microseconds, for a LPTMR instance.

Parameters

| in | *instance* | - LPTMR peripheral instance number |
| in | *compareValue↩ Us* | - Compare value in microseconds |

Returns

Operation status:

- STATUS_SUCCESS: completed successfully

- STATUS_ERROR: cannot reconfigure compare value

- STATUS_TIMEOUT: compare value greater then current counter value

16.52.4.13   void LPTMR_DRV_SetConfig (const uint32_t *instance,* const lptmr_config_t ∗const *config*)

Configure a LPTMR instance.

When (counterUnits == LPTMR_COUNTER_UNITS_MICROSECONDS) the function will automatically configure the timer for the input compareValue in microseconds. The input parameters for 'prescaler' and 'bypassPrescaler' will be ignored - their values will be adapted by the function, to best fit the input compareValue (in microseconds) for the operating clock frequency.

LPTMR_COUNTER_UNITS_MICROSECONDS may only be used for LPTMR_WORKMODE_TIMER mode. Otherwise the function shall not convert 'compareValue' in ticks and this is likely to cause erroneous behavior.

When (counterUnits == LPTMR_COUNTER_UNITS_TICKS) the function will use the 'prescaler' and 'bypass↩ Prescaler' provided in the input configuration structure.

Parameters

| in | *instance* | - LPTMR instance number |
| in | *config* | - Pointer to the input configuration structure |

16.52.4.14   void LPTMR_DRV_SetInterrupt (const uint32_t *instance,* const bool *enableInterrupt*)

Enable/disable the LPTMR interrupt.

Parameters

| in | *instance* | - LPTMR instance number |

| in | *enableInterrupt* | - The new state of the LPTMR interrupt enable flag. |

16.52.4.15   void LPTMR_DRV_SetPinConfiguration (const uint32_t *instance,* const lptmr_pinselect_t *pinSelect,* const lptmr_pinpolarity_t *pinPolarity*)

Set the Input Pin configuration for Pulse Counter mode.

Parameters

in	*instance*	- LPTMR instance number
in	*pinSelect*	- LPTMR pin selection
in	*pinPolarity*	- Polarity on which to increment counter (rising/falling edge)

16.52.4.16   void LPTMR_DRV_StartCounter (const uint32_t *instance*)

Enable the LPTMR / Start the counter.

Parameters

| in | *instance* | - LPTMR instance number |

16.52.4.17   void LPTMR_DRV_StopCounter (const uint32_t *instance*)

Disable the LPTMR / Stop the counter.

Parameters

| in | *instance* | - LPTMR instance number |

16.53 LPUART Driver

16.53.1 Detailed Description

This module covers the functionality of the Low Power Universal Asynchronous Receiver-Transmitter (LPUART) peripheral driver.

The LPUART driver implements serial communication using the LPUART module in the S32K1xx platforms.

Features

- Interrupt based and polling communication

- Provides blocking and non-blocking transmit and receive functions

- Configurable baud rate

- 8/9/10 bits per char

Functionality

In order to use the LPUART driver it must be first initialized, using LPUART_DRV_Init() function. Once initialized, it cannot be initialized again for the same LPUART module instance until it is de-initialized, using LPUART_DRV_↩ Deinit(). The initialization function does the following operations:

- sets the baud rate

- sets parity/bit count/stop bits count

- initializes the state structure for the current instance

- enables receiver/transmitter for the current instance Different LPUART module instances can function independently of each other.

Interrupt-based communication

After initialization, a serial communication can be triggered by calling LPUART_DRV_SendData function; this will save the reference of the data buffer received as parameter in the internal tx buffer pointer, then copy the first byte to the data register. The hw tranceiver then automatically shifts the data and triggers a 'Transmit buffer empty' interrupt when all bits are shifted. The drivers interrupt handler takes care of transmitting the next byte in the buffer, by increasing the data pointer and decreasing the data size. The same sequence of operations is executed until all bytes in the tx buffer have been transmitted.

Similarly, data reception is triggered by calling LPUART_DRV_ReceiveData function, passing the rx buffer as parameter. When the tranceiver copies the received bits in the data register, the 'Receive buffer full' interrupt is triggered; the driver irq handler clears the flag by reading the received byte, saves it in the rx buffer, then increments the data pointer and decrements the data size. This is repeated untill all bytes are received.

The workflow applies to send/receive operations using blocking method (triggered by LPUART_DRV_SendData↩ Blocking and LPUART_DRV_ReceiveDataBlocking), with the single difference that the send/receive function will not return until the send/receive operation is complete (all bytes are successfully transferred or a timeout occured). The timeout for the blocking method is passed as parameter by the user.

If a user callback is installed for rx/tx, the callback has to take care of data handling and aborting the transfer when complete; the driver irq handler does not manipulate the buffers in this case. A target usecase here would be receiving an indefinite number of bytes; the user rx callback will be called by the driver each time a character is received and the application needs to call LPUART_DRV_AbortReceivingData in order to stop the reception.

DMA-based communication

In DMA operation, both blocking and non-blocking transmission methods confiure a DMA channel to copy data from the buffer to the data register (for tx), or viceversa (for rx). The driver assumes the DMA channel is already allocated and the proper requests are routed to it via DMAMUX. After configuring the DMA channel, the driver enables DMA

requests for rx/tx, then the DMA engine takes care of moving data to/from the data buffer. In this scenario, the callback is only called when the full transmission is done, that is when the DMA channel finishes the number of loops configured in the transfer descriptor.

Important Notes

- Before using the LPUART driver the module clock must be configured

- The driver enables the interrupts for the corresponding LPUART module, but any interrupt priority must be done by the application

- The board specific configurations must be done prior to driver calls; the driver has no influence on the functionality of the rx/tx pins - they must be configured by application

- DMA module has to be initialized prior to LPUART usage in DMA mode; also, DMA channels need to be allocated for LPUART usage by the application (the driver only takes care of configuring the DMA channels received in the configuration structure)

- for 9/10 bits characters, the application must provide the appropriate buffers; the size of the tx/rx buffers in this scenario needs to be an even number, as the transferred characters will be split in two bytes (bit 8 for 9-bits chars and bits 8 & 9 for 10-bits chars will be stored in the subsequent byte). 9/10 bits chars are only supported in interrupt-based and polling communications

Data Structures

- struct lpuart_state_t

    Runtime state of the LPUART driver. More...
- struct lpuart_user_config_t

    LPUART configuration structure. More...

Enumerations

- enum lpuart_transfer_type_t { LPUART_USING_DMA = 0, LPUART_USING_INTERRUPTS }

    Type of LPUART transfer (based on interrupts or DMA).
- enum lpuart_bit_count_per_char_t { LPUART_8_BITS_PER_CHAR = 0x0U, LPUART_9_BITS_PER_CHAR = 0x1U, LPUART_10_BITS_PER_CHAR = 0x2U }

    LPUART number of bits in a character.
- enum lpuart_parity_mode_t { LPUART_PARITY_DISABLED = 0x0U, LPUART_PARITY_EVEN = 0x2U, L↩PUART_PARITY_ODD = 0x3U }

    LPUART parity mode.
- enum lpuart_stop_bit_count_t { LPUART_ONE_STOP_BIT = 0x0U, LPUART_TWO_STOP_BIT = 0x1U }

    LPUART number of stop bits.

LPUART Driver

- status_t LPUART_DRV_Init (uint32_t instance, lpuart_state_t ∗lpuartStatePtr, const lpuart_user_config_↩t ∗lpuartUserConfig)

    Initializes an LPUART operation instance.
- status_t LPUART_DRV_Deinit (uint32_t instance)

    Shuts down the LPUART by disabling interrupts and transmitter/receiver.
- uart_callback_t   LPUART_DRV_InstallRxCallback   (uint32_t   instance,   uart_callback_t   function,   void ∗callbackParam)

    Installs callback function for the LPUART receive.
- uart_callback_t   LPUART_DRV_InstallTxCallback   (uint32_t   instance,   uart_callback_t   function,   void ∗callbackParam)

Installs callback function for the LPUART transmit.

- status_t LPUART_DRV_SendDataBlocking (uint32_t instance, const uint8_t ∗txBuff, uint32_t txSize, uint32←-
  _t timeout)

    Sends data out through the LPUART module using a blocking method.

- void LPUART_DRV_SendDataPolling (uint32_t instance, const uint8_t ∗txBuff, uint32_t txSize)

    Send out multiple bytes of data using polling method.

- status_t LPUART_DRV_SendData (uint32_t instance, const uint8_t ∗txBuff, uint32_t txSize)

    Sends data out through the LPUART module using a non-blocking method. This enables an a-sync method for transmitting data. When used with a non-blocking receive, the LPUART can perform a full duplex operation. Non-blocking means that the function returns immediately. The application has to get the transmit status to know when the transmit is complete.

- status_t LPUART_DRV_GetTransmitStatus (uint32_t instance, uint32_t ∗bytesRemaining)

    Returns whether the previous transmit is complete.

- status_t LPUART_DRV_AbortSendingData (uint32_t instance)

    Terminates a non-blocking transmission early.

- status_t LPUART_DRV_ReceiveDataBlocking (uint32_t instance, uint8_t ∗rxBuff, uint32_t rxSize, uint32_t timeout)

    Gets data from the LPUART module by using a blocking method. Blocking means that the function does not return until the receive is complete.

- status_t LPUART_DRV_ReceiveDataPolling (uint32_t instance, uint8_t ∗rxBuff, uint32_t rxSize)

    Receive multiple bytes of data using polling method.

- status_t LPUART_DRV_ReceiveData (uint32_t instance, uint8_t ∗rxBuff, uint32_t rxSize)

    Gets data from the LPUART module by using a non-blocking method. This enables an a-sync method for receiving data. When used with a non-blocking transmission, the LPUART can perform a full duplex operation. Non-blocking means that the function returns immediately. The application has to get the receive status to know when the receive is complete.

- status_t LPUART_DRV_GetReceiveStatus (uint32_t instance, uint32_t ∗bytesRemaining)

    Returns whether the previous receive is complete.

- status_t LPUART_DRV_AbortReceivingData (uint32_t instance)

    Terminates a non-blocking receive early.

- status_t LPUART_DRV_SetBaudRate (uint32_t instance, uint32_t desiredBaudRate)

    Configures the LPUART baud rate.

- void LPUART_DRV_GetBaudRate (uint32_t instance, uint32_t ∗configuredBaudRate)

    Returns the LPUART baud rate.

16.53.2 Data Structure Documentation

16.53.2.1 struct lpuart_state_t

Runtime state of the LPUART driver.

Note that the caller provides memory for the driver state structures during initialization because the driver does not statically allocate memory.

Implements : lpuart_state_t_Class

Definition at line 92 of file lpuart_driver.h.

Data Fields

- const uint8_t ∗ txBuff
- uint8_t ∗ rxBuff
- volatile uint32_t txSize
- volatile uint32_t rxSize
- volatile bool isTxBusy
- volatile bool isRxBusy

- volatile bool isTxBlocking
- volatile bool isRxBlocking
- lpuart_bit_count_per_char_t bitCountPerChar
- uart_callback_t rxCallback
- void ∗ rxCallbackParam
- uart_callback_t txCallback
- void ∗ txCallbackParam
- lpuart_transfer_type_t transferType
- semaphore_t rxComplete
- semaphore_t txComplete
- volatile status_t transmitStatus
- volatile status_t receiveStatus

Field Documentation

16.53.2.1.1   lpuart_bit_count_per_char_t bitCountPerChar

number of bits in a char (8/9/10)

Definition at line 102 of file lpuart_driver.h.

16.53.2.1.2   volatile bool isRxBlocking

True if receive is blocking transaction.

Definition at line 101 of file lpuart_driver.h.

16.53.2.1.3   volatile bool isRxBusy

True if there is an active receive.

Definition at line 99 of file lpuart_driver.h.

16.53.2.1.4   volatile bool isTxBlocking

True if transmit is blocking transaction.

Definition at line 100 of file lpuart_driver.h.

16.53.2.1.5   volatile bool isTxBusy

True if there is an active transmit.

Definition at line 98 of file lpuart_driver.h.

16.53.2.1.6   volatile status_t receiveStatus

Status of last driver receive operation

Definition at line 123 of file lpuart_driver.h.

16.53.2.1.7   uint8_t∗ rxBuff

The buffer of received data.

Definition at line 95 of file lpuart_driver.h.

16.53.2.1.8   uart_callback_t rxCallback

Callback to invoke for data receive Note: when the transmission is interrupt based, the callback is being called upon receiving a byte; when DMA transmission is used, the bytes are copied to the rx buffer by the DMA engine and the callback is called when all the bytes have been transferred.

Definition at line 103 of file lpuart_driver.h.

16.53.2.1.9   void∗ rxCallbackParam

Receive callback parameter pointer.

Definition at line 108 of file lpuart_driver.h.

16.53.2.1.10   semaphore_t rxComplete

Synchronization object for blocking Rx timeout condition

Definition at line 120 of file lpuart_driver.h.

16.53.2.1.11   volatile uint32_t rxSize

The remaining number of bytes to be received.

Definition at line 97 of file lpuart_driver.h.

16.53.2.1.12   lpuart_transfer_type_t transferType

Type of LPUART transfer (interrupt/dma based)

Definition at line 115 of file lpuart_driver.h.

16.53.2.1.13   volatile status_t transmitStatus

Status of last driver transmit operation

Definition at line 122 of file lpuart_driver.h.

16.53.2.1.14   const uint8_t∗ txBuff

The buffer of data being sent.

Definition at line 94 of file lpuart_driver.h.

16.53.2.1.15   uart_callback_t txCallback

Callback to invoke for data send Note: when the transmission is interrupt based, the callback is being called upon sending a byte; when DMA transmission is used, the bytes are copied to the tx buffer by the DMA engine and the callback is called when all the bytes have been transferred.

Definition at line 109 of file lpuart_driver.h.

16.53.2.1.16   void∗ txCallbackParam

Transmit callback parameter pointer.

Definition at line 114 of file lpuart_driver.h.

16.53.2.1.17   semaphore_t txComplete

Synchronization object for blocking Tx timeout condition

Definition at line 121 of file lpuart_driver.h.

16.53.2.1.18   volatile uint32_t txSize

The remaining number of bytes to be transmitted.

Definition at line 96 of file lpuart_driver.h.

16.53.2.2   struct lpuart_user_config_t

LPUART configuration structure.

Implements : lpuart_user_config_t_Class

Definition at line 130 of file lpuart_driver.h.

Data Fields

- uint32_t baudRate
- lpuart_parity_mode_t parityMode
- lpuart_stop_bit_count_t stopBitCount
- lpuart_bit_count_per_char_t bitCountPerChar
- lpuart_transfer_type_t transferType
- uint8_t rxDMAChannel
- uint8_t txDMAChannel

Field Documentation

16.53.2.2.1   uint32_t baudRate

LPUART baud rate

Definition at line 132 of file lpuart_driver.h.

16.53.2.2.2   lpuart_bit_count_per_char_t bitCountPerChar

number of bits in a character (8-default, 9 or 10); for 9/10 bits chars, users must provide appropriate buffers to the send/receive functions (bits 8/9 in subsequent bytes); for DMA transmission only 8-bit char is supported.

Definition at line 135 of file lpuart_driver.h.

16.53.2.2.3   lpuart_parity_mode_t parityMode

parity mode, disabled (default), even, odd

Definition at line 133 of file lpuart_driver.h.

16.53.2.2.4   uint8_t rxDMAChannel

Channel number for DMA rx channel. If DMA mode isn't used this field will be ignored.

Definition at line 140 of file lpuart_driver.h.

16.53.2.2.5   lpuart_stop_bit_count_t stopBitCount

number of stop bits, 1 stop bit (default) or 2 stop bits

Definition at line 134 of file lpuart_driver.h.

16.53.2.2.6   lpuart_transfer_type_t transferType

Type of LPUART transfer (interrupt/dma based)

Definition at line 139 of file lpuart_driver.h.

16.53.2.2.7   uint8_t txDMAChannel

Channel number for DMA tx channel. If DMA mode isn't used this field will be ignored.

Definition at line 142 of file lpuart_driver.h.

16.53.3   Enumeration Type Documentation

16.53.3.1   enum lpuart_bit_count_per_char_t

LPUART number of bits in a character.

Implements : lpuart_bit_count_per_char_t_Class

Enumerator

> ***LPUART_8_BITS_PER_CHAR***   8-bit data characters
> ***LPUART_9_BITS_PER_CHAR***   9-bit data characters
> ***LPUART_10_BITS_PER_CHAR***   10-bit data characters

Definition at line 56 of file lpuart_driver.h.

16.53.3.2 enum lpuart_parity_mode_t

LPUART parity mode.

Implements : lpuart_parity_mode_t_Class

Enumerator

> ***LPUART_PARITY_DISABLED***   parity disabled
> ***LPUART_PARITY_EVEN***   parity enabled, type even, bit setting: PE|PT = 10
> ***LPUART_PARITY_ODD***   parity enabled, type odd, bit setting: PE|PT = 11

Definition at line 67 of file lpuart_driver.h.

16.53.3.3 enum lpuart_stop_bit_count_t

LPUART number of stop bits.

Implements : lpuart_stop_bit_count_t_Class

Enumerator

> ***LPUART_ONE_STOP_BIT***   one stop bit
> ***LPUART_TWO_STOP_BIT***   two stop bits

Definition at line 78 of file lpuart_driver.h.

16.53.3.4 enum lpuart_transfer_type_t

Type of LPUART transfer (based on interrupts or DMA).

Implements : lpuart_transfer_type_t_Class

Enumerator

> ***LPUART_USING_DMA***   The driver will use DMA to perform UART transfer
> ***LPUART_USING_INTERRUPTS***   The driver will use interrupts to perform UART transfer

Definition at line 46 of file lpuart_driver.h.

16.53.4 Function Documentation

16.53.4.1 status_t LPUART_DRV_AbortReceivingData (uint32_t *instance*)

Terminates a non-blocking receive early.

Parameters

instance	LPUART instance number

Returns

> Whether the receiving was successful or not.

Definition at line 780 of file lpuart_driver.c.

16.53.4.2   status_t LPUART_DRV_AbortSendingData (uint32_t *instance*)

Terminates a non-blocking transmission early.

Parameters

instance	LPUART instance number

Returns

Whether the aborting is successful or not.

Definition at line 530 of file lpuart_driver.c.

16.53.4.3   status_t LPUART_DRV_Deinit (uint32_t *instance*)

Shuts down the LPUART by disabling interrupts and transmitter/receiver.

Parameters

instance	LPUART instance number

Returns

STATUS_SUCCESS if successful; STATUS_ERROR if an error occurred

Definition at line 244 of file lpuart_driver.c.

16.53.4.4   void LPUART_DRV_GetBaudRate (uint32_t *instance,* uint32_t ∗ *configuredBaudRate*)

Returns the LPUART baud rate.

This function returns the LPUART configured baud rate.

Parameters

	instance	LPUART instance number.
out	*configured↩ BaudRate*	LPUART configured baud rate.

Definition at line 903 of file lpuart_driver.c.

16.53.4.5   status_t LPUART_DRV_GetReceiveStatus (uint32_t *instance,* uint32_t ∗ *bytesRemaining*)

Returns whether the previous receive is complete.

Parameters

instance	LPUART instance number
bytesRemaining	pointer to value that is filled with the number of bytes that still need to be received in the active transfer.

Returns

The receive status.

Return values

STATUS_SUCCESS	the receive has completed successfully.
STATUS_BUSY	the receive is still in progress. *bytesReceived* will be filled with the number of bytes that have been received so far.

STATUS_UART_ABORT↩ ED	The receive was aborted.
STATUS_TIMEOUT	A timeout was reached.
STATUS_ERROR	An error occurred.

Definition at line 737 of file lpuart_driver.c.

16.53.4.6   status_t LPUART_DRV_GetTransmitStatus (uint32_t *instance,* uint32_t ∗ *bytesRemaining*)

Returns whether the previous transmit is complete.

Parameters

instance	LPUART instance number
bytesRemaining	Pointer to value that is populated with the number of bytes that have been sent in the active transfer

Returns

> The transmit status.

Return values

STATUS_SUCCESS	The transmit has completed successfully.
STATUS_BUSY	The transmit is still in progress. *bytesTransmitted* will be filled with the number of bytes that have been transmitted so far.
STATUS_UART_ABORT↩ ED	The transmit was aborted.
STATUS_TIMEOUT	A timeout was reached.
STATUS_ERROR	An error occurred.

Definition at line 486 of file lpuart_driver.c.

16.53.4.7   status_t LPUART_DRV_Init (uint32_t *instance,* lpuart_state_t ∗ *lpuartStatePtr,* const lpuart_user_config_t ∗ *lpuartUserConfig*)

Initializes an LPUART operation instance.

The caller provides memory for the driver state structures during initialization. The user must select the LPUART clock source in the application to initialize the LPUART.

Parameters

instance	LPUART instance number
lpuartUserConfig	user configuration structure of type lpuart_user_config_t
lpuartStatePtr	pointer to the LPUART driver state structure

Returns

> STATUS_SUCCESS if successful; STATUS_ERROR if an error occurred

Definition at line 158 of file lpuart_driver.c.

16.53.4.8   uart_callback_t LPUART_DRV_InstallRxCallback (uint32_t *instance,* uart_callback_t *function,* void ∗ *callbackParam*)

Installs callback function for the LPUART receive.

Note

> After a callback is installed, it bypasses part of the LPUART IRQHandler logic. Therefore, the callback needs to handle the indexes of txBuff and txSize.

Parameters

instance	The LPUART instance number.
function	The LPUART receive callback function.
rxBuff	The receive buffer used inside IRQHandler. This buffer must be kept as long as the callback is alive.
callbackParam	The LPUART receive callback parameter pointer.

Returns

>   Former LPUART receive callback function pointer.

Definition at line 289 of file lpuart_driver.c.

16.53.4.9   uart_callback_t LPUART_DRV_InstallTxCallback (uint32_t *instance,* uart_callback_t *function,* void ∗ *callbackParam*)

Installs callback function for the LPUART transmit.

Note

>   After a callback is installed, it bypasses part of the LPUART IRQHandler logic. Therefore, the callback needs to handle the indexes of txBuff and txSize.

Parameters

instance	The LPUART instance number.
function	The LPUART transmit callback function.
txBuff	The transmit buffer used inside IRQHandler. This buffer must be kept as long as the callback is alive.
callbackParam	The LPUART transmit callback parameter pointer.

Returns

>   Former LPUART transmit callback function pointer.

Definition at line 312 of file lpuart_driver.c.

16.53.4.10   status_t LPUART_DRV_ReceiveData (uint32_t *instance,* uint8_t ∗ *rxBuff,* uint32_t *rxSize*)

Gets data from the LPUART module by using a non-blocking method. This enables an a-sync method for receiving data. When used with a non-blocking transmission, the LPUART can perform a full duplex operation. Non-blocking means that the function returns immediately. The application has to get the receive status to know when the receive is complete.

Parameters

instance	LPUART instance number
rxBuff	buffer containing 8-bit read data chars received
rxSize	the number of bytes to receive

Returns

>   STATUS_SUCCESS if successful; STATUS_BUSY if a resource is busy; STATUS_ERROR if an error occurred

Definition at line 694 of file lpuart_driver.c.

16.53.4.11   status_t LPUART_DRV_ReceiveDataBlocking (uint32_t *instance,* uint8_t ∗ *rxBuff,* uint32_t *rxSize,* uint32_t *timeout*)

Gets data from the LPUART module by using a blocking method. Blocking means that the function does not return until the receive is complete.

Parameters

instance	LPUART instance number
rxBuff	buffer containing 8-bit read data chars received
rxSize	the number of bytes to receive
timeout	timeout value in milliseconds

Returns

STATUS_SUCCESS if successful; STATUS_TIMEOUT if the timeout was reached; STATUS_BUSY if a resource is busy; STATUS_ERROR if an error occurred

Definition at line 567 of file lpuart_driver.c.

16.53.4.12   status_t LPUART_DRV_ReceiveDataPolling (uint32_t *instance,* uint8_t ∗ *rxBuff,* uint32_t *rxSize*)

Receive multiple bytes of data using polling method.

Parameters

instance	LPUART instance number.
rxBuff	The buffer pointer which saves the data to be received.
rxSize	Size of data need to be received in unit of byte.

Returns

STATUS_SUCCESS if the transaction is success or STATUS_UART_RX_OVERRUN if rx overrun.

Definition at line 633 of file lpuart_driver.c.

16.53.4.13   status_t LPUART_DRV_SendData (uint32_t *instance,* const uint8_t ∗ *txBuff,* uint32_t *txSize*)

Sends data out through the LPUART module using a non-blocking method. This enables an a-sync method for transmitting data. When used with a non-blocking receive, the LPUART can perform a full duplex operation. Non-blocking means that the function returns immediately. The application has to get the transmit status to know when the transmit is complete.

Parameters

instance	LPUART instance number
txBuff	source buffer containing 8-bit data chars to send
txSize	the number of bytes to send

Returns

STATUS_SUCCESS if successful; STATUS_BUSY if a resource is busy; STATUS_ERROR if an error occurred

Definition at line 442 of file lpuart_driver.c.

16.53.4.14   status_t LPUART_DRV_SendDataBlocking (uint32_t *instance,* const uint8_t ∗ *txBuff,* uint32_t *txSize,* uint32_t
**   *timeout*)**

Sends data out through the LPUART module using a blocking method.

Blocking means that the function does not return until the transmission is complete.

Parameters

instance	LPUART instance number
txBuff	source buffer containing 8-bit data chars to send
txSize	the number of bytes to send
timeout	timeout value in milliseconds

Returns

> STATUS_SUCCESS if successful; STATUS_TIMEOUT if the timeout was reached; STATUS_BUSY if a resource is busy; STATUS_ERROR if an error occurred

Definition at line 335 of file lpuart_driver.c.

16.53.4.15   void LPUART_DRV_SendDataPolling (uint32_t *instance,* const uint8_t ∗ *txBuff,* uint32_t *txSize*)

Send out multiple bytes of data using polling method.

Parameters

instance	LPUART instance number.
txBuff	The buffer pointer which saves the data to be sent.
txSize	Size of data to be sent in unit of byte.

Definition at line 401 of file lpuart_driver.c.

16.53.4.16   status_t LPUART_DRV_SetBaudRate (uint32_t *instance,* uint32_t *desiredBaudRate*)

Configures the LPUART baud rate.

This function configures the LPUART baud rate. In some LPUART instances the user must disable the transmitter/receiver before calling this function. Generally, this may be applied to all LPUARTs to ensure safe operation.

Parameters

instance	LPUART instance number.
desiredBaud↩ *Rate*	LPUART desired baud rate.

Returns

> STATUS_SUCCESS

Definition at line 819 of file lpuart_driver.c.

16.54   Local Interconnect Network (LIN)

16.54.1   Detailed Description

The S32 SDK provides both driver and middleware layers for the Local Interconnect Network (LIN) protocol, emulated on top of LPUART serial communication IP.

Modules

- LIN Driver

    This section describes the programming interface of the Peripheral driver for LIN.

- LIN Stack

    This section covers the functionality of the LIN Stack middleware layer in S32 SDK.

16.55 Low Power Inter-Integrated Circuit (LPI2C)

16.55.1 Detailed Description

The LPI2C is a low power Inter-Integrated Circuit (I2C) module that supports an efficient interface to an I2C bus as a master and/or a slave.

Modules

- LPI2C Driver

    Low Power Inter-Integrated Circuit (LPI2C) Peripheral Driver.

16.56   Low Power Interrupt Timer (LPIT)

16.56.1   Detailed Description

The Low Power Periodic Interrupt Timer (LPIT) is a multi-channel timer module generating independent pre-trigger and trigger outputs. These timer channels can operate individually or can be chained together. The LPIT can operate in low power modes if configured to do so. The pre-trigger and trigger outputs can be used to trigger other modules on the device.

The S32 SDK provides Peripheral Drivers for the Low Power Interrupt Timer (LPIT) module of S32 devices.

Modules

- LPIT Driver

  Low Power Interrupt Timer Peripheral Driver.

16.57 Low Power Serial Peripheral Interface (LPSPI)

16.57.1 Detailed Description

Low Power Serial Peripheral Interface (LPSPI) Peripheral Driver.

The LPSPI driver allows communication on an SPI bus using the LPSPI module in the S32144K processor.

Features

- Interrupt based

- Master or slave operation

- Provides blocking and non-blocking transmit and receive functions

- RX and TX hardware buffers (4 words)

- 4 configurable chip select

- Configurable baud rate

How to integrate LPSPI in your application

In order to use the LPSPI driver it must be first initialized in either master or slave mode, using functions LPSP←I_DRV_MasterInit() or LPSPI_DRV_SlaveInit(). Once initialized, it cannot be initialized again for the same LPSPI module instance until it is de-initialized, using LPSPI_DRV_MasterDeinit() or LPSPI_DRV_SlaveDeinit(). Different LPSPI module instances can function independently of each other.

In each mode (master/slave) are available two types of transfers: blocking and non-blocking. The functions which initiate blocking transfers will configure the time out for transmission. If time expires LPSPI_MasterTransferBlocking/←LPSPI_SlaveTransferBlocking will return error and the transmission will be aborted.

Important Notes

- The driver enables the interrupts for the corresponding LPSPI module, but any interrupt priority setting must be done by the application.

- The watermarks will be set by the application.

- The driver will configure SCK to PCS delay, PCS to SCK delay, delay between transfers with default values. If you application needs other values for this parameters LPSPI_DRV_MasterSetDelay function can be uesd.

Example code

```
const lpspi_master_config_t Send_MasterConfig0 = {
    .bitsPerSec = 50000U,
    .whichPcs = LPSPI_PCS0,
    .pcsPolarity = LPSPI_ACTIVE_HIGH,
    .isPcsContinuous = false,
    .bitcount = 8U,
    .lpspiSrcClk = 8000000U,
    .clkPhase = LPSPI_CLOCK_PHASE_1ST_EDGE,
    .clkPolarity = LPSPI_SCK_ACTIVE_HIGH,
    .lsbFirst = false,
    .transferType = LPSPI_USING_INTERRUPTS,
};
const lpspi_slave_config_t Receive_SlaveConfig0 = {
    .pcsPolarity = LPSPI_ACTIVE_HIGH,
    .bitcount = 8U,
    .clkPhase = LPSPI_CLOCK_PHASE_1ST_EDGE,
    .whichPcs = LPSPI_PCS0,
    .clkPolarity = LPSPI_SCK_ACTIVE_HIGH,
    .lsbFirst = false,
    .transferType = LPSPI_USING_INTERRUPTS,
};
/* Initialize clock and pins */
LPSPI_DRV_MasterInit(0U, &masterState, &Send_MasterConfig0);
/* Set delay between transfer, PCStoSCK and SCKtoPCS to 10 microseconds. */
LPSPI_DRV_MasterSetDelay(0U, 10U, 10U, 10u);
/* Initialize LPSPI1 (Slave)*/
```

```
LPSPI_DRV_SlaveInit(1U, &slaveState, &Receive_SlaveConfig0);
/* Allocate memory */
masterDataSend = (uint8_t*)calloc(100, sizeof(uint8_t));
masterDataReceive = (uint8_t*)calloc(100, sizeof(uint8_t));
slaveDataSend = (uint8_t*)calloc(100, sizeof(uint8_t));
slaveDataReceive = (uint8_t*)calloc(100, sizeof(uint8_t));
bufferSize = 100U;
testStatus[0] = true;
LPSPI_DRV_SlaveTransfer(0U, slaveDataSend,
            slaveDataReceive, bufferSize);
LPSPI_DRV_MasterTransferBlocking(1U, &Send_MasterConfig0, masterDataSend,
            masterDataReceive, bufferSize, TIMEOUT);
```

Modules

- LPSPI Driver

     Low Power Serial Peripheral Interface Peripheral Driver.

16.58 Low Power Timer (LPTMR)

16.58.1 Detailed Description

The S32 SDK provides Peripheral Drivers for the Low Power Timer (LPTMR) module of S32 SDK devices.

The LPTMR is a configurable 16-bit counter that can operate in two functional modes:

- Timer mode with selectable prescaler and clock source (periodic or free-running).

- Pulse-Counter mode, with configurable glitch filter, that can count events (internal or external)

Modules

- LPTMR Driver

  Low Power Timer Peripheral Driver.

16.59   Low Power Universal Asynchronous Receiver-Transmitter (LPUART)

16.59.1   Detailed Description

The S32 SDK provides a Peripheral Driver for the Low Power Universal Asynchronous Receiver-Transmitter (LP↩
UART) module of S32 SDK devices.
The LPUART module is used for serial communication, supporting LIN master and slave operation. These sections
describe the S32 SDK software modules API that can be used for initializing and configuring the module, as well as
initiating serial communications using the interrupt-based method.

Modules

- LPUART Driver

  This module covers the functionality of the Low Power Universal Asynchronous Receiver-Transmitter (LPUART) pe-
  ripheral driver.

16.60 Low level API

16.60.1 Detailed Description

Low level layer consists of functions that call LIN driver API.

This layer contains the implementation of LIN hardware initialization and deinitialization, getting LIN node's current state, sending wakeup signals, enabling and disabling interrupts, sending frame data from a buffer, receiving frame data into a buffer, handling timeout and callbacks from LIN driver.

Data Structures

- struct lin_word_status_str_t

  status of LIN bus Implements : lin_word_status_str_t_Class More...

- struct lin_serial_number_t

  Serial number Implements : lin_serial_number_t_Class. More...

- struct lin_node_attribute_t

  Attributes of LIN node Implements : lin_node_attribute_t_Class. More...

- struct lin_associate_frame_t

  Informations of associated frame Implements : lin_associate_frame_t_Class. More...

- struct lin_frame_t

  Frame description structure Implements : lin_frame_t_Class. More...

- struct lin_schedule_data_t

  LIN schedule structure Implements : lin_schedule_data_t_Class. More...

- struct lin_schedule_t

  Schedule table description Implements : lin_schedule_t_Class. More...

- struct lin_transport_layer_queue_t

  Transport layer queue Implements : lin_transport_layer_queue_t_Class. More...

- struct lin_tl_descriptor_t

  Transport layer description Implements : lin_tl_descriptor_t_Class. More...

- struct lin_protocol_user_config_t

  Configuration structure Implements : lin_protocol_user_config_t_Class. More...

- struct lin_master_data_t

  LIN master configuration structure Implements : lin_master_data_t_Class. More...

- struct lin_protocol_state_t

  LIN protocol status structure Implements : lin_protocol_state_t_Class. More...

Macros

- #define SERVICE_ASSIGN_NAD 0xB0U
- #define SERVICE_ASSIGN_FRAME_ID 0xB1U
- #define SERVICE_READ_BY_IDENTIFY 0xB2U
- #define SERVICE_CONDITIONAL_CHANGE_NAD 0xB3U
- #define SERVICE_SAVE_CONFIGURATION 0xB6U
- #define SERVICE_ASSIGN_FRAME_ID_RANGE 0xB7U
- #define SERVICE_READ_DATA_BY_IDENTIFY 0x22U
- #define SERVICE_WRITE_DATA_BY_IDENTIFY 0x2EU
- #define SERVICE_SESSION_CONTROL 0x10U
- #define SERVICE_IO_CONTROL_BY_IDENTIFY 0x2FU
- #define SERVICE_FAULT_MEMORY_READ 0x19U
- #define SERIVCE_FAULT_MEMORY_CLEAR 0x14U
- #define PCI_SAVE_CONFIGURATION 0x01U
- #define PCI_RES_READ_BY_IDENTIFY 0x06U

- #define PCI_RES_SAVE_CONFIGURATION 0x01U
- #define PCI_RES_ASSIGN_FRAME_ID_RANGE 0x01U
- #define LIN_READ_USR_DEF_MIN 32U
- #define LIN_READ_USR_DEF_MAX 63U
- #define LD_ID_NO_RESPONSE 0x52U
- #define LD_NEGATIVE_RESPONSE 0x53U
- #define LD_POSITIVE_RESPONSE 0x54U
- #define LIN_LLD_OK 0x00U
- #define LIN_LLD_ERROR 0xFFU
- #define LIN_SLAVE 0

    Mode of LIN node (master or slave)
- #define LIN_MASTER 1
- #define LIN_TL_CALLBACK_HANDLER(iii, tl_event_id, id) lin_tl_callback_handler((iii), (tl_event_id), (id))
- #define INTERLEAVE_MAX_TIMEOUT (l_u16)(1000000U/TIME_OUT_UNIT_US)
- #define CALLBACK_HANDLER(iii, event_id, id) lin_pid_resp_callback_handler((iii), (event_id), (id))

    CALLBACK_HANDLER.

Typedefs

- typedef l_u8 lin_tl_pdu_data_t[8]

    PDU data. Implements : lin_tl_pdu_data_t_Class.
- typedef l_u8 lin_tl_queue_t[8]

    LIN transport layer queue Implements : lin_tl_queue_t_Class.

Enumerations

- enum lin_lld_event_id_t {
  LIN_LLD_PID_OK = 0x00U, LIN_LLD_TX_COMPLETED = 0x01U, LIN_LLD_RX_COMPLETED = 0x02U,
  LIN_LLD_PID_ERR = 0x03U,
  LIN_LLD_FRAME_ERR = 0x04U, LIN_LLD_CHECKSUM_ERR = 0x05U, LIN_LLD_READBACK_ERR =
  0x06U, LIN_LLD_NODATA_TIMEOUT = 0x07U,
  LIN_LLD_BUS_ACTIVITY_TIMEOUT = 0x08U }

    Event id Implements : lin_lld_event_id_t_Class.
- enum lin_protocol_handle_t { LIN_PROTOCOL_21 = 0x00U, LIN_PROTOCOL_J2602 = 0x01U }

    List of protocols Implements : lin_protocol_handle_t_Class.
- enum lin_diagnostic_class_t { LIN_DIAGNOSTIC_CLASS_I = 0x01U, LIN_DIAGNOSTIC_CLASS_II =
  0x02U, LIN_DIAGNOSTIC_CLASS_III = 0x03U }

    List of diagnostic classes Implements : lin_diagnostic_class_t_Class.
- enum lin_frame_type_t { LIN_FRM_UNCD = 0x00U, LIN_FRM_EVNT = 0x01U, LIN_FRM_SPRDC = 0x10U,
  LIN_FRM_DIAG = 0x11U }

    Types of frame Implements : lin_frame_type_t_Class.
- enum lin_frame_response_t { LIN_RES_PUB = 0x00U, LIN_RES_SUB = 0x01U }

    LIN frame response Implements : lin_frame_response_t_Class.
- enum lin_sch_tbl_type_t {
  LIN_SCH_TBL_NULL = 0x00U, LIN_SCH_TBL_NORM = 0x01U, LIN_SCH_TBL_DIAG = 0x02U, LIN_SC←
  H_TBL_GO_TO_SLEEP = 0x03U,
  LIN_SCH_TBL_COLL_RESOLV = 0x04U }

    Types of schedule tables Implements : lin_sch_tbl_type_t_Class.
- enum l_diagnostic_mode_t { DIAG_NONE = 0x00U, DIAG_INTERLEAVE_MODE = 0x01U, DIAG_ONLY_←
  MODE = 0x02U }

    Diagnostic mode Implements : l_diagnostic_mode_t_Class.
- enum lin_service_status_t { LD_SERVICE_BUSY = 0x00U, LD_REQUEST_FINISHED = 0x01U, LD_SER←
  VICE_IDLE = 0x02U, LD_SERVICE_ERROR = 0x03U }

Status of the last configuration call for LIN 2.1 Implements : lin_service_status_t_Class.

- enum lin_last_cfg_result_t { LD_SUCCESS = 0x00U, LD_NEGATIVE = 0x01U, LD_NO_RESPONSE = 0x02U, LD_OVERWRITTEN = 0x03U }

  Status of the last configuration call completed Implements : lin_last_cfg_result_t_Class.

- enum lin_tl_event_id_t {
  TL_MAKE_RES_DATA = 0x00U, TL_SLAVE_GET_ACTION = 0x01U, TL_TX_COMPLETED = 0x02U, TL↩
  _RX_COMPLETED = 0x03U,
  TL_ERROR = 0x04U, TL_TIMEOUT_SERVICE = 0x05U, TL_HANDLER_INTERLEAVE_MODE = 0x06U,
  TL_RECEIVE_MESSAGE = 0x07U }

  Transport layer event IDs Implements : lin_tl_event_id_t_Class.

- enum lin_tl_callback_return_t { TL_ACTION_NONE = 0x00U, TL_ACTION_ID_IGNORE = 0x01U }

  Transport layer event IDs Implements : lin_tl_callback_return_t_Class.

- enum ld_queue_status_t {
  LD_NO_DATA = 0x00U, LD_DATA_AVAILABLE = 0x01U, LD_RECEIVE_ERROR = 0x02U, LD_QUEUE_↩
  FULL = 0x03U,
  LD_QUEUE_AVAILABLE = 0x04U, LD_QUEUE_EMPTY = 0x05U, LD_TRANSMIT_ERROR = 0x06U, LD↩
  _TRANSFER_ERROR = 0x07U }

  Status of queue Implements : ld_queue_status_t_Class.

- enum lin_message_status_t {
  LD_NO_MSG = 0x00U, LD_IN_PROGRESS = 0x01U, LD_COMPLETED = 0x02U, LD_FAILED = 0x03U,
  LD_N_AS_TIMEOUT = 0x04U, LD_N_CR_TIMEOUT = 0x05U, LD_WRONG_SN = 0x06U }

  Status of LIN message Implements : lin_message_status_t_Class.

- enum lin_diagnostic_state_t {
  LD_DIAG_IDLE = 0x01U, LD_DIAG_TX_PHY = 0x02U, LD_DIAG_TX_FUNCTIONAL = 0x03U, LD_DIAG↩
  _TX_INTERLEAVED = 0x04U,
  LD_DIAG_RX_PHY = 0x05U, LD_DIAG_RX_FUNCTIONAL = 0x06U, LD_DIAG_RX_INTERLEAVED = 0x07U }

  LIN diagnostic state Implements : lin_diagnostic_state_t_Class.

- enum lin_message_timeout_type_t { LD_NO_CHECK_TIMEOUT = 0x00U, LD_CHECK_N_AS_TIMEOUT = 0x01U, LD_CHECK_N_CR_TIMEOUT = 0x02U }

  Types of message timeout Implements : lin_message_timeout_type_t_Class.

- enum diag_interleaved_state_t { DIAG_NOT_START = 0x00U, DIAG_NO_RESPONSE = 0x01U, DIAG_R↩
  ESPONSE = 0x02U }

  State of diagnostic interleaved mode Implements : diag_interleaved_state_t_Class.

Functions

- lin_tl_callback_return_t lin_tl_callback_handler (l_ifc_handle iii, lin_tl_event_id_t tl_event_id, l_u8 id)
- l_u8 ld_read_by_id_callout (l_ifc_handle iii, l_u8 id, l_u8 ∗data)
- static l_u16 lin_calc_max_header_timeout_cnt (l_u32 baudRate)

  Computes maximum header timeout.

- static l_u16 lin_calc_max_res_timeout_cnt (l_u32 baudRate, l_u8 size)

  Computes the maximum response timeout.

- l_u8 lin_process_parity (l_u8 pid, l_u8 typeAction)

  Makes or checks parity bits. If action is checking parity, the function returns ID value if parity bits are correct or 0xFF if parity bits are incorrect. If action is making parity bits, then from input value of ID, the function returns PID.

- void lin_pid_resp_callback_handler (l_ifc_handle iii, const lin_lld_event_id_t event_id, l_u8 id)

  Callback handler for low level events.

- l_bool lin_lld_init (l_ifc_handle iii)

  This function initializes a LIN hardware instance for operation. This function will initialize the run-time state structure to keep track of the on-going transfers, initialize the module to user defined settings and default settings, configure the IRQ state structure and enable the module-level interrupt to the core, and enable the LIN hardware module transmitter and receiver.

- l_u8 lin_lld_deinit (l_ifc_handle iii)

> *This function disconnect the node from the cluster and free all hardware used.*

- l_u8 lin_lld_int_enable (l_ifc_handle iii)

    Enable the interrupt related to the interface.

- l_u8 lin_lld_int_disable (l_ifc_handle iii)

    Disable the interrupt related to the interface.

- l_u8 lin_lld_get_state (l_ifc_handle iii)

    This function gets current state of an interface.

- l_u8 lin_lld_tx_header (l_ifc_handle iii, l_u8 id)

    This function sends frame header for the input PID.

- l_u8 lin_lld_tx_wake_up (l_ifc_handle iii)

    This function send a wakeup signal.

- l_u8 lin_lld_ignore_response (l_ifc_handle iii)

    This function terminates an on-going data transmission/reception.

- l_u8 lin_lld_set_low_power_mode (l_ifc_handle iii)

    Let the low level driver go to low power mode.

- l_u8 lin_lld_set_response (l_ifc_handle iii, l_u8 response_length)

    This function sends frame data that is contained in LIN_lld_response_buffer[iii].

- l_u8 lin_lld_rx_response (l_ifc_handle iii, l_u8 response_length)

    This function receives frame data into the LIN_lld_response_buffer[iii] buffer.

- void lin_lld_timeout_service (l_ifc_handle iii)

    Callback function for Timer Interrupt Handler In timer IRQ handler, call this function. Used to check if frame timeout has occurred during frame data transmission and reception, to check for N_As and N_Cr timeout for LIN 2.1 and above. This function also check if there is no LIN bus communication (no headers and no frame data transferring) for Idle timeout (s), then put LIN node to Sleep mode. Users may initialize a timer (for example FTM)with period of Timeout unit (default: 500 micro seconds) to call lin_lld_timeout_service(). For an interface iii, Idle timeout (s) = max_idle_timeout_cnt ∗ Timeout unit (us) frame timeout (us) = frame_timeout_cnt ∗ Timeout unit (us) N_As timeout (us) = N_As_timeout ∗ Timeout unit (us) N_Cr timeout (us) = N_Cr_timeout ∗ Timeout unit (us)

Variables

- const lin_node_attribute_t g_lin_node_attribute_array [LIN_NUM_OF_SLAVE_IFCS]
- lin_master_data_t g_lin_master_data_array [LIN_NUM_OF_MASTER_IFCS]
- lin_tl_descriptor_t g_lin_tl_descriptor_array [LIN_NUM_OF_IFCS]
- const lin_protocol_user_config_t g_lin_protocol_user_cfg_array [LIN_NUM_OF_IFCS]
- lin_protocol_state_t g_lin_protocol_state_array [LIN_NUM_OF_IFCS]
- l_u8 g_lin_frame_data_buffer [LIN_FRAME_BUF_SIZE]
- l_u8 g_lin_flag_handle_tbl [LIN_FLAG_BUF_SIZE]
- l_bool g_lin_frame_flag_handle_tbl [LIN_NUM_OF_FRMS]
- const l_u32 g_lin_virtual_ifc [LIN_NUM_OF_IFCS]
- const l_ifc_handle g_lin_hardware_ifc [HARDWARE_INSTANCE_COUNT]
- const lin_timer_get_time_interval_t timerGetTimeIntervalCallbackArr [LIN_NUM_OF_IFCS]
- volatile l_u8 g_buffer_backup_data [8]
- volatile l_u8 g_lin_frame_updating_flag_tbl [LIN_NUM_OF_FRMS]

16.60.2   Data Structure Documentation

16.60.2.1   struct lin_word_status_str_t

status of LIN bus Implements : lin_word_status_str_t_Class

Definition at line 150 of file lin.h.

Data Fields

- unsigned int error_in_res: 1
- unsigned int successful_transfer: 1
- unsigned int overrun: 1
- unsigned int go_to_sleep_flg: 1
- unsigned int bus_activity: 1
- unsigned int event_trigger_collision_flg: 1
- unsigned int save_config_flg: 1
- unsigned int reserved: 1
- unsigned int last_pid: 8

Field Documentation

16.60.2.1.1   unsigned int bus_activity

Bus activity

Definition at line 156 of file lin.h.

16.60.2.1.2   unsigned int error_in_res

Error in response

Definition at line 152 of file lin.h.

16.60.2.1.3   unsigned int event_trigger_collision_flg

Event trigger collision

Definition at line 157 of file lin.h.

16.60.2.1.4   unsigned int go_to_sleep_flg

Goto sleep

Definition at line 155 of file lin.h.

16.60.2.1.5   unsigned int last_pid

Last PID

Definition at line 160 of file lin.h.

16.60.2.1.6   unsigned int overrun

Overrun

Definition at line 154 of file lin.h.

16.60.2.1.7   unsigned int reserved

Dummy

Definition at line 159 of file lin.h.

16.60.2.1.8   unsigned int save_config_flg

Save configuration

Definition at line 158 of file lin.h.

16.60.2.1.9   unsigned int successful_transfer

Successful transfer

Definition at line 153 of file lin.h.

16.60.2.2   struct lin_serial_number_t

Serial number Implements : lin_serial_number_t_Class.

Definition at line 177 of file lin.h.

Data Fields

- l_u8 serial_0
- l_u8 serial_1
- l_u8 serial_2
- l_u8 serial_3

Field Documentation

16.60.2.2.1   l_u8 serial_0

Serial 0

Definition at line 179 of file lin.h.

16.60.2.2.2   l_u8 serial_1

Serial 1

Definition at line 180 of file lin.h.

16.60.2.2.3   l_u8 serial_2

Serial 2

Definition at line 181 of file lin.h.

16.60.2.2.4   l_u8 serial_3

Serial 3

Definition at line 182 of file lin.h.

16.60.2.3   struct lin_node_attribute_t

Attributes of LIN node Implements : lin_node_attribute_t_Class.

Definition at line 189 of file lin.h.

Data Fields

- l_u8 ∗ configured_NAD_ptr
- l_u8 initial_NAD
- lin_product_id_t product_id
- lin_serial_number_t serial_number
- l_u8 ∗ resp_err_frm_id_ptr
- l_u8 num_frame_have_esignal
- l_signal_handle response_error
- l_u8 ∗ response_error_byte_offset_ptr
- l_u8 ∗ response_error_bit_offset_ptr
- l_u8 num_of_fault_state_signal
- const l_signal_handle ∗ fault_state_signal_ptr
- l_u16 P2_min
- l_u16 ST_min
- l_u16 N_As_timeout

- l_u16 N_Cr_timeout
- l_u8 number_support_sid
- const l_u8 ∗ service_supported_ptr
- l_u8 ∗ service_flags_ptr

Field Documentation

16.60.2.3.1 l_u8∗ configured_NAD_ptr

NAD value used in configuration command

Definition at line 191 of file lin.h.

16.60.2.3.2 const l_signal_handle∗ fault_state_signal_ptr

List of fault state signal

Definition at line 201 of file lin.h.

16.60.2.3.3 l_u8 initial_NAD

Initial NAD

Definition at line 192 of file lin.h.

16.60.2.3.4 l_u16 N_As_timeout

N_As_timeout

Definition at line 204 of file lin.h.

16.60.2.3.5 l_u16 N_Cr_timeout

N_Cr_timeout

Definition at line 205 of file lin.h.

16.60.2.3.6 l_u8 num_frame_have_esignal

Number of frame contain error signal

Definition at line 196 of file lin.h.

16.60.2.3.7 l_u8 num_of_fault_state_signal

Number of Fault state signal

Definition at line 200 of file lin.h.

16.60.2.3.8 l_u8 number_support_sid

Number of supported diagnostic services

Definition at line 206 of file lin.h.

16.60.2.3.9 l_u16 P2_min

P2 min

Definition at line 202 of file lin.h.

16.60.2.3.10 lin_product_id_t product_id

Product ID

Definition at line 193 of file lin.h.

16.60.2.3.11   l_u8∗ resp_err_frm_id_ptr

List index of frame contain response error signal

Definition at line 195 of file lin.h.

16.60.2.3.12   l_signal_handle response_error

Signal used to update response error

Definition at line 197 of file lin.h.

16.60.2.3.13   l_u8∗ response_error_bit_offset_ptr

Bit offset of response error signal

Definition at line 199 of file lin.h.

16.60.2.3.14   l_u8∗ response_error_byte_offset_ptr

Byte offset of response error signal

Definition at line 198 of file lin.h.

16.60.2.3.15   lin_serial_number_t serial_number

Serial number

Definition at line 194 of file lin.h.

16.60.2.3.16   l_u8∗ service_flags_ptr

List of associated flags with supported diagnostic services

Definition at line 208 of file lin.h.

16.60.2.3.17   const l_u8∗ service_supported_ptr

List of supported diagnostic service

Definition at line 207 of file lin.h.

16.60.2.3.18   l_u16 ST_min

ST min

Definition at line 203 of file lin.h.

16.60.2.4   struct lin_associate_frame_t

Informations of associated frame Implements : lin_associate_frame_t_Class.

Definition at line 240 of file lin.h.

Data Fields

- l_u8 num_of_associated_uncond_frames
- const l_frame_handle ∗ associated_uncond_frame_ptr
- l_u8 coll_resolv_schd

Field Documentation

16.60.2.4.1   const l_frame_handle∗ associated_uncond_frame_ptr

Associated unconditional frame ID

Definition at line 243 of file lin.h.

16.60.2.4.2 l_u8 coll_resolv_schd

Collision resolver index in the schedule table, used in event trigger frame case MASTER

Definition at line 244 of file lin.h.

16.60.2.4.3 l_u8 num_of_associated_uncond_frames

Number of associated unconditional frame ID

Definition at line 242 of file lin.h.

16.60.2.5 struct lin_frame_t

Frame description structure Implements : lin_frame_t_Class.

Definition at line 251 of file lin.h.

Data Fields

- lin_frame_type_t frm_type
- l_u8 frm_len
- lin_frame_response_t frm_response
- l_u16 frm_offset
- l_u8 flag_offset
- l_u8 flag_size
- const lin_associate_frame_t ∗ frame_data_ptr

Field Documentation

16.60.2.5.1 l_u8 flag_offset

Flag byte offset in flag buffer

Definition at line 257 of file lin.h.

16.60.2.5.2 l_u8 flag_size

Flag size in flag buffer

Definition at line 258 of file lin.h.

16.60.2.5.3 const lin_associate_frame_t∗ frame_data_ptr

List of Signal to which the frame is associated and its offset

Definition at line 259 of file lin.h.

16.60.2.5.4 l_u8 frm_len

Length of the frame

Definition at line 254 of file lin.h.

16.60.2.5.5 l_u16 frm_offset

Frame byte offset in frame buffer

Definition at line 256 of file lin.h.

16.60.2.5.6 lin_frame_response_t frm_response

Action response when received PID

Definition at line 255 of file lin.h.

16.60.2.5.7 lin_frame_type_t frm_type

Frame information (unconditional or event triggered..)

Definition at line 253 of file lin.h.

16.60.2.6 struct lin_schedule_data_t

LIN schedule structure Implements : lin_schedule_data_t_Class.

Definition at line 288 of file lin.h.

Data Fields

- l_frame_handle frm_id
- l_u8 delay_integer
- lin_tl_queue_t tl_queue_data

Field Documentation

16.60.2.6.1 l_u8 delay_integer

Actual slot time in INTEGER for one frame

Definition at line 291 of file lin.h.

16.60.2.6.2 l_frame_handle frm_id

Frame ID, in case of unconditional or event triggered frame. For sporadic frame the value will be 0 (zero)

Definition at line 290 of file lin.h.

16.60.2.6.3 lin_tl_queue_t tl_queue_data

Data used in case of diagnostic or configuration frame

Definition at line 292 of file lin.h.

16.60.2.7 struct lin_schedule_t

Schedule table description Implements : lin_schedule_t_Class.

Definition at line 299 of file lin.h.

Data Fields

- l_u8 num_slots
- lin_sch_tbl_type_t sch_tbl_type
- const lin_schedule_data_t ∗ ptr_sch_data_ptr

Field Documentation

16.60.2.7.1 l_u8 num_slots

Number of frame slots in the schedule table

Definition at line 301 of file lin.h.

16.60.2.7.2 const lin_schedule_data_t∗ ptr_sch_data_ptr

Address of the schedule table

Definition at line 303 of file lin.h.

16.60.2.7.3   lin_sch_tbl_type_t sch_tbl_type

Schedule table type

Definition at line 302 of file lin.h.

16.60.2.8   struct lin_transport_layer_queue_t

Transport layer queue Implements : lin_transport_layer_queue_t_Class.

Definition at line 437 of file lin.h.

Data Fields

- l_u16 queue_header
- l_u16 queue_tail
- ld_queue_status_t queue_status
- l_u16 queue_current_size
- l_u16 queue_max_size
- lin_tl_pdu_data_t ∗ tl_pdu_ptr

Field Documentation

16.60.2.8.1   l_u16 queue_current_size

Current size

Definition at line 442 of file lin.h.

16.60.2.8.2   l_u16 queue_header

The first element of queue

Definition at line 439 of file lin.h.

16.60.2.8.3   l_u16 queue_max_size

Maximum size

Definition at line 443 of file lin.h.

16.60.2.8.4   ld_queue_status_t queue_status

Status of queue

Definition at line 441 of file lin.h.

16.60.2.8.5   l_u16 queue_tail

The last element of queue

Definition at line 440 of file lin.h.

16.60.2.8.6   lin_tl_pdu_data_t∗ tl_pdu_ptr

PDU data

Definition at line 444 of file lin.h.

16.60.2.9   struct lin_tl_descriptor_t

Transport layer description Implements : lin_tl_descriptor_t_Class.

Definition at line 464 of file lin.h.

Data Fields

- lin_transport_layer_queue_t tl_tx_queue
- lin_transport_layer_queue_t tl_rx_queue
- lin_message_status_t rx_msg_status
- l_u16 rx_msg_size
- lin_message_status_t tx_msg_status
- l_u16 tx_msg_size
- lin_last_cfg_result_t last_cfg_result
- l_u8 last_RSID
- l_u8 ld_error_code
- lin_message_timeout_type_t check_timeout_type
- l_u16 check_timeout
- lin_product_id_t ∗ product_id_ptr
- l_u8 num_of_pdu
- l_u8 frame_counter
- lin_diagnostic_state_t diag_state
- diag_interleaved_state_t diag_interleave_state
- l_u16 interleave_timeout_counter
- l_u8 slave_resp_cnt
- lin_service_status_t service_status
- bool ld_return_data
- bool FF_pdu_received
- l_u8 ∗ receive_message_ptr
- l_u8 ∗ receive_NAD_ptr
- l_u16 ∗ receive_message_length_ptr

Field Documentation

16.60.2.9.1 l_u16 check_timeout

Timeout counter for N_As and N_Cr timeout

Definition at line 484 of file lin.h.

16.60.2.9.2 lin_message_timeout_type_t check_timeout_type

Timeout type

Definition at line 483 of file lin.h.

16.60.2.9.3 diag_interleaved_state_t diag_interleave_state

state of diagnostic interleaved mode

Definition at line 489 of file lin.h.

16.60.2.9.4 lin_diagnostic_state_t diag_state

Diagnostic state

Definition at line 488 of file lin.h.

16.60.2.9.5 bool FF_pdu_received

Status of FF pdu

Definition at line 495 of file lin.h.

16.60.2.9.6   l_u8 frame_counter

Frame counter in received message

Definition at line 487 of file lin.h.

16.60.2.9.7   l_u16 interleave_timeout_counter

Interleaved timeout counter

Definition at line 490 of file lin.h.

16.60.2.9.8   lin_last_cfg_result_t last_cfg_result

Status of the last configuration service

Definition at line 479 of file lin.h.

16.60.2.9.9   l_u8 last_RSID

RSID of the last node configuration service

Definition at line 480 of file lin.h.

16.60.2.9.10   l_u8 ld_error_code

Error code in case of positive response

Definition at line 481 of file lin.h.

16.60.2.9.11   bool ld_return_data

Decide return data of diagnostic frame to pointer of ld_receive_message function

Definition at line 494 of file lin.h.

16.60.2.9.12   l_u8 num_of_pdu

Number of received pdu

Definition at line 486 of file lin.h.

16.60.2.9.13   lin_product_id_t∗ product_id_ptr

To store address of RAM area contain response

Definition at line 485 of file lin.h.

16.60.2.9.14   l_u16∗ receive_message_length_ptr

Pointer to receive_message_length of user

Definition at line 500 of file lin.h.

16.60.2.9.15   l_u8∗ receive_message_ptr

Pointer to receive_message array of user

Definition at line 498 of file lin.h.

16.60.2.9.16   l_u8∗ receive_NAD_ptr

Pointer to receive_NAD of user

Definition at line 499 of file lin.h.

16.60.2.9.17   l_u16 rx_msg_size

Size of message in queue

Definition at line 473 of file lin.h.

16.60.2.9.18   lin_message_status_t rx_msg_status

Cooked rx status

Definition at line 472 of file lin.h.

16.60.2.9.19   lin_service_status_t service_status

Status of the last configuration service

Definition at line 492 of file lin.h.

16.60.2.9.20   l_u8 slave_resp_cnt

Slave Response data counter

Definition at line 491 of file lin.h.

16.60.2.9.21   lin_transport_layer_queue_t tl_rx_queue

Pointer to receive queue on TL

Definition at line 468 of file lin.h.

16.60.2.9.22   lin_transport_layer_queue_t tl_tx_queue

Pointer to transmit queue on TL

Definition at line 467 of file lin.h.

16.60.2.9.23   l_u16 tx_msg_size

Size of message in queue

Definition at line 477 of file lin.h.

16.60.2.9.24   lin_message_status_t tx_msg_status

Cooked tx status

Definition at line 476 of file lin.h.

16.60.2.10   struct lin_protocol_user_config_t

Configuration structure Implements : lin_protocol_user_config_t_Class.

Definition at line 510 of file lin.h.

Data Fields

- lin_protocol_handle_t protocol_version
- lin_protocol_handle_t language_version
- lin_diagnostic_class_t diagnostic_class
- bool function
- l_u8 number_of_configurable_frames
- l_u8 frame_start
- const lin_frame_t ∗ frame_tbl_ptr
- const l_u16 ∗ list_identifiers_ROM_ptr
- l_u8 ∗ list_identifiers_RAM_ptr

- l_u16 max_idle_timeout_cnt
- l_u8 num_of_schedules
- l_u8 schedule_start
- const lin_schedule_t ∗ schedule_tbl
- l_ifc_slave_handle slave_ifc_handle
- l_ifc_master_handle master_ifc_handle
- lin_user_config_t ∗ lin_user_config_ptr
- lin_tl_pdu_data_t ∗ tl_tx_queue_data_ptr
- lin_tl_pdu_data_t ∗ tl_rx_queue_data_ptr
- l_u16 max_message_length

Field Documentation

16.60.2.10.1 lin_diagnostic_class_t diagnostic_class

Diagnostic class

Definition at line 514 of file lin.h.

16.60.2.10.2 l_u8 frame_start

Start index of frame list

Definition at line 518 of file lin.h.

16.60.2.10.3 const lin_frame_t∗ frame_tbl_ptr

Frame list except diagnostic frames

Definition at line 519 of file lin.h.

16.60.2.10.4 bool function

Function LIN_MASTER or LIN_SLAVE_)

Definition at line 515 of file lin.h.

16.60.2.10.5 lin_protocol_handle_t language_version

Language version

Definition at line 513 of file lin.h.

16.60.2.10.6 lin_user_config_t∗ lin_user_config_ptr

Pointer to LIN driver user configuration structure

Definition at line 529 of file lin.h.

16.60.2.10.7 l_u8∗ list_identifiers_RAM_ptr

Configuration in RAM

Definition at line 522 of file lin.h.

16.60.2.10.8 const l_u16∗ list_identifiers_ROM_ptr

Configuration in ROM

Definition at line 521 of file lin.h.

16.60.2.10.9 l_ifc_master_handle master_ifc_handle

Interface handler of master node

Definition at line 528 of file lin.h.

16.60.2.10.10   l_u16 max_idle_timeout_cnt

Max Idle timeout counter

Definition at line 523 of file lin.h.

16.60.2.10.11   l_u16 max_message_length

Max message length

Definition at line 533 of file lin.h.

16.60.2.10.12   l_u8 num_of_schedules

Number of schedule table

Definition at line 524 of file lin.h.

16.60.2.10.13   l_u8 number_of_configurable_frames

Number of frame except diagnostic frames

Definition at line 517 of file lin.h.

16.60.2.10.14   lin_protocol_handle_t protocol_version

Protocol version

Definition at line 512 of file lin.h.

16.60.2.10.15   l_u8 schedule_start

Start index of schedule table list

Definition at line 525 of file lin.h.

16.60.2.10.16   const lin_schedule_t∗ schedule_tbl

Schedule table list

Definition at line 526 of file lin.h.

16.60.2.10.17   l_ifc_slave_handle slave_ifc_handle

Interface handler of slave node

Definition at line 527 of file lin.h.

16.60.2.10.18   lin_tl_pdu_data_t∗ tl_rx_queue_data_ptr

Rx queue data

Definition at line 532 of file lin.h.

16.60.2.10.19   lin_tl_pdu_data_t∗ tl_tx_queue_data_ptr

Tx queue data

Definition at line 531 of file lin.h.

16.60.2.11   struct lin_master_data_t

LIN master configuration structure Implements : lin_master_data_t_Class.

Definition at line 541 of file lin.h.

Data Fields

- l_u8 active_schedule_id
- l_u8 previous_schedule_id
- l_u8 ∗ schedule_start_entry_ptr
- l_bool event_trigger_collision_flg
- l_u8 master_data_buffer [8]
- l_u16 frm_offset
- l_u8 frm_size
- l_u8 flag_offset
- l_u8 flag_size
- l_bool send_slave_res_flg
- l_bool send_functional_request_flg

Field Documentation

16.60.2.11.1   l_u8 active_schedule_id

Active schedule table id

Definition at line 543 of file lin.h.

16.60.2.11.2   l_bool event_trigger_collision_flg

Flag trigger collision event

Definition at line 546 of file lin.h.

16.60.2.11.3   l_u8 flag_offset

Flag offset

Definition at line 550 of file lin.h.

16.60.2.11.4   l_u8 flag_size

Flag size

Definition at line 551 of file lin.h.

16.60.2.11.5   l_u16 frm_offset

Frame offset

Definition at line 548 of file lin.h.

16.60.2.11.6   l_u8 frm_size

Size of frame

Definition at line 549 of file lin.h.

16.60.2.11.7   l_u8 master_data_buffer[8]

Master data buffer

Definition at line 547 of file lin.h.

16.60.2.11.8   l_u8 previous_schedule_id

Previous schedule table id

Definition at line 544 of file lin.h.

16.60.2.11.9   l_u8∗ schedule_start_entry_ptr

Start entry of each schedule table

Definition at line 545 of file lin.h.

16.60.2.11.10   l_bool send_functional_request_flg

Flag send Functional Request

Definition at line 553 of file lin.h.

16.60.2.11.11   l_bool send_slave_res_flg

Flag to send Slave Response Schedule

Definition at line 552 of file lin.h.

16.60.2.12   struct lin_protocol_state_t

LIN protocol status structure Implements : lin_protocol_state_t_Class.

Definition at line 560 of file lin.h.

Data Fields

- l_u16 baud_rate
- l_u8 ∗ response_buffer_ptr
- l_u8 response_length
- l_u8 successful_transfer
- l_u8 error_in_response
- l_bool go_to_sleep_flg
- l_u8 current_id
- l_u8 last_pid
- l_u8 num_of_processed_frame
- l_u8 overrun_flg
- lin_word_status_str_t word_status
- l_u8 next_transmit_tick
- l_bool save_config_flg
- l_diagnostic_mode_t diagnostic_mode
- l_u16 frame_timeout_cnt
- l_u16 idle_timeout_cnt
- l_bool transmit_error_resp_sig_flg

Field Documentation

16.60.2.12.1   l_u16 baud_rate

Adjusted baud rate

Definition at line 563 of file lin.h.

16.60.2.12.2   l_u8 current_id

Current PID

Definition at line 569 of file lin.h.

16.60.2.12.3   l_diagnostic_mode_t diagnostic_mode

Diagnostic mode

Definition at line 576 of file lin.h.

16.60.2.12.4   l_u8 error_in_response

Error response

Definition at line 567 of file lin.h.

16.60.2.12.5   l_u16 frame_timeout_cnt

Frame timeout counter for monitoring if timeout occurs during data transferring

Definition at line 577 of file lin.h.

16.60.2.12.6   l_bool go_to_sleep_flg

Go to sleep flag

Definition at line 568 of file lin.h.

16.60.2.12.7   l_u16 idle_timeout_cnt

Idle timeout counter

Definition at line 578 of file lin.h.

16.60.2.12.8   l_u8 last_pid

Last PID

Definition at line 570 of file lin.h.

16.60.2.12.9   l_u8 next_transmit_tick

Used to count the next transmit tick

Definition at line 574 of file lin.h.

16.60.2.12.10   l_u8 num_of_processed_frame

Number of processed frames

Definition at line 571 of file lin.h.

16.60.2.12.11   l_u8 overrun_flg

overrun flag

Definition at line 572 of file lin.h.

16.60.2.12.12   l_u8∗ response_buffer_ptr

Response buffer

Definition at line 564 of file lin.h.

16.60.2.12.13   l_u8 response_length

Response length

Definition at line 565 of file lin.h.

16.60.2.12.14   l_bool save_config_flg

Set when save configuration request has been received

Definition at line 575 of file lin.h.

16.60.2.12.15 l_u8 successful_transfer

Transfer flag

Definition at line 566 of file lin.h.

16.60.2.12.16 l_bool transmit_error_resp_sig_flg

Flag indicates that the error response signal is going to be sent

Definition at line 579 of file lin.h.

16.60.2.12.17 lin_word_status_str_t word_status

Word status

Definition at line 573 of file lin.h.

16.60.3 Macro Definition Documentation

16.60.3.1 #define CALLBACK_HANDLER(*iii, event_id, id*) lin_pid_resp_callback_handler((iii), (event_id), (id))

CALLBACK_HANDLER.

Note

> call lin_pid_resp_callback_handler() function in MASTER mode

Definition at line 687 of file lin.h.

16.60.3.2 #define INTERLEAVE_MAX_TIMEOUT (l_u16)(1000000U/TIME_OUT_UNIT_US)

Slave node interleaved diagnostic response timeout

Definition at line 447 of file lin.h.

16.60.3.3 #define LD_ID_NO_RESPONSE 0x52U

Positive response

Definition at line 87 of file lin.h.

16.60.3.4 #define LD_NEGATIVE_RESPONSE 0x53U

Negative response

Definition at line 88 of file lin.h.

16.60.3.5 #define LD_POSITIVE_RESPONSE 0x54U

Positive response

Definition at line 89 of file lin.h.

16.60.3.6 #define LIN_LLD_ERROR 0xFFU

Return value is ERROR

Definition at line 93 of file lin.h.

16.60.3.7 #define LIN_LLD_OK 0x00U

Return value is OK

Definition at line 92 of file lin.h.

16.60.3.8   #define LIN_MASTER 1

Master node

Definition at line 168 of file lin.h.

16.60.3.9   #define LIN_READ_USR_DEF_MAX 63U

Max user defined

Definition at line 84 of file lin.h.

16.60.3.10   #define LIN_READ_USR_DEF_MIN 32U

Min user defined

Definition at line 83 of file lin.h.

16.60.3.11   #define LIN_SLAVE 0

Mode of LIN node (master or slave)

Slave node

Definition at line 167 of file lin.h.

16.60.3.12   #define LIN_TL_CALLBACK_HANDLER(*iii,   tl_event_id,   id*) lin_tl_callback_handler((iii), (tl_event_id), (id))

Definition at line 374 of file lin.h.

16.60.3.13   #define PCI_RES_ASSIGN_FRAME_ID_RANGE 0x01U

PCI response value assign frame id range

Definition at line 80 of file lin.h.

16.60.3.14   #define PCI_RES_READ_BY_IDENTIFY 0x06U

PCI response value read by identify

Definition at line 78 of file lin.h.

16.60.3.15   #define PCI_RES_SAVE_CONFIGURATION 0x01U

PCI response value save configuration

Definition at line 79 of file lin.h.

16.60.3.16   #define PCI_SAVE_CONFIGURATION 0x01U

PCI value save configuration

Definition at line 75 of file lin.h.

16.60.3.17   #define SERIVCE_FAULT_MEMORY_CLEAR 0x14U

Service fault memory clear

Definition at line 72 of file lin.h.

16.60.3.18   #define SERVICE_ASSIGN_FRAME_ID 0xB1U

Assign frame id service

Definition at line 61 of file lin.h.

16.60.3.19   #define SERVICE_ASSIGN_FRAME_ID_RANGE 0xB7U

Assign frame id range service

Definition at line 65 of file lin.h.

16.60.3.20   #define SERVICE_ASSIGN_NAD 0xB0U

Assign NAD service

Definition at line 60 of file lin.h.

16.60.3.21   #define SERVICE_CONDITIONAL_CHANGE_NAD 0xB3U

Conditional change NAD service

Definition at line 63 of file lin.h.

16.60.3.22   #define SERVICE_FAULT_MEMORY_READ 0x19U

Service fault memory read

Definition at line 71 of file lin.h.

16.60.3.23   #define SERVICE_IO_CONTROL_BY_IDENTIFY 0x2FU

Service I/O control

Definition at line 70 of file lin.h.

16.60.3.24   #define SERVICE_READ_BY_IDENTIFY 0xB2U

Read by identify service

Definition at line 62 of file lin.h.

16.60.3.25   #define SERVICE_READ_DATA_BY_IDENTIFY 0x22U

Service read data by identifier

Definition at line 67 of file lin.h.

16.60.3.26   #define SERVICE_SAVE_CONFIGURATION 0xB6U

Save configuration service

Definition at line 64 of file lin.h.

16.60.3.27   #define SERVICE_SESSION_CONTROL 0x10U

Service session control

Definition at line 69 of file lin.h.

16.60.3.28   #define SERVICE_WRITE_DATA_BY_IDENTIFY 0x2EU

Service write data by identifier

Definition at line 68 of file lin.h.

16.60.4   Typedef Documentation

16.60.4.1   typedef l_u8 lin_tl_pdu_data_t[8]

PDU data. Implements : lin_tl_pdu_data_t_Class.

Definition at line 99 of file lin.h.

16.60.4.2 typedef l_u8 lin_tl_queue_t[8]

LIN transport layer queue Implements : lin_tl_queue_t_Class.

Definition at line 269 of file lin.h.

16.60.5 Enumeration Type Documentation

16.60.5.1 enum diag_interleaved_state_t

State of diagnostic interleaved mode Implements : diag_interleaved_state_t_Class.

Enumerator

> ***DIAG_NOT_START***   Not into slave response schedule with interleaved mode
> ***DIAG_NO_RESPONSE***   Master send 0x3D but slave does not response
> ***DIAG_RESPONSE***   Response receive

Definition at line 453 of file lin.h.

16.60.5.2 enum l_diagnostic_mode_t

Diagnostic mode Implements : l_diagnostic_mode_t_Class.

Enumerator

> ***DIAG_NONE***   None
> ***DIAG_INTERLEAVE_MODE***   Interleave mode
> ***DIAG_ONLY_MODE***   Diagnostic only mode

Definition at line 313 of file lin.h.

16.60.5.3 enum ld_queue_status_t

Status of queue Implements : ld_queue_status_t_Class.

Enumerator

> ***LD_NO_DATA***   Rx Queue is empty, has no data
> ***LD_DATA_AVAILABLE***   Data in queue is available
> ***LD_RECEIVE_ERROR***   Receive data is error for LIN21 and above
> ***LD_QUEUE_FULL***   The queue is full
> ***LD_QUEUE_AVAILABLE***   Queue is available for insert data for LIN21 and above
> ***LD_QUEUE_EMPTY***   Tx Queue is empty
> ***LD_TRANSMIT_ERROR***   Error while transmitting for LIN21 and above
> ***LD_TRANSFER_ERROR***   Error while transmitting/receiving for LIN20 and J2602

Definition at line 380 of file lin.h.

16.60.5.4 enum lin_diagnostic_class_t

List of diagnostic classes Implements : lin_diagnostic_class_t_Class.

Enumerator

> ***LIN_DIAGNOSTIC_CLASS_I***   LIN Diagnostic Class 1

> ***LIN_DIAGNOSTIC_CLASS_II***   LIN Diagnostic Class 2
>
> ***LIN_DIAGNOSTIC_CLASS_III***   LIN Diagnostic Class 3

Definition at line 139 of file lin.h.

16.60.5.5   enum lin_diagnostic_state_t

LIN diagnostic state Implements : lin_diagnostic_state_t_Class.

Enumerator

> ***LD_DIAG_IDLE***   IDLE
>
> ***LD_DIAG_TX_PHY***   Diagnostic transmit physical
>
> ***LD_DIAG_TX_FUNCTIONAL***   Diagnostic transmit active
>
> ***LD_DIAG_TX_INTERLEAVED***   Diagnostic transmit in interleave mode
>
> ***LD_DIAG_RX_PHY***   Diagnostic receive in physical
>
> ***LD_DIAG_RX_FUNCTIONAL***   Diagnostic receive functional request
>
> ***LD_DIAG_RX_INTERLEAVED***   Diagnostic receive in interleave mode

Definition at line 411 of file lin.h.

16.60.5.6   enum lin_frame_response_t

LIN frame response Implements : lin_frame_response_t_Class.

Enumerator

> ***LIN_RES_PUB***   Publisher response
>
> ***LIN_RES_SUB***   Subscriber response

Definition at line 230 of file lin.h.

16.60.5.7   enum lin_frame_type_t

Types of frame Implements : lin_frame_type_t_Class.

Enumerator

> ***LIN_FRM_UNCD***   Unconditional frame
>
> ***LIN_FRM_EVNT***   Event triggered frame
>
> ***LIN_FRM_SPRDC***   Sporadic frame
>
> ***LIN_FRM_DIAG***   Diagnostic frame

Definition at line 218 of file lin.h.

16.60.5.8   enum lin_last_cfg_result_t

Status of the last configuration call completed Implements : lin_last_cfg_result_t_Class.

Enumerator

> ***LD_SUCCESS***   The service was successfully carried out
>
> ***LD_NEGATIVE***   The service failed, more information can be found by parsing error_code
>
> ***LD_NO_RESPONSE***   No response was received on the request
>
> ***LD_OVERWRITTEN***   The slave response frame has been overwritten by another operation

Definition at line 336 of file lin.h.

16.60.5.9 enum lin_lld_event_id_t

Event id Implements : lin_lld_event_id_t_Class.

Enumerator

>*LIN_LLD_PID_OK* LIN_LLD_PID_OK
>
>*LIN_LLD_TX_COMPLETED* LIN_LLD_TX_COMPLETED
>
>*LIN_LLD_RX_COMPLETED* LIN_LLD_RX_COMPLETED
>
>*LIN_LLD_PID_ERR* LIN_LLD_PID_ERR
>
>*LIN_LLD_FRAME_ERR* LIN_LLD_FRAME_ERR
>
>*LIN_LLD_CHECKSUM_ERR* LIN_LLD_CHECKSUM_ERR
>
>*LIN_LLD_READBACK_ERR* LIN_LLD_READBACK_ERR
>
>*LIN_LLD_NODATA_TIMEOUT* No data timeout or received part of data but not completed
>
>*LIN_LLD_BUS_ACTIVITY_TIMEOUT* LIN_LLD_BUS_ACTIVITY_TIMEOUT

Definition at line 109 of file lin.h.

16.60.5.10 enum lin_message_status_t

Status of LIN message Implements : lin_message_status_t_Class.

Enumerator

>*LD_NO_MSG* No message
>
>*LD_IN_PROGRESS* In progress
>
>*LD_COMPLETED* Completed
>
>*LD_FAILED* Failed
>
>*LD_N_AS_TIMEOUT* N_As timeout
>
>*LD_N_CR_TIMEOUT* N_Cr timeout
>
>*LD_WRONG_SN* Wrong sequence number

Definition at line 396 of file lin.h.

16.60.5.11 enum lin_message_timeout_type_t

Types of message timeout Implements : lin_message_timeout_type_t_Class.

Enumerator

>*LD_NO_CHECK_TIMEOUT* No check timeout
>
>*LD_CHECK_N_AS_TIMEOUT* check N_As timeout
>
>*LD_CHECK_N_CR_TIMEOUT* check N_Cr timeout

Definition at line 426 of file lin.h.

16.60.5.12 enum lin_protocol_handle_t

List of protocols Implements : lin_protocol_handle_t_Class.

Enumerator

>*LIN_PROTOCOL_21* LIN protocol version 2.1
>
>*LIN_PROTOCOL_J2602* J2602 protocol

Definition at line 129 of file lin.h.

16.60.5.13   enum **lin_sch_tbl_type_t**

Types of schedule tables Implements : lin_sch_tbl_type_t_Class.

Enumerator

> ***LIN_SCH_TBL_NULL***   Run nothing
>
> ***LIN_SCH_TBL_NORM***   Normal schedule table
>
> ***LIN_SCH_TBL_DIAG***   Diagnostic schedule table
>
> ***LIN_SCH_TBL_GO_TO_SLEEP***   Goto sleep schedule table
>
> ***LIN_SCH_TBL_COLL_RESOLV***   Collision resolving schedule table

Definition at line 275 of file lin.h.

16.60.5.14   enum **lin_service_status_t**

Status of the last configuration call for LIN 2.1 Implements : lin_service_status_t_Class.

Enumerator

> ***LD_SERVICE_BUSY***   Service is ongoing
>
> ***LD_REQUEST_FINISHED***   The configuration request has been completed
>
> ***LD_SERVICE_IDLE***   The configuration request/response combination has been completed
>
> ***LD_SERVICE_ERROR***   The configuration request or response experienced an error

Definition at line 324 of file lin.h.

16.60.5.15   enum **lin_tl_callback_return_t**

Transport layer event IDs Implements : lin_tl_callback_return_t_Class.

Enumerator

> ***TL_ACTION_NONE***   Default return value of call back function
>
> ***TL_ACTION_ID_IGNORE***   Ignore this ID

Definition at line 364 of file lin.h.

16.60.5.16   enum **lin_tl_event_id_t**

Transport layer event IDs Implements : lin_tl_event_id_t_Class.

Enumerator

> ***TL_MAKE_RES_DATA***   Make master request data
>
> ***TL_SLAVE_GET_ACTION***   Get slave action
>
> ***TL_TX_COMPLETED***   Transmit completed
>
> ***TL_RX_COMPLETED***   Receive completed
>
> ***TL_ERROR***   Transport error
>
> ***TL_TIMEOUT_SERVICE***   Transmit timeout
>
> ***TL_HANDLER_INTERLEAVE_MODE***   Interleave mode
>
> ***TL_RECEIVE_MESSAGE***   Return data for ld_receive_message function

Definition at line 348 of file lin.h.

16.60.6   Function Documentation

16.60.6.1   l_u8 ld_read_by_id_callout (l_ifc_handle *iii,* l_u8 *id,* l_u8 ∗ *data*)

16.60.6.2   static l_u16 lin_calc_max_header_timeout_cnt (l_u32 *baudRate*)   `[inline]`,`[static]`

Computes maximum header timeout.

Theader_Maximum = 1.4 ∗ THeader_Nominal, THeader_Nominal = 34 ∗ TBit, (13 nominal bits of break; 1 nominal bit of break delimiter; 10 bits for SYNC and 10 bits of PID) TIME_OUT_UNIT_US is in micro second

Parameters

in	*baudRate*	LIN network baud rate

Returns

maximum timeout for the selected baud rate

Implements : lin_calc_max_header_timeout_cnt_Activity

Definition at line 629 of file lin.h.

16.60.6.3   static l_u16 lin_calc_max_res_timeout_cnt (l_u32 *baudRate,* l_u8 *size*)   `[inline]`,`[static]`

Computes the maximum response timeout.

TResponse_Maximum = 1.4 ∗ TResponse_Nominal, TResponse_Nominal = 10 ∗ (NData+ 1) ∗ TBit

Parameters

in	*baudRate*	LIN network baud rate
in	*size*	frame size in bytes

Returns

maximum response timeout for the given baud rate and frame size

Implements : lin_calc_max_res_timeout_cnt_Activity

Definition at line 645 of file lin.h.

16.60.6.4   l_u8 lin_lld_deinit (l_ifc_handle *iii*)

This function disconnect the node from the cluster and free all hardware used.

Parameters

in	*iii*	LIN interface that is being handled

Returns

Zero for success
Non-zero for error

Definition at line 157 of file lin.c.

16.60.6.5   l_u8 lin_lld_get_state (l_ifc_handle *iii*)

This function gets current state of an interface.

Parameters

in	*iii*	LIN interface that is being handled

Returns

> current LIN node state

Definition at line 180 of file lin.c.

16.60.6.6   l_u8 lin_lld_ignore_response (l_ifc_handle *iii*)

This function terminates an on-going data transmission/reception.

Parameters

in	*iii*	LIN interface that is being handled

Returns

> Zero for success
> Non-zero for error

Definition at line 310 of file lin.c.

16.60.6.7   l_bool lin_lld_init (l_ifc_handle *iii*)

This function initializes a LIN hardware instance for operation. This function will initialize the run-time state structure to keep track of the on-going transfers, initialize the module to user defined settings and default settings, configure the IRQ state structure and enable the module-level interrupt to the core, and enable the LIN hardware module transmitter and receiver.

Parameters

in	*iii*	LIN interface that is being handled

Returns

> zero if the initialization was successful and non-zero if failed

Definition at line 91 of file lin.c.

16.60.6.8   l_u8 lin_lld_int_disable (l_ifc_handle *iii*)

Disable the interrupt related to the interface.

Parameters

in	*iii*	LIN interface that is being handled

Returns

> Zero for success
> Non-zero for error

Definition at line 287 of file lin.c.

16.60.6.9   l_u8 lin_lld_int_enable (l_ifc_handle *iii*)

Enable the interrupt related to the interface.

Parameters

in	*iii*	LIN interface that is being handled

Returns

> Zero for success
> Non-zero for error

Definition at line 264 of file lin.c.

16.60.6.10 l_u8 lin_lld_rx_response (l_ifc_handle *iii,* l_u8 *response_length*)

This function receives frame data into the LIN_lld_response_buffer[iii] buffer.

This function will prepare LIN interface to receive data and then return. Data bytes will be received to the buffer in the interrupt handler of LIN interface. This function returns zero if preparation of receiving data was successful.

Parameters

in	*iii*	LIN interface that is being handled
in	*response_length*	Length of response

Returns

> Zero for success
> Non-zero for error

Definition at line 397 of file lin.c.

16.60.6.11 l_u8 lin_lld_set_low_power_mode (l_ifc_handle *iii*)

Let the low level driver go to low power mode.

Parameters

in	*iii*	LIN interface that is being handled

Returns

> Zero for success
> Non-zero for error

Definition at line 333 of file lin.c.

16.60.6.12 l_u8 lin_lld_set_response (l_ifc_handle *iii,* l_u8 *response_length*)

This function sends frame data that is contained in LIN_lld_response_buffer[iii].

This function will send the first data byte in the buffer and then return. Next data bytes will be sent in the interrupt handler of LIN interface. This function returns zero if sending of first data byte was successful.

Parameters

in	*iii*	LIN interface that is being handled
in	*response_length*	Length of response

Returns

> Zero for success
> Non-zero for error

Definition at line 356 of file lin.c.

16.60.6.13   void lin_lld_timeout_service (l_ifc_handle *iii*)

Callback function for Timer Interrupt Handler In timer IRQ handler, call this function. Used to check if frame timeout has occurred during frame data transmission and reception, to check for N_As and N_Cr timeout for LIN 2.1 and above. This function also check if there is no LIN bus communication (no headers and no frame data transferring) for Idle timeout (s), then put LIN node to Sleep mode. Users may initialize a timer (for example FTM)with period of Timeout unit (default: 500 micro seconds) to call lin_lld_timeout_service(). For an interface iii, Idle timeout (s) = max_idle_timeout_cnt ∗ Timeout unit (us) frame timeout (us) = frame_timeout_cnt ∗ Timeout unit (us) N_As timeout (us) = N_As_timeout ∗ Timeout unit (us) N_Cr timeout (us) = N_Cr_timeout ∗ Timeout unit (us)

Parameters

in	*iii*	LIN interface that is being handled

Returns

> void

Definition at line 433 of file lin.c.

16.60.6.14   l_u8 lin_lld_tx_header (l_ifc_handle *iii,* l_u8 *id*)

This function sends frame header for the input PID.

This function only initializes the sending of break field and then return. Then the sync byte and PID will be sent in the interrupt handler of LIN interface.

Parameters

in	*iii*	LIN interface that is being handled
in	*id*	ID of the header to be sent

Returns

> Zero for success
> Non-zero for error

Definition at line 203 of file lin.c.

16.60.6.15   l_u8 lin_lld_tx_wake_up (l_ifc_handle *iii*)

This function send a wakeup signal.

Parameters

in	*iii*	LIN interface that is being handled

Returns

> Zero for success
> Non-zero for error

Definition at line 236 of file lin.c.

16.60.6.16   void lin_pid_resp_callback_handler (l_ifc_handle *iii,* const lin_lld_event_id_t *event_id,* l_u8 *id*)

Callback handler for low level events.

This callback handler is being called from the LIN driver callback

Parameters

in	*iii*	LIN interface that is being handled
in	*event_id*	Low level event id lin_lld_event_id_t
in	*id*	Current protected identifier under processing by driver

Definition at line 69 of file lin_common_proto.c.

16.60.6.17 l_u8 lin_process_parity (l_u8 *pid,* l_u8 *typeAction*)

Makes or checks parity bits. If action is checking parity, the function returns ID value if parity bits are correct or 0xFF if parity bits are incorrect. If action is making parity bits, then from input value of ID, the function returns PID.

Parameters

pid	PID byte in case of checking parity bits or ID byte in case of making parity bits.
typeAction	TRUE for Checking parity bits, FALSE for making parity bits

Returns

    0xFF if parity bits are incorrect, ID in case of checking parity bits and they are correct. Function returns PID in case of making parity bits.

Definition at line 74 of file lin.c.

16.60.6.18 lin_tl_callback_return_t lin_tl_callback_handler (l_ifc_handle *iii,* lin_tl_event_id_t *tl_event_id,* l_u8 *id*)

Definition at line 86 of file lin_commontl_proto.c.

16.60.7 Variable Documentation

16.60.7.1 volatile l_u8 g_buffer_backup_data[8]

16.60.7.2 l_u8 g_lin_flag_handle_tbl[LIN_FLAG_BUF_SIZE]

16.60.7.3 l_u8 g_lin_frame_data_buffer[LIN_FRAME_BUF_SIZE]

16.60.7.4 l_bool g_lin_frame_flag_handle_tbl[LIN_NUM_OF_FRMS]

16.60.7.5 volatile l_u8 g_lin_frame_updating_flag_tbl[LIN_NUM_OF_FRMS]

16.60.7.6 const l_ifc_handle g_lin_hardware_ifc[HARDWARE_INSTANCE_COUNT]

16.60.7.7 lin_master_data_t g_lin_master_data_array[LIN_NUM_OF_MASTER_IFCS]

Global array for storing the master interfaces configurations

Definition at line 52 of file lin.c.

16.60.7.8 const lin_node_attribute_t g_lin_node_attribute_array[LIN_NUM_OF_SLAVE_IFCS]

16.60.7.9 lin_protocol_state_t g_lin_protocol_state_array[LIN_NUM_OF_IFCS]

Global array for storing the protocol state for each interface

Definition at line 50 of file lin.c.

16.60.7.10 const lin_protocol_user_config_t g_lin_protocol_user_cfg_array[LIN_NUM_OF_IFCS]

16.60.7.11 lin_tl_descriptor_t g_lin_tl_descriptor_array[LIN_NUM_OF_IFCS]

Global array for storing transport configuration for each interface

Definition at line 49 of file lin.c.

16.60.7.12   const l_u32 g_lin_virtual_ifc[LIN_NUM_OF_IFCS]

16.60.7.13   const lin_timer_get_time_interval_t timerGetTimeIntervalCallbackArr[LIN_NUM_OF_IFCS]

16.61 MPU Driver

16.61.1 Detailed Description

Memory Protection Unit Peripheral Driver.

Pre-Initialization information of MPU module

1. Before using the MPU driver the protocol clock of the module must be configured by the application using PCC module.

2. Bus fault or Hard fault exception must be configured to handle MPU access violation.

To initialize the MPU module, call the MPU_DRV_Init() function and provide the user configuration data structure. This function sets the configuration of the MPU module automatically and enables the MPU module.
The MPU includes default settings and protections for the Region Descriptor 0 (RGD0) such that the Debugger always has access to the entire address space and those rights cannot be changed by the core or any other bus master

- The access right for **CORE, DMA,..** can be **changed** except **DEBUGGER** master.

- The **start address**, **end address**, **process identifier** and **process identifier mask** are **ignored**.

This is example code to configure the MPU driver:

1. Define MPU instance

```
/* MPU 0 */
#define INST_MPU 0U

/* Status variable */
status_t status;
```

2. Configuration
   User configuration

```
/* Region count */
#define REGION_CNT (1U)

/* Master access configuration
   FEATURE_MPU_MASTER_COUNT macro has been already defined (number of masters supported by hardware)
 */
mpu_master_access_right_t masterAccRight[FEATURE_MPU_MASTER_COUNT] =
{
    /* CORE */
    {
        .masterNum   = FEATURE_MPU_MASTER_CORE,        /* Master number */
        .accessRight = MPU_SUPERVISOR_RWX_USER_RWX,    /* Access right */
        .processIdentifierEnable = false,              /* Process identifier enable */
    },
    /* The rest masters should be defined here */
    ...
}
/* User configuration */
mpu_user_config_t userConfig[REGION_CNT] =
{
    /* Region 0 */
    {
        .startAddr         = 0x00000000U,      /* Memory region start address */
        .endAddr           = 0xFFFFFFFFU,      /* Memory region end address */
        .masterAccRight    = masterAccRight,   /* Master access right */
        .processIdEnable   = false,            /* Process identifier enable */
        .processIdentifier = 0x00U,            /* Process identifier */
        .processIdMask     = 0x00U             /* Process identifier mask */
    }
}
```

or get default configuration

```
/* Defines master access right structure */
mpu_master_access_right_t masterAccRight[FEATURE_MPU_MASTER_COUNT];

/* Gets default region configuration
   Cover entire memory
   Access right of all masters are allowed
 */
mpu_user_config_t regionConfig0 =
     MPU_DRV_GetDefaultRegionConfig(masterAccRight);
mpu_user_config_t userConfig[REGION_CNT] =
{
    regionConfig0
};
```

3. Initializes

```
/* Initializes the MPU instance */
status = MPU_DRV_Init(INST_MPU, REGION_CNT, userConfig);
```

4. De-initializes

```
/* De-initializes the MPU instance */
MPU_DRV_Deinit(INST_MPU);
```

After MPU initialization:

- The MPU_DRV_SetRegionConfig() can be used to configure the region descriptor.

- The MPU_DRV_SetRegionAddr() can be used to configure the region start and end address.

- The MPU_DRV_SetMasterAccessRights() can be used to configure access permission of master in the region.

- The MPU_DRV_GetDetailErrorAccessInfo() API can be used to get the status of a slave port and the detail when an error occurred.

- The MPU_DRV_EnableRegion() can be used to enable or disable region descriptor.

Power management:

- To minimizes power dissipation, disables MPU module or regions by using MPU_DRV_Deinit()/MPU_DRV↵ _EnableRegion() when they are unused anymore.

Data Structures

- struct mpu_access_err_info_t

    MPU detail error access info Implements : mpu_access_err_info_t_Class. More...
- struct mpu_master_access_right_t

    MPU master access rights. Implements : mpu_master_access_right_t_Class. More...
- struct mpu_user_config_t

    MPU user region configuration structure. This structure is used when calling the MPU_DRV_Init function. Implements : mpu_user_config_t_Class. More...

Enumerations

- enum mpu_err_access_type_t { MPU_ERR_TYPE_READ = 0U, MPU_ERR_TYPE_WRITE = 1U }

    MPU access error Implements : mpu_err_access_type_t_Class.
- enum mpu_err_attributes_t { MPU_INSTRUCTION_ACCESS_IN_USER_MODE = 0U, MPU_DATA_ACC↵ ESS_IN_USER_MODE = 1U, MPU_INSTRUCTION_ACCESS_IN_SUPERVISOR_MODE = 2U, MPU_DA↵ TA_ACCESS_IN_SUPERVISOR_MODE = 3U }

    MPU access error attributes Implements : mpu_err_attributes_t_Class.

- enum mpu_access_rights_t {
MPU_SUPERVISOR_RWX_USER_NONE = 0x00U, MPU_SUPERVISOR_RWX_USER_X = 0x01U, MP↩
U_SUPERVISOR_RWX_USER_W = 0x02U, MPU_SUPERVISOR_RWX_USER_WX = 0x03U,
MPU_SUPERVISOR_RWX_USER_R = 0x04U, MPU_SUPERVISOR_RWX_USER_RX = 0x05U, MPU_S↩
UPERVISOR_RWX_USER_RW = 0x06U, MPU_SUPERVISOR_RWX_USER_RWX = 0x07U,
MPU_SUPERVISOR_RX_USER_NONE = 0x08U, MPU_SUPERVISOR_RX_USER_X = 0x09U, MPU_S↩
UPERVISOR_RX_USER_W = 0x0AU, MPU_SUPERVISOR_RX_USER_WX = 0x0BU,
MPU_SUPERVISOR_RX_USER_R = 0x0CU, MPU_SUPERVISOR_RX_USER_RX = 0x0DU, MPU_SUP↩
ERVISOR_RX_USER_RW = 0x0EU, MPU_SUPERVISOR_RX_USER_RWX = 0x0FU,
MPU_SUPERVISOR_RW_USER_NONE = 0x10U, MPU_SUPERVISOR_RW_USER_X = 0x11U, MPU_S↩
UPERVISOR_RW_USER_W = 0x12U, MPU_SUPERVISOR_RW_USER_WX = 0x13U,
MPU_SUPERVISOR_RW_USER_R = 0x14U, MPU_SUPERVISOR_RW_USER_RX = 0x15U, MPU_SUP↩
ERVISOR_RW_USER_RW = 0x16U, MPU_SUPERVISOR_RW_USER_RWX = 0x17U,
MPU_SUPERVISOR_USER_NONE = 0x18U, MPU_SUPERVISOR_USER_X = 0x19U, MPU_SUPERVIS↩
OR_USER_W = 0x1AU, MPU_SUPERVISOR_USER_WX = 0x1BU,
MPU_SUPERVISOR_USER_R = 0x1CU, MPU_SUPERVISOR_USER_RX = 0x1DU, MPU_SUPERVISO↩
R_USER_RW = 0x1EU, MPU_SUPERVISOR_USER_RWX = 0x1FU,
MPU_NONE = 0x80U, MPU_W = 0xA0U, MPU_R = 0xC0U, MPU_RW = 0xE0U }

MPU access rights.

Code	Supervisor	User	Description
MPU_SUPERVISOR_↩RWX_USER_NONE	*r w x*	*- - -*	*Allow Read, write, execute in supervisor mode; no access in user mode*
MPU_SUPERVISOR_↩RWX_USER_X	*r w x*	*- - x*	*Allow Read, write, execute in supervisor mode; execute in user mode*
MPU_SUPERVISOR_↩RWX_USER_W	*r w x*	*- w -*	*Allow Read, write, execute in supervisor mode; write in user mode*
MPU_SUPERVISOR_↩RWX_USER_WX	*r w x*	*- w x*	*Allow Read, write, execute in supervisor mode; write and execute in user mode*
MPU_SUPERVISOR_↩RWX_USER_R	*r w x*	*r - -*	*Allow Read, write, execute in supervisor mode; read in user mode*
MPU_SUPERVISOR_↩RWX_USER_RX	*r w x*	*r - x*	*Allow Read, write, execute in supervisor mode; read and execute in user mode*
MPU_SUPERVISOR_↩RWX_USER_RW	*r w x*	*r w -*	*Allow Read, write, execute in supervisor mode; read and write in user mode*
MPU_SUPERVISOR_↩RWX_USER_RWX	*r w x*	*r w x*	*Allow Read, write, execute in supervisor mode; read, write and execute in user mode*
MPU_SUPERVISOR_↩RX_USER_NONE	*r - x*	*- - -*	*Allow Read, execute in supervisor mode; no access in user mode*
MPU_SUPERVISOR_↩RX_USER_X	*r - x*	*- - x*	*Allow Read, execute in supervisor mode; execute in user mode*

MPU_SUPERVISOR_↩ RX_USER_W	r - x	- w -	Allow Read, execute in supervisor mode; write in user mode
MPU_SUPERVISOR_↩ RX_USER_WX	r - x	- w x	Allow Read, execute in supervisor mode; write and execute in user mode
MPU_SUPERVISOR_↩ RX_USER_R	r - x	r - -	Allow Read, execute in supervisor mode; read in user mode
MPU_SUPERVISOR_↩ RX_USER_RX	r - x	r - x	Allow Read, execute in supervisor mode; read and execute in user mode
MPU_SUPERVISOR_↩ RX_USER_RW	r - x	r w -	Allow Read, execute in supervisor mode; read and write in user mode
MPU_SUPERVISOR_↩ RX_USER_RWX	r - x	r w x	Allow Read, execute in supervisor mode; read, write and execute in user mode
MPU_SUPERVISOR_↩ RW_USER_NONE	r w -	- - -	Allow Read, write in supervisor mode; no access in user mode
MPU_SUPERVISOR_↩ RW_USER_X	r w -	- - x	Allow Read, write in supervisor mode; execute in user mode
MPU_SUPERVISOR_↩ RW_USER_W	r w -	- w -	Allow Read, write in supervisor mode; write in user mode
MPU_SUPERVISOR_↩ RW_USER_WX	r w -	- w x	Allow Read, write in supervisor mode; write and execute in user mode
MPU_SUPERVISOR_↩ RW_USER_R	r w -	r - -	Allow Read, write in supervisor mode; read in user mode
MPU_SUPERVISOR_↩ RW_USER_RX	r w -	r - x	Allow Read, write in supervisor mode; read and execute in user mode
MPU_SUPERVISOR_↩ RW_USER_RW	r w -	r w -	Allow Read, write in supervisor mode; read and write in user mode
MPU_SUPERVISOR_↩ RW_USER_RWX	r w -	r w x	Allow Read, write in supervisor mode; read, write and execute in user mode
MPU_SUPERVISOR_↩ USER_NONE	- - -	- - -	No access allowed in user and supervisor modes
MPU_SUPERVISOR_↩ USER_X	- - x	- - x	Execute operation is allowed in user and supervisor modes
MPU_SUPERVISOR_↩ USER_W	- w -	- w -	Write operation is allowed in user and supervisor modes

MPU_SUPERVISOR_↩ USER_WX	- w x	- w x	Write and execute operations are allowed in user and supervisor modes
MPU_SUPERVISOR_↩ USER_R	r - -	r - -	Read operation is allowed in user and supervisor modes
MPU_SUPERVISOR_↩ USER_RX	r - x	r - x	Read and execute operations are allowed in user and supervisor modes
MPU_SUPERVISOR_↩ USER_RW	r w -	r w -	Read and write operations are allowed in user and supervisor modes
MPU_SUPERVISOR_↩ USER_RWX	r w x	r w x	Read write and execute operations are allowed in user and supervisor modes

MPU Driver API

- status_t MPU_DRV_Init (uint32_t instance, uint8_t regionCnt, const mpu_user_config_t ∗userConfigArr)

  The function sets the MPU regions according to user input and then enables the MPU. Please note that access rights for region 0 will always be configured and regionCnt takes values between 1 and the maximum region count supported by the hardware. e.g. In S32K144 the number of supported regions is 8. The user must make sure that the clock is enabled.

- void MPU_DRV_Deinit (uint32_t instance)

  De-initializes the MPU region by resetting and disabling MPU module.

- void MPU_DRV_SetRegionAddr (uint32_t instance, uint8_t regionNum, uint32_t startAddr, uint32_t endAddr)

  Sets the region start and end address.

- status_t MPU_DRV_SetRegionConfig (uint32_t instance, uint8_t regionNum, const mpu_user_config_↩ t ∗userConfigPtr)

  Sets the region configuration.

- status_t MPU_DRV_SetMasterAccessRights (uint32_t instance, uint8_t regionNum, const mpu_master_↩ access_right_t ∗accessRightsPtr)

  Configures access permission.

- bool MPU_DRV_GetDetailErrorAccessInfo (uint32_t instance, uint8_t slavePortNum, mpu_access_err_info↩ _t ∗errInfoPtr)

  Checks and gets the MPU access error detail information for a slave port.

- mpu_user_config_t MPU_DRV_GetDefaultRegionConfig (mpu_master_access_right_t ∗masterAccRight)

  Gets default region configuration. Grants all access rights for masters and disable PID.

- void MPU_DRV_EnableRegion (uint32_t instance, uint8_t regionNum, bool enable)

  Enables/Disables region descriptor. Please note that region 0 should not be disabled.

16.61.2 Data Structure Documentation

16.61.2.1 struct mpu_access_err_info_t

MPU detail error access info Implements : mpu_access_err_info_t_Class.

Definition at line 66 of file mpu_driver.h.

Data Fields

- uint8_t master
- mpu_err_attributes_t attributes

- mpu_err_access_type_t accessType
- uint16_t accessCtr
- uint32_t addr

Field Documentation

16.61.2.1.1 uint16_t accessCtr

Access error control

Definition at line 71 of file mpu_driver.h.

16.61.2.1.2 mpu_err_access_type_t accessType

Access error type

Definition at line 70 of file mpu_driver.h.

16.61.2.1.3 uint32_t addr

Access error address

Definition at line 72 of file mpu_driver.h.

16.61.2.1.4 mpu_err_attributes_t attributes

Access error attributes

Definition at line 69 of file mpu_driver.h.

16.61.2.1.5 uint8_t master

Access error master

Definition at line 68 of file mpu_driver.h.

16.61.2.2 struct mpu_master_access_right_t

MPU master access rights. Implements : mpu_master_access_right_t_Class.

Definition at line 176 of file mpu_driver.h.

Data Fields

- uint8_t masterNum
- mpu_access_rights_t accessRight

Field Documentation

16.61.2.2.1 mpu_access_rights_t accessRight

Access right

Definition at line 179 of file mpu_driver.h.

16.61.2.2.2 uint8_t masterNum

Master number

Definition at line 178 of file mpu_driver.h.

16.61.2.3 struct mpu_user_config_t

MPU user region configuration structure. This structure is used when calling the MPU_DRV_Init function. Implements : mpu_user_config_t_Class.

Definition at line 190 of file mpu_driver.h.

Data Fields

- uint32_t startAddr
- uint32_t endAddr
- const mpu_master_access_right_t ∗ masterAccRight

Field Documentation

16.61.2.3.1 uint32_t endAddr

Memory region end address

Definition at line 193 of file mpu_driver.h.

16.61.2.3.2 const mpu_master_access_right_t∗ masterAccRight

Access permission for masters

Definition at line 194 of file mpu_driver.h.

16.61.2.3.3 uint32_t startAddr

Memory region start address

Definition at line 192 of file mpu_driver.h.

16.61.3 Enumeration Type Documentation

16.61.3.1 enum mpu_access_rights_t

MPU access rights.

Code	Supervisor	User	Description
MPU_SUPERVISOR_↩ RWX_USER_NONE	r w x	- - -	Allow Read, write, execute in supervisor mode; no access in user mode
MPU_SUPERVISOR_↩ RWX_USER_X	r w x	- - x	Allow Read, write, execute in supervisor mode; execute in user mode
MPU_SUPERVISOR_↩ RWX_USER_W	r w x	- w -	Allow Read, write, execute in supervisor mode; write in user mode
MPU_SUPERVISOR_↩ RWX_USER_WX	r w x	- w x	Allow Read, write, execute in supervisor mode; write and execute in user mode
MPU_SUPERVISOR_↩ RWX_USER_R	r w x	r - -	Allow Read, write, execute in supervisor mode; read in user mode
MPU_SUPERVISOR_↩ RWX_USER_RX	r w x	r - x	Allow Read, write, execute in supervisor mode; read and execute in user mode

MPU_SUPERVISOR_↩ RWX_USER_RW	r w x	r w -	Allow Read, write, execute in supervisor mode; read and write in user mode
MPU_SUPERVISOR_↩ RWX_USER_RWX	r w x	r w x	Allow Read, write, execute in supervisor mode; read, write and execute in user mode
MPU_SUPERVISOR_↩ RX_USER_NONE	r - x	- - -	Allow Read, execute in supervisor mode; no access in user mode
MPU_SUPERVISOR_↩ RX_USER_X	r - x	- - x	Allow Read, execute in supervisor mode; execute in user mode
MPU_SUPERVISOR_↩ RX_USER_W	r - x	- w -	Allow Read, execute in supervisor mode; write in user mode
MPU_SUPERVISOR_↩ RX_USER_WX	r - x	- w x	Allow Read, execute in supervisor mode; write and execute in user mode
MPU_SUPERVISOR_↩ RX_USER_R	r - x	r - -	Allow Read, execute in supervisor mode; read in user mode
MPU_SUPERVISOR_↩ RX_USER_RX	r - x	r - x	Allow Read, execute in supervisor mode; read and execute in user mode
MPU_SUPERVISOR_↩ RX_USER_RW	r - x	r w -	Allow Read, execute in supervisor mode; read and write in user mode
MPU_SUPERVISOR_↩ RX_USER_RWX	r - x	r w x	Allow Read, execute in supervisor mode; read, write and execute in user mode
MPU_SUPERVISOR_↩ RW_USER_NONE	r w -	- - -	Allow Read, write in supervisor mode; no access in user mode
MPU_SUPERVISOR_↩ RW_USER_X	r w -	- - x	Allow Read, write in supervisor mode; execute in user mode
MPU_SUPERVISOR_↩ RW_USER_W	r w -	- w -	Allow Read, write in supervisor mode; write in user mode
MPU_SUPERVISOR_↩ RW_USER_WX	r w -	- w x	Allow Read, write in supervisor mode; write and execute in user mode
MPU_SUPERVISOR_↩ RW_USER_R	r w -	r - -	Allow Read, write in supervisor mode; read in user mode

MPU_SUPERVISOR_↩RW_USER_RX	r w -	r - x	Allow Read, write in supervisor mode; read and execute in user mode
MPU_SUPERVISOR_↩RW_USER_RW	r w -	r w -	Allow Read, write in supervisor mode; read and write in user mode
MPU_SUPERVISOR_↩RW_USER_RWX	r w -	r w x	Allow Read, write in supervisor mode; read, write and execute in user mode
MPU_SUPERVISOR_↩USER_NONE	- - -	- - -	No access allowed in user and supervisor modes
MPU_SUPERVISOR_↩USER_X	- - x	- - x	Execute operation is allowed in user and supervisor modes
MPU_SUPERVISOR_↩USER_W	- w -	- w -	Write operation is allowed in user and supervisor modes
MPU_SUPERVISOR_↩USER_WX	- w x	- w x	Write and execute operations are allowed in user and supervisor modes
MPU_SUPERVISOR_↩USER_R	r - -	r - -	Read operation is allowed in user and supervisor modes
MPU_SUPERVISOR_↩USER_RX	r - x	r - x	Read and execute operations are allowed in user and supervisor modes
MPU_SUPERVISOR_↩USER_RW	r w -	r w -	Read and write operations are allowed in user and supervisor modes
MPU_SUPERVISOR_↩USER_RWX	r w x	r w x	Read write and execute operations are allowed in user and supervisor modes

Code	Read/Write permission	Description
MPU_NONE	- -	No Read/Write access permission
MPU_W	- w	Write access permission
MPU_R	r -	Read access permission
MPU_RW	r w	Read/Write access permission

Implements : mpu_access_rights_t_Class

Enumerator

> ***MPU_SUPERVISOR_RWX_USER_NONE***   0b00000000U : rwx|—
>
> ***MPU_SUPERVISOR_RWX_USER_X***   0b00000001U : rwx|–x
>
> ***MPU_SUPERVISOR_RWX_USER_W***   0b00000010U : rwx|-w-
>
> ***MPU_SUPERVISOR_RWX_USER_WX***   0b00000011U : rwx|-wx
>
> ***MPU_SUPERVISOR_RWX_USER_R***   0b00000100U : rwx|r–
>
> ***MPU_SUPERVISOR_RWX_USER_RX***   0b00000101U : rwx|r-x
>
> ***MPU_SUPERVISOR_RWX_USER_RW***   0b00000110U : rwx|rw-
>
> ***MPU_SUPERVISOR_RWX_USER_RWX***   0b00000111U : rwx|rwx

MPU_SUPERVISOR_RX_USER_NONE 0b00001000U : r-x|—

MPU_SUPERVISOR_RX_USER_X 0b00001001U : r-x|–x

MPU_SUPERVISOR_RX_USER_W 0b00001010U : r-x|-w-

MPU_SUPERVISOR_RX_USER_WX 0b00001011U : r-x|-wx

MPU_SUPERVISOR_RX_USER_R 0b00001100U : r-x|r–

MPU_SUPERVISOR_RX_USER_RX 0b00001101U : r-x|r-x

MPU_SUPERVISOR_RX_USER_RW 0b00001110U : r-x|rw-

MPU_SUPERVISOR_RX_USER_RWX 0b00001111U : r-x|rwx

MPU_SUPERVISOR_RW_USER_NONE 0b00010000U : rw-|—

MPU_SUPERVISOR_RW_USER_X 0b00010001U : rw-|–x

MPU_SUPERVISOR_RW_USER_W 0b00010010U : rw-|-w-

MPU_SUPERVISOR_RW_USER_WX 0b00010011U : rw-|-wx

MPU_SUPERVISOR_RW_USER_R 0b00010100U : rw-|r–

MPU_SUPERVISOR_RW_USER_RX 0b00010101U : rw-|r-x

MPU_SUPERVISOR_RW_USER_RW 0b00010110U : rw-|rw-

MPU_SUPERVISOR_RW_USER_RWX 0b00010111U : rw-|rwx

MPU_SUPERVISOR_USER_NONE 0b00011000U : —|—

MPU_SUPERVISOR_USER_X 0b00011001U : –x|–x

MPU_SUPERVISOR_USER_W 0b00011010U : -w-|-w-

MPU_SUPERVISOR_USER_WX 0b00011011U : -wx|-wx

MPU_SUPERVISOR_USER_R 0b00011100U : r–|r–

MPU_SUPERVISOR_USER_RX 0b00011101U : r-x|r-x

MPU_SUPERVISOR_USER_RW 0b00011110U : rw-|rw-

MPU_SUPERVISOR_USER_RWX 0b00011111U : rwx|rwx

MPU_NONE 0b10000000U : –

MPU_W 0b10100000U : w-

MPU_R 0b11000000U : -r

MPU_RW 0b11100000U : wr

Definition at line 124 of file mpu_driver.h.

16.61.3.2 enum mpu_err_access_type_t

MPU access error Implements : mpu_err_access_type_t_Class.

Enumerator

> *MPU_ERR_TYPE_READ* MPU error type: read
>
> *MPU_ERR_TYPE_WRITE* MPU error type: write

Definition at line 44 of file mpu_driver.h.

16.61.3.3 enum mpu_err_attributes_t

MPU access error attributes Implements : mpu_err_attributes_t_Class.

Enumerator

> *MPU_INSTRUCTION_ACCESS_IN_USER_MODE* Access instruction error in user mode
>
> *MPU_DATA_ACCESS_IN_USER_MODE* Access data error in user mode
>
> *MPU_INSTRUCTION_ACCESS_IN_SUPERVISOR_MODE* Access instruction error in supervisor mode
>
> *MPU_DATA_ACCESS_IN_SUPERVISOR_MODE* Access data error in supervisor mode

Definition at line 54 of file mpu_driver.h.

16.61.4 Function Documentation

16.61.4.1 void MPU_DRV_Deinit (uint32_t *instance*)

De-initializes the MPU region by resetting and disabling MPU module.

Parameters

in	*instance*	The MPU peripheral instance number.

Definition at line 105 of file mpu_driver.c.

16.61.4.2 void MPU_DRV_EnableRegion (uint32_t *instance,* uint8_t *regionNum,* bool *enable*)

Enables/Disables region descriptor. Please note that region 0 should not be disabled.

Parameters

in	*instance*	The MPU peripheral instance number.
in	*regionNum*	The region number.
in	*enable*	Valid state • true : Enable region. • false : Disable region.

Definition at line 323 of file mpu_driver.c.

16.61.4.3 mpu_user_config_t MPU_DRV_GetDefaultRegionConfig (mpu_master_access_right_t ∗ *masterAccRight*)

Gets default region configuration. Grants all access rights for masters and disable PID.

Parameters

in	*instance*	The MPU peripheral instance number.
out	*masterAccRight*	The pointer to master configuration structure, see mpu_master_access_↵ right_t. The length of array should be defined by number of masters supported by hardware.

Returns

      The default region configuration, see mpu_user_config_t.

Definition at line 287 of file mpu_driver.c.

16.61.4.4 bool MPU_DRV_GetDetailErrorAccessInfo (uint32_t *instance,* uint8_t *slavePortNum,* mpu_access_err_info_t ∗ *errInfoPtr*)

Checks and gets the MPU access error detail information for a slave port.

Parameters

in	*instance*	The MPU peripheral instance number.
in	*slavePortNum*	The slave port number to get Error Detail.
out	*errInfoPtr*	The pointer to access error info structure.

Returns

      operation status

• true : An error has occurred.

• false : No error has occurred.

Definition at line 254 of file mpu_driver.c.

16.61.4.5   status_t MPU_DRV_Init (uint32_t *instance,* uint8_t *regionCnt,* const mpu_user_config_t ∗ *userConfigArr*)

The function sets the MPU regions according to user input and then enables the MPU. Please note that access rights for region 0 will always be configured and regionCnt takes values between 1 and the maximum region count supported by the hardware. e.g. In S32K144 the number of supported regions is 8. The user must make sure that the clock is enabled.

Parameters

in	*instance*	The MPU peripheral instance number.
in	*regionCnt*	The number of configured regions.
in	*userConfigArr*	The pointer to the array of MPU user configure structure, see mpu_user_↩ config_t.

Returns

> operation status
>
> - STATUS_SUCCESS: Operation was successful.
> - STATUS_ERROR: Operation failed due to master number is out of range supported by hardware.

Definition at line 63 of file mpu_driver.c.

16.61.4.6   status_t MPU_DRV_SetMasterAccessRights (uint32_t *instance,* uint8_t *regionNum,* const mpu_master_access_right_t ∗ *accessRightsPtr*)

Configures access permission.

Parameters

in	*instance*	The MPU peripheral instance number.
in	*regionNum*	The MPU region number.
in	*accessRightsPtr*	The pointer to access permission structure.

Returns

> operation status
>
> - STATUS_SUCCESS: Operation was successful.
> - STATUS_ERROR: Operation failed due to master number is out of range supported by hardware.

Definition at line 222 of file mpu_driver.c.

16.61.4.7   void MPU_DRV_SetRegionAddr (uint32_t *instance,* uint8_t *regionNum,* uint32_t *startAddr,* uint32_t *endAddr*)

Sets the region start and end address.

Parameters

in	*instance*	The MPU peripheral instance number.
in	*regionNum*	The region number.
in	*startAddr*	The region start address.
in	*endAddr*	The region end address.

Definition at line 137 of file mpu_driver.c.

16.61.4.8   status_t MPU_DRV_SetRegionConfig (uint32_t *instance,* uint8_t *regionNum,* const mpu_user_config_t ∗ *userConfigPtr*)

Sets the region configuration.

Parameters

in	*instance*	The MPU peripheral instance number.
in	*regionNum*	The region number.
in	*userConfigPtr*	The region configuration structure pointer.

Returns

operation status

- STATUS_SUCCESS: Operation was successful.
- STATUS_ERROR: Operation failed due to master number is out of range supported by hardware.

Definition at line 162 of file mpu_driver.c.

16.62 Memory Protection Unit (MPU)

16.62.1 Detailed Description

The S32 SDK provides Peripheral Driver for the Memory Protection Unit (MPU) module of S32 SDK devices.

The memory protection unit (MPU) provides hardware access control for all memory references generated in the device.

Hardware background

The MPU concurrently monitors all system bus transactions and evaluates their appropriateness using pre-programmed region descriptors that define memory spaces and their access rights. Memory references that have sufficient access control rights are allowed to complete, while references that are not mapped to any region descriptor or have insufficient rights are terminated with a protection error response.

The MPU implements a two-dimensional hardware array of memory region descriptors and the crossbar slave ports to continuously monitor the legality of every memory reference generated by each bus master in the system.

The feature set includes:

- 8(16 for S32K148) program-visible 128-bit region descriptors, accessible by four 32-bit words each
  - Each region descriptor defines a modulo-32 byte space, aligned anywhere in memory
    * Region sizes can vary from 32 bytes to 4 Gbytes
  - Two access control permissions defined in a single descriptor word
    * Masters 0–3: read, write, and execute attributes for supervisor and user accesses
    * Masters 4–7: read and write attributes
  - Hardware-assisted maintenance of the descriptor valid bit minimizes coherency issues
  - Alternate programming model view of the access control permissions word
  - Priority given to granting permission over denying access for overlapping region descriptors
- Detects access protection errors if a memory reference does not hit in any memory region, or if the reference is illegal in all hit memory regions. If an access error occurs, the reference is terminated with an error response, and the MPU inhibits the bus cycle being sent to the targeted slave device.
- Error registers, per slave port, capture the last faulting address, attributes, and other information
- Global MPU enable/disable control bit

Logical Bus Master Assignments and Possible Access Types

ID	Master	User	Super-visor	Data	Instruc-tion	Read	Write	Exe-cute	PID
0	Core	x	x	x	x	x	x	x	x
1	Debug-ger	x	x	x	x	x	x	x	x
2	DMA		x	x		x	x		
3	ENET	x		x		x	x		

ID	S32K116	S32K142	S32K144	S32K146	S32K148	S32MTV
0	x	x	x	x	x	x
1	x	x	x	x	x	x
2	x	x	x	x	x	x
3					x	

Logical Slave Port Assignments

Port	Source	Destination
0	Crossbar slave port 0	Flash Controller
1	Crossbar slave port 1	SRAM backdoor
2	Code Bus	SRAM_L frontdoor
3	System Bus	SRAM_U frontdoor
4	Crossbar slave port 3	QuadSPI

Port	S32K116	S32K142	S32K144	S32K146	S32K148	S32MTV
0	x	x	x	x	x	x
1	x(1)	x	x	x	x	x
2		x	x	x	x	x
3		x	x	x	x	x
4					x	

1: Destination: SRAM controller/MTB/DWT/MCM

Modules

- MPU Driver

    Memory Protection Unit Peripheral Driver.

16.63   Node configuration

16.63.1   Detailed Description

This group contains APIs that used for node configuration purpose.

Functions

- l_bool ld_is_ready_j2602 (l_ifc_handle iii)

    Verifies a state of node setting (using for J2602 and LIN 2.0).

- l_u8 ld_check_response_j2602 (l_ifc_handle iii, l_u8 ∗const RSID, l_u8 ∗const error_code)

    Verifies the state of response (using for J2602 and LIN 2.0) Master node only.

- void ld_assign_frame_id (l_ifc_handle iii, l_u8 NAD, l_u16 supplier_id, l_u16 message_id, l_u8 PID)

    This function assigns the protected identifier to a slave node with the address NAD and specified supplier id (using for J2602 and LIN 2.0). Master node only.

- l_bool ld_assign_NAD_j2602 (l_ifc_handle iii, l_u8 dnn)

    This function assigns NAD of a J2602 slave device based on input DNN that is Device Node Number. NAD is (0x60+ DNN).

- l_bool ld_reconfig_msg_ID (l_ifc_handle iii, l_u8 dnn)

    This function reconfigures frame identifiers of a J2602 slave node based on input dnn.

16.63.2   Function Documentation

16.63.2.1   void ld_assign_frame_id (l_ifc_handle *iii,* l_u8 *NAD,* l_u16 *supplier_id,* l_u16 *message_id,* l_u8 *PID*)

This function assigns the protected identifier to a slave node with the address NAD and specified supplier id (using for J2602 and LIN 2.0). Master node only.

Parameters

in	*iii*	LIN interface handle
in	*initial_NAD*	Initial node address of the target node
in	*supplier_id*	Supplier ID of the target node
in	*message_id*	Message ID of the target node
in	*PID*	Protected ID of the target node

Returns

    void

Definition at line 1516 of file lin_diagnostic_service.c.

16.63.2.2   l_bool ld_assign_NAD_j2602 (l_ifc_handle *iii,* l_u8 *dnn*)

This function assigns NAD of a J2602 slave device based on input DNN that is Device Node Number. NAD is (0x60+ DNN).

Parameters

in	*iii*	LIN interface handle
in	*dnn*	DNN of the device

Returns

    l_bool: 0: successful: New Configured NAD is 0x60 + DNN
    l_bool: 1: Unsuccesfull: for either one of the following reasons:

- The protocol of this interface is not J2602

- This device is a Master node in this interface

- The input DNN is greater than 0xD that is invalid

Definition at line 1558 of file lin_diagnostic_service.c.

16.63.2.3   l_u8 ld_check_response_j2602 (l_ifc_handle *iii,* l_u8 ∗const *RSID,* l_u8 ∗const *error_code*)

Verifies the state of response (using for J2602 and LIN 2.0) Master node only.

Parameters

in	*iii*	LIN interface handle
out	*RSID*	buffer for saving the response ID
out	*error_code*	buffer for saving the error code

Returns

> l_u8 status of the last service

Definition at line 1472 of file lin_diagnostic_service.c.

16.63.2.4   l_bool ld_is_ready_j2602 (l_ifc_handle *iii*)

Verifies a state of node setting (using for J2602 and LIN 2.0).

Parameters

in	*iii*	LIN interface handle

Returns

> l_bool

Definition at line 1445 of file lin_diagnostic_service.c.

16.63.2.5   l_bool ld_reconfig_msg_ID (l_ifc_handle *iii,* l_u8 *dnn*)

This function reconfigures frame identifiers of a J2602 slave node based on input dnn.

Parameters

in	*iii*	LIN interface handle
in	*dnn*	DNN of the device

Returns

> l_bool: 0: successful: Frame Identifiers were reconfigured based on input DNN according to NAD Message
> ID mapping table.
> l_bool: 1: Unsuccesfull: for either one of the following reasons:

- The protocol of this interface is not J2602

- This device is a Master node in this interface

- The input DNN is greater than 0xD that is invalid

- The slave has more than 16 configurable frames

- The slave has 9-16 configurable frames, and dnn is 0xC or 0xD

- The slave has 5-8 configurable frames, and dnn is not 0x00, 0x2, 0x4, 0x6, 0x8, 0xA, 0xC.

Definition at line 1587 of file lin_diagnostic_service.c.

16.64   Node configuration

16.64.1   Detailed Description

This group contains APIs that used for node configuration purpose.

With protocol lin2.1 in slave node, some service like (Data dump, Conditional change nad with id from 2 to 255) are not supported by LinStack but user can implement it in application by use function ld_receive_message and ld_send_message in transport layer.

With protocol J2602 in slave node, some service like (Data dump, Assign NAD, Conditional change NAD) are not supported by LinStack but user can implement it in application by choosing these services in supported_sid in PEX GUI and use function ld_receive_message and ld_send_message in transport layer. When received target reset master request slave node just update status_byte and send response positive message.

Functions

- l_u8 ld_is_ready (l_ifc_handle iii)

    This call returns the status of the last requested configuration service.
- void ld_check_response (l_ifc_handle iii, l_u8 ∗const RSID, l_u8 ∗const error_code)

    This call returns the result of the last node configuration service, in the parameters RSID and error_code. A value in RSID is always returned but not always in the error_code. Default values for RSID and error_code is 0 (zero).
- void ld_assign_frame_id_range (l_ifc_handle iii, l_u8 NAD, l_u8 start_index, const l_u8 ∗const PIDs)

    This function assigns the protected identifier of up to four frames.
- void ld_save_configuration (l_ifc_handle iii, l_u8 NAD)

    This function to issue a save configuration request to a slave node.
- l_u8 ld_read_configuration (l_ifc_handle iii, l_u8 ∗const data, l_u8 ∗const length)

    This function copies current configuration in a reserved area.
- l_u8 ld_set_configuration (l_ifc_handle iii, const l_u8 ∗const data, l_u16 length)

    This function configures slave node according to data.
- void ld_assign_NAD (l_ifc_handle iii, l_u8 initial_NAD, l_u16 supplier_id, l_u16 function_id, l_u8 new_NAD)

    This call assigns the NAD (node diagnostic address) of all slave nodes that matches the initial_NAD, the supplier ID and the function ID. Master node only.
- void ld_conditional_change_NAD (l_ifc_handle iii, l_u8 NAD, l_u8 id, l_u8 byte_data, l_u8 mask, l_u8 invert, l_u8 new_NAD)

    This call changes the NAD if the node properties fulfill the test specified by id, byte, mask and invert. Master node only.

16.64.2   Function Documentation

16.64.2.1   void ld_assign_frame_id_range (l_ifc_handle *iii,* l_u8 *NAD,* l_u8 *start_index,* const l_u8 ∗const *PIDs*)

This function assigns the protected identifier of up to four frames.

Parameters

in	*iii*	lin interface handle
in	*NAD*	Node address value of the target node
in	*start_index*	specifies which is the first frame to assign a PID
in	*PIDs*	list of protected identifier

Returns

    void

This API is available for master interfaces only

Definition at line 136 of file lin_diagnostic_service.c.

16.64.2.2 void ld_assign_NAD (l_ifc_handle *iii,* l_u8 *initial_NAD,* l_u16 *supplier_id,* l_u16 *function_id,* l_u8 *new_NAD*)

This call assigns the NAD (node diagnostic address) of all slave nodes that matches the initial_NAD, the supplier ID and the function ID. Master node only.

Parameters

in	*iii*	LIN interface handle
in	*initial_NAD*	Initial node address of the target node
in	*supplier_id*	Supplier ID of the target node
in	*function_id*	Function identifier of the target node
in	*new_NAD*	New node address

Returns

> void

This call assigns the NAD (node diagnostic address) of all slave nodes that matches the initial_NAD, the supplier ID and the function ID. The new NAD of the slave node will be new_NAD. This function is used for master node only.

Definition at line 860 of file lin_diagnostic_service.c.

16.64.2.3   void ld_check_response (l_ifc_handle *iii,* l_u8 ∗const *RSID,* l_u8 ∗const *error_code*)

This call returns the result of the last node configuration service, in the parameters RSID and error_code. A value in RSID is always returned but not always in the error_code. Default values for RSID and error_code is 0 (zero).

For slave interfaces ld_check_response shall do nothing

Parameters

in	*iii*	lin interface handle
out	*RSID*	buffer for saving the response ID
out	*error_code*	buffer for saving the error code

This API is available for master interfaces only

Definition at line 106 of file lin_diagnostic_service.c.

16.64.2.4   void ld_conditional_change_NAD (l_ifc_handle *iii,* l_u8 *NAD,* l_u8 *id,* l_u8 *byte_data,* l_u8 *mask,* l_u8 *invert,* l_u8 *new_NAD*)

This call changes the NAD if the node properties fulfill the test specified by id, byte, mask and invert. Master node only.

Parameters

in	*iii*	:LIN interface handle
in	*NAD*	Current NAD value of the target node
in	*id*	Property ID of the target node
in	*byte*	Byte location of property value to be read from the target node
in	*mask*	Value for masking the read property byte
in	*invert*	Value for excluding the read property byte
in	*new_NAD*	New NAD value to be assigned when the condition is met

Returns

> void

This call changes the NAD if the node properties fulfill the test specified by id, byte, mask and invert.

Definition at line 902 of file lin_diagnostic_service.c.

16.64.2.5   l_u8 ld_is_ready (l_ifc_handle *iii*)

This call returns the status of the last requested configuration service.

Parameters

in		*iii*	lin interface handle

Returns

> LD_SERVICE_BUSY Service is ongoing.
>
> LD_REQUEST_FINISHED The configuration request has been completed. This is a intermediate status between the configuration request and configuration response.
>
> LD_SERVICE_IDLE The configuration request/response combination has been completed, i.e. the response is valid and may be analyzed. Also, this value is returned if no request has yet been called.
>
> LD_SERVICE_ERROR The configuration request or response experienced an error. Error here means error on the bus, and not a negative configuration response from the slave node.

Definition at line 80 of file lin_diagnostic_service.c.

16.64.2.6 l_u8 ld_read_configuration (l_ifc_handle *iii,* l_u8 ∗const *data,* l_u8 ∗const *length*)

This function copies current configuration in a reserved area.

Parameters

in		*iii*	Lin interface handle
out		*data*	Data area to save configuration,
out		*length*	Length of data area (1 + n, NAD + PIDs)

Returns

> LD_READ_OK If the service was successful.
>
> LD_LENGTH_TOO_SHORT If the configuration size is greater than the length. It means that the data area does not contain a valid configuration.

This function is implemented Slave Only. Set the expected length value to EXP = NN + NF, where : NN = the number of NAD. NF = the number of configurable frames; Moreover: Not taken PID's diagnostics frame: 3C, 3D

Definition at line 438 of file lin_diagnostic_service.c.

16.64.2.7 void ld_save_configuration (l_ifc_handle *iii,* l_u8 *NAD*)

This function to issue a save configuration request to a slave node.

Parameters

in		*iii*	Interface name
in		*NAD*	Node address of target

Returns

> void

This function is available for master nodes only. This function is available for all diagnostic classes and only for LIN2.1 and above. This function is called to send a save configuration request to a specific slave node with the given NAD, or to all slave nodes if NAD is set to broadcast This function is implemented for Master Only.

Definition at line 178 of file lin_diagnostic_service.c.

16.64.2.8 l_u8 ld_set_configuration (l_ifc_handle *iii,* const l_u8 ∗const *data,* l_u16 *length*)

This function configures slave node according to data.

Parameters

in	*iii*	Lin interface handle
in	*data*	Structure containing the NAD and all the n PIDs for the frames of the specified NAD,
in	*length*	Length of data area (1 + n, NAD + PIDs)

Returns

LD_SET_OK If the service was successful

LD_LENGTH_NOT_CORRECT If the required size of the configuration is not equal to the given length.

LD_DATA_ERROR The set of configuration could not be made.

This function is implemented Slave Only. Set the expected length value to EXP = NN + NF, where : NN = the number of NAD. NF = the number of configurable frames; Moreover: Not taken PID's diagnostics frame: 3C, 3D

Definition at line 503 of file lin_diagnostic_service.c.

16.65 Node identification

16.65.1 Detailed Description

This group contains API that used for node identification purpose.

Read by identifier service just support id 0 and 1. User can implement for other id by modify function ld_read_by↩
_id_callout in generated file lin_cfg.c.

Functions

- void ld_read_by_id (l_ifc_handle iii, l_u8 NAD, l_u16 supplier_id, l_u16 function_id, l_u8 id, lin_product_id_t
  ∗const data)

    The call requests the slave node selected with the NAD to return the property associated with the id parameter.
    Master node only.

16.65.2 Function Documentation

16.65.2.1 void ld_read_by_id (l_ifc_handle *iii,* l_u8 *NAD,* l_u16 *supplier_id,* l_u16 *function_id,* l_u8 *id,* lin_product_id_t ∗const *data*)

The call requests the slave node selected with the NAD to return the property associated with the id parameter.
Master node only.

Parameters

in	*iii*	LIN interface handle
in	*NAD*	Value of the target node
in	*supplier_id*	Supplier ID of the target node
in	*function_id*	Function ID of the target node
in	*id*	ID of the target node
out	*data*	Buffer for saving the data read from the node

Returns

   void

The call requests the slave node selected with the NAD to return the property associated with the id parameter.

Definition at line 950 of file lin_diagnostic_service.c.

16.66   Notification

This group contains APIs that let users know when a signal's value changed.

16.67 OS Interface (OSIF)

16.67.1 Detailed Description

OS Interface Layer (OSIF)

The OSIF layer is a minimal wrapper layer for common RTOS services, intended to be used by SDK drivers and middlewares. I can be used by the user application, but it is not recommended. The operations supported by OSIF:

- mutex lock/unlock

- semaphore post/wait

- time delay and get time elapsed

OSIF currently comes in two variants: bare-metal and FreeRTOS. Steps to use each one are described below.

FreeRTOS

To integrate the FreeRTOS OSIF variant, two steps are necessary:

- compile and link the osif_freertos.c file

- define a project-wide compile symbol: USING_OS_FREERTOS

FreeRTOSConfig.h dependencies

FreeRTOS configuration file needs to have these options activated:

- INCLUDE_xQueueGetMutexHolder

- INCLUDE_xTaskGetCurrentTaskHandle

Hardware resources

FreeRTOS OSIF uses the FreeRTOS API and services, does not use any additional hardware or software resources.

FreeRTOS supported platforms

The SDK platforms supported by FreeRTOS can be found in the following table. If a platform is supported by Free↩ RTOS, both osif variants, bare-metal and freertos, are supported. If the platform is not supported by FreeRTOS, only osif bare-metal variant is applicable:

Platform	FreeRTOS support
S32K11x	No
S32K14x	Yes
MPC5746C	Yes
MPC5748G	Yes
MPC5744P	Yes

Bare-metal

To integrate the bare-metal OSIF variant:

- compile and link the osif_baremetal.c file

- define a project-wide compile symbol: USING_OS_BAREMETAL. Note: this symbol is optional as the default behavior is to assume a bare-metal environment. But this symbol is recommended in case a future implementation of OSIF will depend on it.

Mutex operations are dummy operations (always return success) and semaphore is implemented as a simple counter.

Hardware resources

Bare-metal OSIF uses a hardware timer to accurately measure time. The timer and channel used are platform-dependent, are chosen to be the same as the FreeRTOS implementation if possible.

The table below shows which timers and channels are used on each platform:

Platform	Timer	Channel
S32K11x	Systick	N/A
S32K14x	Systick	N/A
MPC5746C	PIT	15
MPC5748G	PIT	15
MPC5744P	PIT	3

Bare-metal timing limitations

For bare-metal OSIF, the timer is initialized at the first call in OSIF that needs timing. That is either OSIF_Time↵Delay, OSIF_MutexLock or OSIF_SemaWait (functions with timeout). The timer configuration, but not the counter, is updated at each subsequent call to these functions.

Do not assume OSIF_GetMilliseconds will return a global value since system initialization. It will return the global value since the very first timer initialization, mentioned above.

Macros

- #define OSIF_WAIT_FOREVER 0xFFFFFFFFu

Functions

- void OSIF_TimeDelay (const uint32_t delay)

    Delays execution for a number of milliseconds.
- uint32_t OSIF_GetMilliseconds (void)

    Returns the number of miliseconds elapsed since starting the internal timer or starting the scheduler.
- status_t OSIF_MutexLock (const mutex_t ∗const pMutex, const uint32_t timeout)

    Waits for a mutex and locks it.
- status_t OSIF_MutexUnlock (const mutex_t ∗const pMutex)

    Unlocks a previously locked mutex.
- status_t OSIF_MutexCreate (mutex_t ∗const pMutex)

    Create an unlocked mutex.
- status_t OSIF_MutexDestroy (const mutex_t ∗const pMutex)

    Destroys a previously created mutex.
- status_t OSIF_SemaWait (semaphore_t ∗const pSem, const uint32_t timeout)

    Decrement a semaphore with timeout.
- status_t OSIF_SemaPost (semaphore_t ∗const pSem)

    Increment a semaphore.
- status_t OSIF_SemaCreate (semaphore_t ∗const pSem, const uint8_t initValue)

    Creates a semaphore with a given value.
- status_t OSIF_SemaDestroy (const semaphore_t ∗const pSem)

    Destroys a previously created semaphore.

16.67.2   Macro Definition Documentation

16.67.2.1   #define OSIF_WAIT_FOREVER 0xFFFFFFFFu

Definition at line 65 of file osif.h.

16.67.3   Function Documentation

16.67.3.1   uint32_t OSIF_GetMilliseconds (void)

Returns the number of miliseconds elapsed since starting the internal timer or starting the scheduler.

Returns

the number of miliseconds elapsed

Definition at line 228 of file osif_baremetal.c.

16.67.3.2   status_t OSIF_MutexCreate (mutex_t ∗const pMutex)

Create an unlocked mutex.

Parameters

in	pMutex	reference to the mutex object

Returns

One of the possible status codes:

- STATUS_SUCCESS: mutex created
- STATUS_ERROR: mutex could not be created

Definition at line 276 of file osif_baremetal.c.

16.67.3.3   status_t OSIF_MutexDestroy (const mutex_t ∗const pMutex)

Destroys a previously created mutex.

Parameters

in	pMutex	reference to the mutex object

Returns

One of the possible status codes:

- STATUS_SUCCESS: mutex destroyed

Definition at line 290 of file osif_baremetal.c.

16.67.3.4   status_t OSIF_MutexLock (const mutex_t ∗const pMutex, const uint32_t timeout)

Waits for a mutex and locks it.

Parameters

in	pMutex	reference to the mutex object
in	timeout	time-out value in milliseconds

Returns

One of the possible status codes:

- STATUS_SUCCESS: mutex lock operation success
- STATUS_ERROR: mutex already owned by current thread
- STATUS_TIMEOUT: mutex lock operation timed out

Definition at line 246 of file osif_baremetal.c.

16.67.3.5   status_t OSIF_MutexUnlock (const mutex_t ∗const *pMutex*)

Unlocks a previously locked mutex.

Parameters

in	*pMutex*	reference to the mutex object

Returns

One of the possible status codes:
- STATUS_SUCCESS: mutex unlock operation success
- STATUS_ERROR: mutex unlock failed

Definition at line 262 of file osif_baremetal.c.

16.67.3.6 status_t OSIF_SemaCreate (semaphore_t ∗const *pSem,* const uint8_t *initValue*)

Creates a semaphore with a given value.

Parameters

in	*pSem*	reference to the semaphore object
in	*initValue*	initial value of the semaphore

Returns

One of the possible status codes:
- STATUS_SUCCESS: semaphore created
- STATUS_ERROR: semaphore could not be created

Definition at line 382 of file osif_baremetal.c.

16.67.3.7 status_t OSIF_SemaDestroy (const semaphore_t ∗const *pSem*)

Destroys a previously created semaphore.

Parameters

in	*pSem*	reference to the semaphore object

Returns

One of the possible status codes:
- STATUS_SUCCESS: semaphore destroyed

Definition at line 400 of file osif_baremetal.c.

16.67.3.8 status_t OSIF_SemaPost (semaphore_t ∗const *pSem*)

Increment a semaphore.

Parameters

in	*pSem*	reference to the semaphore object

Returns

One of the possible status codes:
- STATUS_SUCCESS: semaphore post operation success
- STATUS_ERROR: semaphore could not be incremented

Definition at line 355 of file osif_baremetal.c.

16.67.3.9 status_t OSIF_SemaWait (semaphore_t ∗const *pSem,* const uint32_t *timeout*)

Decrement a semaphore with timeout.

Parameters

in	*pSem*	reference to the semaphore object
in	*timeout*	time-out value in milliseconds

Returns

One of the possible status codes:

- STATUS_SUCCESS: semaphore wait operation success
- STATUS_TIMEOUT: semaphore wait timed out

Definition at line 304 of file osif_baremetal.c.

16.67.3.10   void OSIF_TimeDelay (const uint32_t *delay*)

Delays execution for a number of milliseconds.

Parameters

in	*delay*	Time delay in milliseconds.

Definition at line 203 of file osif_baremetal.c.

16.68 PDB Driver

16.68.1 Detailed Description

Programmable Delay Block Peripheral Driver.

Overview

This section describes the programming interface of the PDB Peripheral driver. The PDB peripheral driver configures the PDB (Programmable Delay Block). It handles the triggers for ADC and pulse out to the CMP and the PDB counter.

PDB Driver model building

There is one main PDB counter for all triggers. When the indicated external trigger input arrives, the PDB counter launches and is increased by setting clock. The counter trigger milestones for ADC and the PDB counter and wait for the PDB counter. Once the PDB counter hits each milestone, also called the critical delay value, the corresponding event is triggered and the trigger signal is sent out to trigger other peripherals. Therefore, the PDB module is a collector and manager of triggers.

PDB Initialization

The core feature of the PDB module is a programmable timer/counter. Additional features enable and set the milestone for the corresponding trigger. The user should provide a configuration suitable for the application requirements. Call the API of **PDB_DRV_Init()** function to initialize the PDB timer/counter.

All triggers share the same counter.

The basic timing/counting step is set when initializing the main PDB counter:

The basic timing/counting step = F_BusClkHz / pdb_timer_config_t.clkPreDiv / pdb_timer_config_t.clkPreMultFactor

The F_BusClkHz is the frequency of bus clock in Hertz. The "clkPreDiv" and "clkPreMultFactor" are in the pdb_↩ timer_config_t structure. All triggering milestones are based on this step.

PDB Call diagram

Three kinds of typical use cases are designed for the PDB module.

- Normal Timer/Counter. Normal Timer/Counter is the basic case. The Timer/Counter starts after the PDB is initialized and the milestone for the PDB Timer/Counter is set. After it is triggered and when the counter hits the milestone, the interrupt request occurs if enabled. In continuous mode, when the counter hits the upper limit, it returns zero and counts again.

- Trigger for ADC module. When the ADC trigger is enabled, a delay value for ADC trigger is set as the milestone. At least two ADC channel groups are provided. Likewise, there are more than two pre-triggers for ADC. Each pre-trigger is related to one channel group and can be enabled separately in the PDB module. When the PDB counter hits the milestone for the ADC pre-trigger, it triggers the ADC's conversion on the indicated channel group. To maximize the feature, the ADC should be configured to enable the hardware trigger mode.

- Trigger for pulse out to the CMP module. The pulse-out trigger is attached to the main PDB counter. There are two milestones for each pulse out channel, a milestone for level high and for level low, which makes a sample window for the CMP module.

These are the examples to initialize and configure the PDB driver for typical use cases.

Normal Timer/Counter:

```
#define PDB_INSTANCE    0UL

static volatile uint32_t gPdbIntCounter = 0U;
static volatile uint32_t gPdbInstance = 0U;
static void PDB_ISR_Counter(void);
```

```
void PDB_TEST_NormalTimer(uint32_t instance)
{
    pdb_timer_config_t PdbTimerConfig;
    PdbTimerConfig.loadValueMode        = PDB_LOAD_VAL_IMMEDIATELY;
    PdbTimerConfig.seqErrIntEnable      = false;
    PdbTimerConfig.clkPreDiv            = PDB_CLK_PREDIV_BY_8;
    PdbTimerConfig.clkPreMultFactor     =
      PDB_CLK_PREMULT_FACT_AS_40;
    PdbTimerConfig.triggerInput         = PDB_SOFTWARE_TRIGGER;
    PdbTimerConfig.continuousModeEnable = true;
    PdbTimerConfig.dmaEnable            = false;
    PdbTimerConfig.intEnable            = true;
    PDB_DRV_Init(instance, &PdbTimerConfig);
    PDB_DRV_SetTimerModulusValue(instance, 0xFFFU);
    PDB_DRV_SetValueForTimerInterrupt(instance, 0xFFU);
    PDB_DRV_LoadValuesCmd(instance);
    gPdbIntCounter = 0U;
    gPdbInstance = instance;
    PDB_DRV_SoftTriggerCmd(instance);
    while (gPdbIntCounter < 20U) {}
    PRINTF("PDB Timer's delay interrupt generated.\r\n");
    PDB_DRV_Deinit(instance);
    PRINTF("OK.\r\n");
}

void PDB_IRQHandler()
{
    PDB_DRV_ClearTimerIntFlag(PDB_INSTANCE);
    if (gPdbIntCounter >= 0xFFFFU)
    {
        gPdbIntCounter = 0U;
    }
    else
    {
        gPdbIntCounter++;
    }
}

#if PDB_INSTANCE < 1
    void PDB0_IRQHandler(void)
    {
        PDB_IRQHandler();
    }
#elif PDB_INSTANCE < 2
    void PDB1_IRQHandler(void)
    {
        PDB_IRQHandler();
    }
#endif
```

Trigger for ADC module:

```
void PDB_TEST_AdcPreTrigger(uint32_t instance)
{
    pdb_timer_config_t PdbTimerConfig;
    pdb_adc_pretrigger_config_t PdbAdcPreTriggerConfig;
    PdbTimerConfig.loadValueMode        = PDB_LOAD_VAL_IMMEDIATELY;
    PdbTimerConfig.seqErrIntEnable      = false;
    PdbTimerConfig.clkPreDiv            = PDB_CLK_PREDIV_BY_8;
    PdbTimerConfig.clkPreMultFactor     =
      PDB_CLK_PREMULT_FACT_AS_40;
    PdbTimerConfig.triggerInput         = PDB_SOFTWARE_TRIGGER;
    PdbTimerConfig.continuousModeEnable = false;
    PdbTimerConfig.dmaEnable            = false;
    PdbTimerConfig.intEnable            = false;
    PDB_DRV_Init(instance, &PdbTimerConfig);

    PdbAdcPreTriggerConfig.adcPreTriggerIdx          = 0U;
    PdbAdcPreTriggerConfig.preTriggerEnable          = true;
    PdbAdcPreTriggerConfig.preTriggerOutputEnable    = true;
    PdbAdcPreTriggerConfig.preTriggerBackToBackEnable = false;
    PDB_DRV_ConfigAdcPreTrigger(instance, 0U, &PdbAdcPreTriggerConfig);

    PDB_DRV_SetTimerModulusValue(instance, 0xFFFU);
    PDB_DRV_SetAdcPreTriggerDelayValue(instance, 0U, 0U, 0xFFU);
    PDB_DRV_LoadValuesCmd(instance);
    PDB_DRV_SoftTriggerCmd(instance);
    while (1U != PDB_DRV_GetAdcPreTriggerFlags(instance, 0U, 1U)) {}
    PDB_DRV_ClearAdcPreTriggerFlags(instance, 0U, 1U);
    PRINTF("PDB ADC PreTrigger generated.\r\n");
    PDB_DRV_Deinit(instance);
    PRINTF("OK.\r\n");
}
```

Data Structures

- struct pdb_timer_config_t

  Defines the type of structure for basic timer in PDB. More...
- struct pdb_adc_pretrigger_config_t

  Defines the type of structure for configuring ADC's pre_trigger. More...

Enumerations

- enum pdb_load_value_mode_t { PDB_LOAD_VAL_IMMEDIATELY = 0U, PDB_LOAD_VAL_AT_MODUL↩
  O_COUNTER = 1U, PDB_LOAD_VAL_AT_NEXT_TRIGGER = 2U, PDB_LOAD_VAL_AT_MODULO_CO↩
  UNTER_OR_NEXT_TRIGGER = 3U }

  Defines the type of value load mode for the PDB module.
- enum pdb_clk_prescaler_div_t {
  PDB_CLK_PREDIV_BY_1 = 0U, PDB_CLK_PREDIV_BY_2 = 1U, PDB_CLK_PREDIV_BY_4 = 2U, PDB_↩
  CLK_PREDIV_BY_8 = 3U,
  PDB_CLK_PREDIV_BY_16 = 4U, PDB_CLK_PREDIV_BY_32 = 5U, PDB_CLK_PREDIV_BY_64 = 6U, P↩
  DB_CLK_PREDIV_BY_128 = 7U }

  Defines the type of prescaler divider for the PDB counter clock. Implements : pdb_clk_prescaler_div_t_Class.
- enum pdb_trigger_src_t { PDB_TRIGGER_IN0 = 0U, PDB_SOFTWARE_TRIGGER = 15U }

  Defines the type of trigger source mode for the PDB.
- enum pdb_clk_prescaler_mult_factor_t { PDB_CLK_PREMULT_FACT_AS_1 = 0U, PDB_CLK_PREMUL↩
  T_FACT_AS_10 = 1U, PDB_CLK_PREMULT_FACT_AS_20 = 2U, PDB_CLK_PREMULT_FACT_AS_40 =
  3U }

  Defines the type of the multiplication source mode for PDB.

Functions

- void PDB_DRV_Init (const uint32_t instance, const pdb_timer_config_t *userConfigPtr)

  Initializes the PDB counter and triggers input.
- void PDB_DRV_Deinit (const uint32_t instance)

  De-initializes the PDB module.
- void PDB_DRV_GetDefaultConfig (pdb_timer_config_t *const config)

  Gets the default configuration structure of PDB with default settings.
- void PDB_DRV_Enable (const uint32_t instance)

  Enables the PDB module.
- void PDB_DRV_Disable (const uint32_t instance)

  Disables the PDB module.
- void PDB_DRV_SoftTriggerCmd (const uint32_t instance)

  Triggers the PDB with a software trigger.
- uint32_t PDB_DRV_GetTimerValue (const uint32_t instance)

  Gets the current counter value in the PDB module.
- bool PDB_DRV_GetTimerIntFlag (const uint32_t instance)

  Gets the PDB interrupt flag.
- void PDB_DRV_ClearTimerIntFlag (const uint32_t instance)

  Clears the interrupt flag.
- void PDB_DRV_LoadValuesCmd (const uint32_t instance)

  Executes the command of loading values.
- void PDB_DRV_SetTimerModulusValue (const uint32_t instance, const uint16_t value)

  Sets the value of timer modulus.
- void PDB_DRV_SetValueForTimerInterrupt (const uint32_t instance, const uint16_t value)

  Sets the value for the timer interrupt.

- void PDB_DRV_ConfigAdcPreTrigger (const uint32_t instance, const uint32_t chn, const pdb_adc_↩
pretrigger_config_t ∗configPtr)

  Configures the ADC pre_trigger in the PDB module.

- uint32_t PDB_DRV_GetAdcPreTriggerFlags (const uint32_t instance, const uint32_t chn, const uint32_↩
t preChnMask)

  Gets the ADC pre_trigger flag in the PDB module.

- void PDB_DRV_ClearAdcPreTriggerFlags (const uint32_t instance, const uint32_t chn, const uint32_t pre↩
ChnMask)

  Clears the ADC pre_trigger flag in the PDB module.

- uint32_t PDB_DRV_GetAdcPreTriggerSeqErrFlags (const uint32_t instance, const uint32_t chn, const
uint32_t preChnMask)

  Gets the ADC pre_trigger flag in the PDB module.

- void PDB_DRV_ClearAdcPreTriggerSeqErrFlags (const uint32_t instance, const uint32_t chn, const uint32↩
_t preChnMask)

  Clears the ADC pre_trigger flag in the PDB module.

- void PDB_DRV_SetAdcPreTriggerDelayValue (const uint32_t instance, const uint32_t chn, const uint32_t
preChn, const uint32_t value)

  Sets the ADC pre_trigger delay value in the PDB module.

- void PDB_DRV_SetCmpPulseOutEnable (const uint32_t instance, const uint32_t pulseChnMask, bool en-
able)

  Switches on/off the CMP pulse out in the PDB module.

- void PDB_DRV_SetCmpPulseOutDelayForHigh (const uint32_t instance, const uint32_t pulseChn, const
uint32_t value)

  Sets the CMP pulse out delay value for high in the PDB module.

- void PDB_DRV_SetCmpPulseOutDelayForLow (const uint32_t instance, const uint32_t pulseChn, const
uint32_t value)

  Sets the CMP pulse out delay value for low in the PDB module.

16.68.2   Data Structure Documentation

16.68.2.1   struct pdb_timer_config_t

Defines the type of structure for basic timer in PDB.

Definition at line 106 of file pdb_driver.h.

Data Fields

- pdb_load_value_mode_t loadValueMode
- bool seqErrIntEnable
- pdb_clk_prescaler_div_t clkPreDiv
- pdb_clk_prescaler_mult_factor_t clkPreMultFactor
- pdb_trigger_src_t triggerInput
- bool continuousModeEnable
- bool dmaEnable
- bool intEnable

Field Documentation

16.68.2.1.1   pdb_clk_prescaler_div_t clkPreDiv

Select the prescaler divider.

Definition at line 110 of file pdb_driver.h.

16.68.2.1.2   pdb_clk_prescaler_mult_factor_t clkPreMultFactor

Select multiplication factor for prescaler.

Definition at line 111 of file pdb_driver.h.

16.68.2.1.3   bool continuousModeEnable

Enable the continuous mode.

Definition at line 113 of file pdb_driver.h.

16.68.2.1.4   bool dmaEnable

Enable the dma for timer.

Definition at line 114 of file pdb_driver.h.

16.68.2.1.5   bool intEnable

Enable the interrupt for timer.

Definition at line 115 of file pdb_driver.h.

16.68.2.1.6   pdb_load_value_mode_t loadValueMode

Select the load mode.

Definition at line 108 of file pdb_driver.h.

16.68.2.1.7   bool seqErrIntEnable

Enable PDB Sequence Error Interrupt.

Definition at line 109 of file pdb_driver.h.

16.68.2.1.8   pdb_trigger_src_t triggerInput

Select the trigger input source.

Definition at line 112 of file pdb_driver.h.

16.68.2.2   struct pdb_adc_pretrigger_config_t

Defines the type of structure for configuring ADC's pre_trigger.

Definition at line 123 of file pdb_driver.h.

Data Fields

- uint32_t adcPreTriggerIdx
- bool preTriggerEnable
- bool preTriggerOutputEnable
- bool preTriggerBackToBackEnable

Field Documentation

16.68.2.2.1   uint32_t adcPreTriggerIdx

Setting pre_trigger's index.

Definition at line 125 of file pdb_driver.h.

16.68.2.2.2   bool preTriggerBackToBackEnable

Enable the back to back mode for ADC pre_trigger.

Definition at line 128 of file pdb_driver.h.

16.68.2.2.3   bool preTriggerEnable

Enable the pre_trigger.

Definition at line 126 of file pdb_driver.h.

16.68.2.2.4   bool preTriggerOutputEnable

Enable the pre_trigger output.

Definition at line 127 of file pdb_driver.h.

16.68.3   Enumeration Type Documentation

16.68.3.1   enum pdb_clk_prescaler_div_t

Defines the type of prescaler divider for the PDB counter clock. Implements : pdb_clk_prescaler_div_t_Class.

Enumerator

> **PDB_CLK_PREDIV_BY_1**   Counting divided by multiplication factor selected by MULT.
> **PDB_CLK_PREDIV_BY_2**   Counting divided by multiplication factor selected by 2 times ofMULT.
> **PDB_CLK_PREDIV_BY_4**   Counting divided by multiplication factor selected by 4 times ofMULT.
> **PDB_CLK_PREDIV_BY_8**   Counting divided by multiplication factor selected by 8 times ofMULT.
> **PDB_CLK_PREDIV_BY_16**   Counting divided by multiplication factor selected by 16 times ofMULT.
> **PDB_CLK_PREDIV_BY_32**   Counting divided by multiplication factor selected by 32 times ofMULT.
> **PDB_CLK_PREDIV_BY_64**   Counting divided by multiplication factor selected by 64 times ofMULT.
> **PDB_CLK_PREDIV_BY_128**   Counting divided by multiplication factor selected by 128 times ofMULT.

Definition at line 61 of file pdb_driver.h.

16.68.3.2   enum pdb_clk_prescaler_mult_factor_t

Defines the type of the multiplication source mode for PDB.

Selects the multiplication factor of the prescaler divider for the PDB counter clock. Implements : pdb_clk_$\hookleftarrow$ prescaler_mult_factor_t_Class

Enumerator

> **PDB_CLK_PREMULT_FACT_AS_1**   Multiplication factor is 1.
> **PDB_CLK_PREMULT_FACT_AS_10**   Multiplication factor is 10.
> **PDB_CLK_PREMULT_FACT_AS_20**   Multiplication factor is 20.
> **PDB_CLK_PREMULT_FACT_AS_40**   Multiplication factor is 40.

Definition at line 92 of file pdb_driver.h.

16.68.3.3   enum pdb_load_value_mode_t

Defines the type of value load mode for the PDB module.

Some timing related registers, such as the MOD, IDLY, CHnDLYm, INTx and POyDLY, buffer the setting values. Only the load operation is triggered. The setting value is loaded from a buffer and takes effect. There are four loading modes to fit different applications. Implements : pdb_load_value_mode_t_Class

Enumerator

> **PDB_LOAD_VAL_IMMEDIATELY**   Loaded immediately after load operation.

> **PDB_LOAD_VAL_AT_MODULO_COUNTER** Loaded when counter hits the modulo after load operation.
>
> **PDB_LOAD_VAL_AT_NEXT_TRIGGER** Loaded when detecting an input trigger after load operation.
>
> **PDB_LOAD_VAL_AT_MODULO_COUNTER_OR_NEXT_TRIGGER** Loaded when counter hits the modulo or detecting an input trigger after load operation.

Definition at line 45 of file pdb_driver.h.

16.68.3.4   enum pdb_trigger_src_t

Defines the type of trigger source mode for the PDB.

Selects the trigger input source for the PDB. The trigger input source can be internal or the software trigger. Implements : pdb_trigger_src_t_Class

Enumerator

> **PDB_TRIGGER_IN0** Source trigger comes from TRGMUX.
>
> **PDB_SOFTWARE_TRIGGER** Select software trigger.

Definition at line 80 of file pdb_driver.h.

16.68.4   Function Documentation

16.68.4.1   void PDB_DRV_ClearAdcPreTriggerFlags (const uint32_t *instance,* const uint32_t *chn,* const uint32_t *preChnMask*)

Clears the ADC pre_trigger flag in the PDB module.

This function clears the ADC pre_trigger flags in the PDB module.

Parameters

in	*instance*	PDB instance ID.
in	*chn*	ADC channel.
in	*preChnMask*	ADC pre_trigger channels mask.

Definition at line 335 of file pdb_driver.c.

16.68.4.2   void PDB_DRV_ClearAdcPreTriggerSeqErrFlags (const uint32_t *instance,* const uint32_t *chn,* const uint32_t *preChnMask*)

Clears the ADC pre_trigger flag in the PDB module.

This function clears the ADC pre_trigger sequence error flags in the PDB module.

Parameters

in	*instance*	PDB instance ID.
in	*chn*	ADC channel.
in	*preChnMask*	ADC pre_trigger channels mask.

Definition at line 371 of file pdb_driver.c.

16.68.4.3   void PDB_DRV_ClearTimerIntFlag (const uint32_t *instance*)

Clears the interrupt flag.

This function clears the interrupt flag.

Parameters

in	*instance*	PDB instance ID.

Definition at line 234 of file pdb_driver.c.

16.68.4.4   void PDB_DRV_ConfigAdcPreTrigger (const uint32_t *instance,* const uint32_t *chn,* const pdb_adc_pretrigger_config_t ∗ *configPtr*)

Configures the ADC pre_trigger in the PDB module.

This function configures the ADC pre_trigger in the PDB module.

Parameters

in	*instance*	PDB instance ID.
in	*chn*	ADC channel.
in	*configPtr*	Pointer to the user configuration structure. See the "pdb_adc_pretrigger_↵config_t".

Definition at line 296 of file pdb_driver.c.

16.68.4.5   void PDB_DRV_Deinit (const uint32_t *instance*)

De-initializes the PDB module.

This function de-initializes the PDB module. Calling this function shuts down the PDB module and reduces the power consumption.

Parameters

in	*instance*	PDB instance ID.

Definition at line 111 of file pdb_driver.c.

16.68.4.6   void PDB_DRV_Disable (const uint32_t *instance*)

Disables the PDB module.

This function disables the PDB module, counter is off also.

Parameters

in	*instance*	PDB instance ID.

Definition at line 171 of file pdb_driver.c.

16.68.4.7   void PDB_DRV_Enable (const uint32_t *instance*)

Enables the PDB module.

This function enables the PDB module, counter is on.

Parameters

in	*instance*	PDB instance ID.

Definition at line 156 of file pdb_driver.c.

16.68.4.8   uint32_t PDB_DRV_GetAdcPreTriggerFlags (const uint32_t *instance,* const uint32_t *chn,* const uint32_t *preChnMask*)

Gets the ADC pre_trigger flag in the PDB module.

This function gets the ADC pre_trigger flags in the PDB module.

Parameters

in	*instance*	PDB instance ID.
in	*chn*	ADC channel.
in	*preChnMask*	ADC pre_trigger channels mask.

Returns

Assertion of indicated flag.

Definition at line 317 of file pdb_driver.c.

16.68.4.9 uint32_t PDB_DRV_GetAdcPreTriggerSeqErrFlags (const uint32_t *instance,* **const uint32_t** *chn,* **const uint32_t** *preChnMask* **)**

Gets the ADC pre_trigger flag in the PDB module.

This function gets the ADC pre_trigger flags in the PDB module.

Parameters

in	*instance*	PDB instance ID.
in	*chn*	ADC channel.
in	*preChnMask*	ADC pre_trigger channels mask.

Returns

Assertion of indicated flag.

Definition at line 353 of file pdb_driver.c.

16.68.4.10 void PDB_DRV_GetDefaultConfig (pdb_timer_config_t ∗**const** *config* **)**

Gets the default configuration structure of PDB with default settings.

This function initializes the hardware configuration structure to default values (Reference Manual Resets). This function should be called before configuring the hardware feature by PDB_DRV_Init() function, otherwise all members be written by user. This function insures that all members are written with safe values, but the user still can modify the desired members.

Parameters

out	*config*	Pointer to PDB configuration structure.

Definition at line 128 of file pdb_driver.c.

16.68.4.11 bool PDB_DRV_GetTimerIntFlag (const uint32_t *instance* **)**

Gets the PDB interrupt flag.

This function gets the PDB interrupt flag. It is asserted if the PDB interrupt occurs.

Parameters

in	*instance*	PDB instance ID.

Returns

Assertion of indicated event.

Definition at line 219 of file pdb_driver.c.

16.68.4.12 uint32_t PDB_DRV_GetTimerValue (const uint32_t *instance* **)**

Gets the current counter value in the PDB module.

This function gets the current counter value.

Parameters

in	*instance*	PDB instance ID.

Returns

> Current PDB counter value.

Definition at line 203 of file pdb_driver.c.

16.68.4.13   void PDB_DRV_Init (const uint32_t *instance,* const pdb_timer_config_t ∗ *userConfigPtr*)

Initializes the PDB counter and triggers input.

This function initializes the PDB counter and triggers the input. It resets PDB registers and enables the PDB clock. Therefore, it should be called before any other operation. After it is initialized, the PDB can act as a triggered timer, which enables other features in PDB module.

Parameters

in	*instance*	PDB instance ID.
in	*userConfigPtr*	Pointer to the user configuration structure. See the "pdb_user_config_t".

Definition at line 63 of file pdb_driver.c.

16.68.4.14   void PDB_DRV_LoadValuesCmd (const uint32_t *instance*)

Executes the command of loading values.

This function executes the command of loading values.

Parameters

in	*instance*	PDB instance ID.

Definition at line 249 of file pdb_driver.c.

16.68.4.15   void PDB_DRV_SetAdcPreTriggerDelayValue (const uint32_t *instance,* const uint32_t *chn,* const uint32_t *preChn,* const uint32_t *value*)

Sets the ADC pre_trigger delay value in the PDB module.

This function sets Set the ADC pre_trigger delay value in the PDB module.

Parameters

instance	PDB instance ID.
chn	ADC channel.
preChn	ADC pre_channel.
value	Setting value.

Definition at line 389 of file pdb_driver.c.

16.68.4.16   void PDB_DRV_SetCmpPulseOutDelayForHigh (const uint32_t *instance,* const uint32_t *pulseChn,* const uint32_t *value*)

Sets the CMP pulse out delay value for high in the PDB module.

This function sets the CMP pulse out delay value for high in the PDB module.

Parameters

in	*instance*	PDB instance ID.
in	*pulseChn*	Pulse channel.
in	*value*	Setting value.

Definition at line 426 of file pdb_driver.c.

16.68.4.17   void PDB_DRV_SetCmpPulseOutDelayForLow (const uint32_t *instance,* const uint32_t *pulseChn,* const uint32_t *value*)

Sets the CMP pulse out delay value for low in the PDB module.

This function sets the CMP pulse out delay value for low in the PDB module.

Parameters

in	*instance*	PDB instance ID.
in	*pulseChn*	Pulse channel.
in	*value*	Setting value.

Definition at line 444 of file pdb_driver.c.

16.68.4.18   void PDB_DRV_SetCmpPulseOutEnable (const uint32_t *instance,* const uint32_t *pulseChnMask,* bool *enable*)

Switches on/off the CMP pulse out in the PDB module.

This function switches the CMP pulse on/off in the PDB module.

Parameters

in	*instance*	PDB instance ID.
in	*pulseChnMask*	Pulse channel mask.
in	*enable*	Switcher to assert the feature.

Definition at line 409 of file pdb_driver.c.

16.68.4.19   void PDB_DRV_SetTimerModulusValue (const uint32_t *instance,* const uint16_t *value*)

Sets the value of timer modulus.

This function sets the value of timer modulus.

Parameters

in	*instance*	PDB instance ID.
in	*value*	Setting value.

Definition at line 264 of file pdb_driver.c.

16.68.4.20   void PDB_DRV_SetValueForTimerInterrupt (const uint32_t *instance,* const uint16_t *value*)

Sets the value for the timer interrupt.

This function sets the value for the timer interrupt.

Parameters

in	*instance*	PDB instance ID.
in	*value*	Setting value.

Definition at line 280 of file pdb_driver.c.

16.68.4.21   void PDB_DRV_SoftTriggerCmd (const uint32_t *instance*)

Triggers the PDB with a software trigger.

This function triggers the PDB with a software trigger. When the PDB is set to use the software trigger as input, calling this function triggers the PDB.

Parameters

in	*instance*	PDB instance ID.

Definition at line 188 of file pdb_driver.c.

16.69 PINS Driver

16.69.1 Detailed Description

This section describes the programming interface of the PINS driver.

Data Structures

- struct pin_settings_config_t

  Defines the converter configuration. More...

Typedefs

- typedef uint8_t pins_level_type_t

  Type of a port levels representation. Implements : pins_level_type_t_Class.

Enumerations

- enum port_data_direction_t { GPIO_INPUT_DIRECTION = 0x0U, GPIO_OUTPUT_DIRECTION = 0x1U, G↩
  PIO_UNSPECIFIED_DIRECTION = 0x2U }

  Configures the port data direction Implements : port_data_direction_t_Class.

PINS DRIVER API.

- status_t PINS_DRV_Init (uint32_t pinCount, const pin_settings_config_t config[ ])

  Initializes the pins with the given configuration structure.
- void PINS_DRV_WritePin (GPIO_Type ∗const base, pins_channel_type_t pin, pins_level_type_t value)

  Write a pin of a port with a given value.
- void PINS_DRV_WritePins (GPIO_Type ∗const base, pins_channel_type_t pins)

  Write all pins of a port.
- pins_channel_type_t PINS_DRV_GetPinsOutput (const GPIO_Type ∗const base)

  Get the current output from a port.
- void PINS_DRV_SetPins (GPIO_Type ∗const base, pins_channel_type_t pins)

  Write pins with 'Set' value.
- void PINS_DRV_ClearPins (GPIO_Type ∗const base, pins_channel_type_t pins)

  Write pins to 'Clear' value.
- void PINS_DRV_TogglePins (GPIO_Type ∗const base, pins_channel_type_t pins)

  Toggle pins value.
- pins_channel_type_t PINS_DRV_ReadPins (const GPIO_Type ∗const base)

  Read input pins.

16.69.2 Data Structure Documentation

16.69.2.1 struct pin_settings_config_t

Defines the converter configuration.

This structure is used to configure the pins Implements : pin_settings_config_t_Class

Definition at line 560 of file pins_driver.h.

Data Fields

- uint32_t pinPortIdx
- port_mux_t mux
    - *Pin (C55: Out) mux selection.*
- GPIO_Type ∗ gpioBase
- port_data_direction_t direction
- bool digitalFilter
- pins_level_type_t initValue

Field Documentation

16.69.2.1.1   bool digitalFilter

Enables digital filter.

Definition at line 597 of file pins_driver.h.

16.69.2.1.2   port_data_direction_t direction

Configures the port data direction.

Definition at line 596 of file pins_driver.h.

16.69.2.1.3   GPIO_Type∗ gpioBase

GPIO base pointer.

Definition at line 595 of file pins_driver.h.

16.69.2.1.4   pins_level_type_t initValue

Initial value

Definition at line 633 of file pins_driver.h.

16.69.2.1.5   port_mux_t mux

Pin (C55: Out) mux selection.

Definition at line 583 of file pins_driver.h.

16.69.2.1.6   uint32_t pinPortIdx

Port pin number.

Definition at line 567 of file pins_driver.h.

16.69.3   Typedef Documentation

16.69.3.1   typedef uint8_t pins_level_type_t

Type of a port levels representation. Implements : pins_level_type_t_Class.

Definition at line 56 of file pins_driver.h.

16.69.4   Enumeration Type Documentation

16.69.4.1   enum port_data_direction_t

Configures the port data direction Implements : port_data_direction_t_Class.

Enumerator

> ***GPIO_INPUT_DIRECTION***   General purpose input direction.
>
> ***GPIO_OUTPUT_DIRECTION***   General purpose output direction.
>
> ***GPIO_UNSPECIFIED_DIRECTION***   General purpose unspecified direction.

Definition at line 62 of file pins_driver.h.

16.69.5   Function Documentation

16.69.5.1   void PINS_DRV_ClearPins (GPIO_Type ∗const *base,* pins_channel_type_t *pins*)

Write pins to 'Clear' value.

This function configures output pins listed in parameter pins (bits that are '1') to have a 'cleared' value (LOW). Pins corresponding to '0' will be unaffected.

Parameters

in	*base*	GPIO base pointer (PTA, PTB, PTC, etc.)
in	*pins*	Pin mask of bits to be cleared. Each bit represents one pin (LSB is pin 0, MSB is pin 31). For each bit: • 0: corresponding pin is unaffected • 1: corresponding pin is cleared(set to LOW)

Definition at line 523 of file pins_driver.c.

16.69.5.2   pins_channel_type_t PINS_DRV_GetPinsOutput (const GPIO_Type ∗const *base*)

Get the current output from a port.

This function returns the current output that is written to a port. Only pins that are configured as output will have meaningful values.

Parameters

in	*base*	GPIO base pointer (PTA, PTB, PTC, etc.)

Returns

> GPIO outputs. Each bit represents one pin (LSB is pin 0, MSB is pin 31). For each bit:
> • 0: corresponding pin is set to LOW
> • 1: corresponding pin is set to HIGH

Definition at line 494 of file pins_driver.c.

16.69.5.3   status_t PINS_DRV_Init (uint32_t *pinCount,* const pin_settings_config_t *config[ ]*)

Initializes the pins with the given configuration structure.

This function configures the pins with the options provided in the provided structure.

Parameters

in	*pinCount*	The number of configured pins in structure

in	*config*	The configuration structure

Returns

    The status of the operation

Definition at line 53 of file pins_driver.c.

16.69.5.4 pins_channel_type_t PINS_DRV_ReadPins (const GPIO_Type ∗const *base*)

Read input pins.

This function returns the current input values from a port. Only pins configured as input will have meaningful values.

Parameters

in	*base*	GPIO base pointer (PTA, PTB, PTC, etc.)

Returns

    GPIO inputs. Each bit represents one pin (LSB is pin 0, MSB is pin 31). For each bit:

       • 0: corresponding pin is read as LOW

       • 1: corresponding pin is read as HIGH

Definition at line 551 of file pins_driver.c.

16.69.5.5 void PINS_DRV_SetPins (GPIO_Type ∗const *base,* pins_channel_type_t *pins*)

Write pins with 'Set' value.

This function configures output pins listed in parameter pins (bits that are '1') to have a value of 'set' (HIGH). Pins corresponding to '0' will be unaffected.

Parameters

in	*base*	GPIO base pointer (PTA, PTB, PTC, etc.)
in	*pins*	Pin mask of bits to be set. Each bit represents one pin (LSB is pin 0, MSB is pin 31). For each bit: • 0: corresponding pin is unaffected • 1: corresponding pin is set to HIGH

Definition at line 508 of file pins_driver.c.

16.69.5.6 void PINS_DRV_TogglePins (GPIO_Type ∗const *base,* pins_channel_type_t *pins*)

Toggle pins value.

This function toggles output pins listed in parameter pins (bits that are '1'). Pins corresponding to '0' will be unaffected.

Parameters

in	*base*	GPIO base pointer (PTA, PTB, PTC, etc.)
in	*pins*	Pin mask of bits to be toggled. Each bit represents one pin (LSB is pin 0, MSB is pin 31). For each bit: • 0: corresponding pin is unaffected • 1: corresponding pin is toggled

Definition at line 537 of file pins_driver.c.

16.69.5.7   void PINS_DRV_WritePin (GPIO_Type ∗const *base,* pins_channel_type_t *pin,* pins_level_type_t *value*)

Write a pin of a port with a given value.

This function writes the given pin from a port, with the given value ('0' represents LOW, '1' represents HIGH).

Parameters

in	*base*	GPIO base pointer (PTA, PTB, PTC, etc.)
in	*pin*	Pin number to be written
in	*value*	Pin value to be written
		• 0: corresponding pin is set to LOW
		• 1: corresponding pin is set to HIGH

Definition at line 465 of file pins_driver.c.

16.69.5.8   void PINS_DRV_WritePins (GPIO_Type ∗const *base,* pins_channel_type_t *pins*)

Write all pins of a port.

This function writes all pins configured as output with the values given in the parameter pins. '0' represents LOW, '1' represents HIGH.

Parameters

in	*base*	GPIO base pointer (PTA, PTB, PTC, etc.)
in	*pins*	Pin mask to be written
		• 0: corresponding pin is set to LOW
		• 1: corresponding pin is set to HIGH

Definition at line 480 of file pins_driver.c.

16.70 Peripheral access layer for S32K116

This module covers all memory mapped register available on SoC.

16.71   Pins Driver (PINS)

16.71.1   Detailed Description

The S32 SDK provides Peripheral Drivers for the PINS module of S32K1xx, S32MTV, S32V234 and MPC574xx devices.

The module provides dedicated pad control to general-purpose pads that can be configured as either inputs or outputs. The PINS module provides registers that enable user software to read values from GPIO pads configured as inputs, and write values to GPIO pads configured as outputs:

- When configured as output, you can write to an internal register to control the state driven on the associated output pad.

- When configured as input, you can detect the state of the associated pad by reading the value from an internal register.

- When configured as input and output, the pad value can be read back, which can be used as a method of checking if the written value appeared on the pad.

The PINS supports these following features: For S32K1xx and S32MTV devices: Pins driver is based on PORT (Port Control and Interrupt) and GPIO (General-Purpose Input/Output) modules Pin interrupt

- Interrupt flag and enable registers for each pin

- Support for edge sensitive (rising, falling, both) or level sensitive (low, high) configured per pin

- Support for interrupt or DMA request configured per pin

- Asynchronous wake-up in low-power modes

- Pin interrupt is functional in all digital pin muxing modes

- Peripheral trigger output (active high, low) configured per pin Digital input filter

- Digital input filter for each pin, usable by any digital peripheral muxed onto the pin

- Individual enable or bypass control field per pin

- Selectable clock source for digital input filter with a five bit resolution on filter size

- Functional in all digital pin multiplexing modes Port control

- Individual pull control fields with pullup, pulldown, and pull-disable support

- Individual drive strength field supporting high and low drive strength

- Individual slew rate field supporting fast and slow slew rates

- Individual input passive filter field supporting enable and disable of the individual input passive filter

- Individual open drain field supporting enable and disable of the individual open drain output

- Individual over-current detect enable with over-current detect flag and associated interrupt

- Individual mux control field supporting analog or pin disabled, GPIO, and up to 6 chip-specific digital functions

- Pad configuration fields are functional in all digital pin muxing modes For MPC574xx and S32V234 devices↩
  : Pins driver is based on SIUL2 (System Integration Unit Lite2) module The System Integration Unit Lite2
  supports these distinctive features:

- 1 to 32 GPIO ports with data control

- Drive data to as many as 16 independent I/O channels

- Sample data from as many as 16 independent I/O channels Two 16-bit registers can be read/written with one access for a 32-bit port, if needed. External interrupt/DMA request support with:

- 1 to 4 system interrupt vectors for 1 to 4 interrupt sources with independent interrupt masks. For 32 external interrupt sources (REQ pins), four groups have eight interrupt sources each, and each of the four groups is assigned one system interrupt vector.

- 1 to 32 programmable digital glitch filters, one for each REQ pin

- 1 to 4 system DMA request channels for 1 to 4 REQ pins

- Edge detection Additionally the SIUL2 contains the Multiplexed Signal Configuration Registers (MSCR) that configure the electrical parameters and settings for as many as 512 functional pads. The number of these registers that is actually implemented varies by device. These registers configure the following pad features:

- Drive strength

- Output impedance control

- Open drain/source output enable

- Slew rate control

- Hysteresis control

- Internal pull control and pull selection

- Pin function assignment

- Control of analog path switches

- Safe mode behavior configuration

Modules

- PINS Driver

16.72 Power Manager

16.72.1 Detailed Description

The S32 SDK Power Manager provides a set of API/services that enables applications to configure and select among various operational and low power modes.

Driver consideration

The Power Manager driver is developed on top of an appropriate hardware access layer (SMC, MC_ME etc). The Power Manager provides API to handle the device power modes. It also supports run-time switching between multiple power modes. Each power mode is described by configuration structures with multiple power-related options. The Power Manager provides a notification mechanism for registered callbacks and API for static and dynamic callback registration.

The Driver uses structures for configuration. The actual format of the structure is defined by the underlying device specific header file. There is a power mode and a callback configuration structure. These structures may be generated using Processor Expert. The user application can use the default for most settings, changing only what is necessary.

This driver provides functions for initializing power manager and changing the power mode.

All methods that access the hardware layer will return an error code to signal if the operation succeeded or failed. The values are defined by the status_t enumeration, and the possible values include: success, switch error, callback notification errors, wrong clock setup error.

Modules

- Power Manager Driver

    This module covers the device-specific clock_manager functionality implemented for S32K1xx and S32MTV SOC.
- Power_s32k1xx

Data Structures

- struct power_manager_notify_struct_t

    Power mode user configuration structure. More...
- struct power_manager_callback_user_config_t

    callback configuration structure More...
- struct power_manager_state_t

    Power manager internal state structure. More...

Typedefs

- typedef void power_manager_callback_data_t

    Callback-specific data.
- typedef status_t(∗ power_manager_callback_t) (power_manager_notify_struct_t ∗notify, power_manager_↩
callback_data_t ∗dataPtr)

    Callback prototype.

Enumerations

- enum power_manager_policy_t { POWER_MANAGER_POLICY_AGREEMENT, POWER_MANAGER_P↩
OLICY_FORCIBLE }

    Power manager policies.
- enum power_manager_notify_t { POWER_MANAGER_NOTIFY_RECOVER = 0x00U, POWER_MANAGE↩
R_NOTIFY_BEFORE = 0x01U, POWER_MANAGER_NOTIFY_AFTER = 0x02U }

The PM notification type. Used to notify registered callbacks. Callback notifications can be invoked in following situations:

- enum power_manager_callback_type_t { POWER_MANAGER_CALLBACK_BEFORE = 0x01U, POWER↩ _MANAGER_CALLBACK_AFTER = 0x02U, POWER_MANAGER_CALLBACK_BEFORE_AFTER = 0x03U }

  The callback type indicates when a callback will be invoked.

Functions

- status_t POWER_SYS_Init (power_manager_user_config_t ∗(∗powerConfigsPtr)[ ], uint8_t configsNumber, power_manager_callback_user_config_t ∗(∗callbacksPtr)[ ], uint8_t callbacksNumber)

  Power manager initialization for operation.
- status_t POWER_SYS_Deinit (void)

  This function deinitializes the Power manager.
- status_t POWER_SYS_SetMode (uint8_t powerModeIndex, power_manager_policy_t policy)

  This function configures the power mode.
- status_t POWER_SYS_GetLastMode (uint8_t ∗powerModeIndexPtr)

  This function returns the last successfully set power mode.
- status_t POWER_SYS_GetLastModeConfig (power_manager_user_config_t ∗∗powerModePtr)

  This function returns the user configuration structure of the last successfully set power mode.
- power_manager_modes_t POWER_SYS_GetCurrentMode (void)

  This function returns currently running power mode.
- uint8_t POWER_SYS_GetErrorCallbackIndex (void)

  This function returns the last failed notification callback.
- power_manager_callback_user_config_t ∗ POWER_SYS_GetErrorCallback (void)

  This function returns the callback configuration structure for the last failed notification.

16.72.2 Data Structure Documentation

16.72.2.1 struct power_manager_notify_struct_t

Power mode user configuration structure.

This structure defines power mode with additional power options. This structure is implementation-defend. Please refer to actual definition based on the underlying HAL (SMC, MC_ME etc). Applications may define multiple power modes and switch between them. A list of all defined power modes is passed to the Power manager during initialization as an array of references to structures of this type (see POWER_SYS_Init()). Power modes can be switched by calling POWER_SYS_SetMode(), which takes as argument the index of the reqested power mode in the list passed during manager initialization. The power mode currently in use can be retrieved by calling POWER_SYS_GetLast↩ Mode(), which provides the index of the current power mode, or by calling POWER_SYS_GetLastModeConfig(), which provides a pointer to the configuration structure of the current power mode. The members of the power mode configuration structure depend on power options available for a specific chip, and includes at least the power mode. The available power modes are chip-specific. See power_manager_modes_t defined in the underlying HAL for a list of all supported modes.

Power notification structure passed to registered callback function

Implements power_manager_notify_struct_t_Class

Definition at line 143 of file power_manager.h.

Data Fields

- power_manager_user_config_t ∗ targetPowerConfigPtr
- uint8_t targetPowerConfigIndex
- power_manager_policy_t policy
- power_manager_notify_t notifyType

Field Documentation

16.72.2.1.1   power_manager_notify_t notifyType

Power mode notification type.

Definition at line 148 of file power_manager.h.

16.72.2.1.2   power_manager_policy_t policy

Power mode transition policy.

Definition at line 147 of file power_manager.h.

16.72.2.1.3   uint8_t targetPowerConfigIndex

Target power configuration index.

Definition at line 146 of file power_manager.h.

16.72.2.1.4   power_manager_user_config_t∗ targetPowerConfigPtr

Pointer to target power configuration

Definition at line 145 of file power_manager.h.

16.72.2.2   struct power_manager_callback_user_config_t

callback configuration structure

This structure holds configuration of callbacks passed to the Power manager during its initialization. Structures of this type are expected to be statically allocated. This structure contains following application-defined data: callback - pointer to the callback function callbackType - specifies when the callback is called callbackData - pointer to the data passed to the callback Implements power_manager_callback_user_config_t_Class

Definition at line 188 of file power_manager.h.

Data Fields

- power_manager_callback_t callbackFunction
- power_manager_callback_type_t callbackType
- power_manager_callback_data_t ∗ callbackData

Field Documentation

16.72.2.2.1   power_manager_callback_data_t∗ callbackData

Definition at line 192 of file power_manager.h.

16.72.2.2.2   power_manager_callback_t callbackFunction

Definition at line 190 of file power_manager.h.

16.72.2.2.3   power_manager_callback_type_t callbackType

Definition at line 191 of file power_manager.h.

16.72.2.3   struct power_manager_state_t

Power manager internal state structure.

Power manager internal structure. Contains data necessary for Power manager proper functionality. Stores references to registered power mode configurations, callbacks, and other internal data. This structure is statically allocated and initialized by POWER_SYS_Init(). Implements power_manager_state_t_Class

Definition at line 204 of file power_manager.h.

Data Fields

- power_manager_user_config_t ∗(∗ configs)[ ]
- uint8_t configsNumber
- power_manager_callback_user_config_t ∗(∗ staticCallbacks)[ ]
- uint8_t staticCallbacksNumber
- uint8_t errorCallbackIndex
- uint8_t currentConfig

Field Documentation

16.72.2.3.1   power_manager_user_config_t∗(∗ **configs)[ ]**

Pointer to power configure table.

Definition at line 206 of file power_manager.h.

16.72.2.3.2   uint8_t configsNumber

Number of power configurations

Definition at line 207 of file power_manager.h.

16.72.2.3.3   uint8_t currentConfig

Index of current configuration.

Definition at line 211 of file power_manager.h.

16.72.2.3.4   uint8_t errorCallbackIndex

Index of callback returns error.

Definition at line 210 of file power_manager.h.

16.72.2.3.5   power_manager_callback_user_config_t∗(∗ **staticCallbacks)[ ]**

Pointer to callback table.

Definition at line 208 of file power_manager.h.

16.72.2.3.6   uint8_t staticCallbacksNumber

Max. number of callback configurations

Definition at line 209 of file power_manager.h.

16.72.3   Typedef Documentation

16.72.3.1   typedef void power_manager_callback_data_t

Callback-specific data.

Pointer to data of this type is passed during callback registration. The pointer is part of the power_manager_←
callback_user_config_t structure and is passed to the callback during power mode change notifications. Implements
power_manager_callback_data_t_Class

Definition at line 118 of file power_manager.h.

16.72.3.2 typedef status_t(∗ power_manager_callback_t) (power_manager_notify_struct_t ∗notify, power_manager_callback_data_t ∗dataPtr)

Callback prototype.

Declaration of callback. It is common for all registered callbacks. Function pointer of this type is part of power↩ _manager_callback_user_config_t callback configuration structure. Depending on the callback type, the callback function is invoked during power mode change (see POWER_SYS_SetMode()) before the mode change, after it, or in both cases to notify about the change progress (see power_manager_callback_type_t). When called, the type of the notification is passed as parameter along with a pointer to power mode configuration structure (see power↩ _manager_notify_struct_t) and any data passed during the callback registration (see power_manager_callback_↩ data_t). When notified before a mode change, depending on the power mode change policy (see power_manager↩ _policy_t) the callback may deny the mode change by returning any error code other than STATUS_SUCCESS (see POWER_SYS_SetMode()).

Parameters

notify	Notification structure.
dataPtr	Callback data. Pointer to the data passed during callback registration. Intended to pass any driver or application data such as internal state information.

Returns

> An error code or STATUS_SUCCESS. Implements power_manager_callback_t_Class

Definition at line 172 of file power_manager.h.

16.72.4 Enumeration Type Documentation

16.72.4.1 enum power_manager_callback_type_t

The callback type indicates when a callback will be invoked.

Used in the callback configuration structures (power_manager_callback_user_config_t) to specify when the registered callback will be called during power mode change initiated by POWER_SYS_SetMode().

Implements power_manager_callback_type_t_Class

Enumerator

> **POWER_MANAGER_CALLBACK_BEFORE** Before callback.
> **POWER_MANAGER_CALLBACK_AFTER** After callback.
> **POWER_MANAGER_CALLBACK_BEFORE_AFTER** Before-After callback.

Definition at line 103 of file power_manager.h.

16.72.4.2 enum power_manager_notify_t

The PM notification type. Used to notify registered callbacks. Callback notifications can be invoked in following situations:

- before a power mode change (Callback return value can affect POWER_SYS_SetMode() execution. Refer to the POWER_SYS_SetMode() and power_manager_policy_t documentation).

- after a successful change of the power mode.

- after an unsuccessful attempt to switch power mode, in order to recover to a working state. Implements power_manager_notify_t_Class

Enumerator

> **POWER_MANAGER_NOTIFY_RECOVER** Notify IP to recover to previous work state.

POWER_MANAGER_NOTIFY_BEFORE Notify IP that the system will change the power setting.

POWER_MANAGER_NOTIFY_AFTER Notify IP that the system has changed to a new power setting.

Definition at line 87 of file power_manager.h.

16.72.4.3   enum power_manager_policy_t

Power manager policies.

Defines whether the mode switch initiated by the POWER_SYS_SetMode() is agreed upon (depending on the result of notification callbacks), or forced. For POWER_MANAGER_POLICY_FORCIBLE the power mode is changed regardless of the callback results, while for POWER_MANAGER_POLICY_AGREEMENT policy any error code returned by one of the callbacks aborts the mode change. See also POWER_SYS_SetMode() description. Implements power_manager_policy_t_Class

Enumerator

POWER_MANAGER_POLICY_AGREEMENT Power mode is changed if all of the callbacks return success.

POWER_MANAGER_POLICY_FORCIBLE Power mode is changed regardless of the result of callbacks.

Definition at line 72 of file power_manager.h.

16.72.5   Function Documentation

16.72.5.1   status_t POWER_SYS_Deinit (void)

This function deinitializes the Power manager.

Returns

An error code or STATUS_SUCCESS.

Definition at line 120 of file power_manager.c.

16.72.5.2   power_manager_modes_t POWER_SYS_GetCurrentMode (void)

This function returns currently running power mode.

This function reads hardware settings and returns currently running power mode.

Returns

Currently used run power mode.

Definition at line 218 of file power_manager_S32K1xx.c.

16.72.5.3   power_manager_callback_user_config_t∗ POWER_SYS_GetErrorCallback (void)

This function returns the callback configuration structure for the last failed notification.

This function returns a pointer to configuration structure of the last callback that failed during the power mode switch when POWER_SYS_SetMode() was called. If the last POWER_SYS_SetMode() call ended successfully, a NULL value is returned.

Returns

Pointer to the callback configuration which returns error.

Definition at line 218 of file power_manager.c.

16.72.5.4   uint8_t POWER_SYS_GetErrorCallbackIndex (void)

This function returns the last failed notification callback.

This function returns the index of the last callback that failed during the power mode switch when POWER_SY↩
S_SetMode() was called. The returned value represents the index in the array of registered callbacks. If the last
POWER_SYS_SetMode() call ended successfully, a value equal to the number of registered callbacks is returned.

Returns

> Callback index of last failed callback or value equal to callbacks count.

Definition at line 206 of file power_manager.c.

16.72.5.5   status_t POWER_SYS_GetLastMode (uint8_t ∗ powerModeIndexPtr)

This function returns the last successfully set power mode.

This function returns index of power mode which was last set using POWER_SYS_SetMode(). If the power mode
was entered even though some of the registered callbacks denied the mode change, or if any of the callbacks
invoked after the entering/restoring run mode failed, then the return code of this function has STATUS_ERROR
value.

Parameters

out	powerMode↩ IndexPtr	Power mode which has been set represented as an index into array of power mode configurations passed to the POWER_SYS_Init().

Returns

> An error code or STATUS_SUCCESS.

Definition at line 143 of file power_manager.c.

16.72.5.6   status_t POWER_SYS_GetLastModeConfig (power_manager_user_config_t ∗∗ powerModePtr)

This function returns the user configuration structure of the last successfully set power mode.

This function returns a pointer to configuration structure which was last set using POWER_SYS_SetMode(). If the
current power mode was entered even though some of the registered callbacks denied the mode change, or if
any of the callbacks invoked after the entering/restoring run mode failed, then the return code of this function has
STATUS_ERROR value.

Parameters

out	powerModePtr	Pointer to power mode configuration structure of the last set power mode.

Returns

> An error code or STATUS_SUCCESS.

Definition at line 175 of file power_manager.c.

16.72.5.7   status_t POWER_SYS_Init (power_manager_user_config_t ∗(∗) powerConfigsPtr[ ], uint8_t configsNumber,
power_manager_callback_user_config_t ∗(∗) callbacksPtr[ ], uint8_t callbacksNumber)

Power manager initialization for operation.

This function initializes the Power manager and its run-time state structure. Pointer to an array of Power mode
configuration structures needs to be passed as a parameter along with a parameter specifying its size. At least one
power mode configuration is required. Optionally, pointer to the array of predefined callbacks can be passed with
its corresponding size parameter. For details about callbacks, refer to the power_manager_callback_user_config_t.
As Power manager stores only pointers to arrays of these structures, they need to exist and be valid for the entire
life cycle of Power manager.

Parameters

in	*powerConfigsPtr*	A pointer to an array of pointers to all power configurations which will be handled by Power manager.
in	*configsNumber*	Number of power configurations. Size of powerConfigsPtr array.
in	*callbacksPtr*	A pointer to an array of pointers to callback configurations. If there are no callbacks to register during Power manager initialization, use NULL value.
in	*callbacks↩ Number*	Number of registered callbacks. Size of callbacksPtr array.

Returns

> An error code or STATUS_SUCCESS.

Definition at line 80 of file power_manager.c.

16.72.5.8   status_t POWER_SYS_SetMode (uint8_t *powerModeIndex,* power_manager_policy_t *policy*)

This function configures the power mode.

This function switches to one of the defined power modes. Requested mode number is passed as an input parameter. This function notifies all registered callback functions before the mode change (using POWER_MANAGE↩ R_CALLBACK_BEFORE set as callback type parameter), sets specific power options defined in the power mode configuration and enters the specified mode. In case of run modes (for example, Run, Very low power run, or High speed run), this function also invokes all registered callbacks after the mode change (using POWER_MANAGER↩ _CALLBACK_AFTER). In case of sleep or deep sleep modes, if the requested mode is not exited through a reset, these notifications are sent after the core wakes up. Callbacks are invoked in the following order: All registered callbacks are notified ordered by index in the callbacks array (see callbacksPtr parameter of POWER_SYS_Init()). The same order is used for before and after switch notifications. The notifications before the power mode switch can be used to obtain confirmation about the change from registered callbacks. If any registered callback denies the power mode change, further execution of this function depends on mode change policy: the mode change is either forced(POWER_MANAGER_POLICY_FORCIBLE) or aborted(POWER_MANAGER_POLICY_AGREE↩ MENT). When mode change is forced, the results of the before switch notifications are ignored. If agreement is requested, in case any callback returns an error code then further before switch notifications are cancelled and all already notified callbacks are re-invoked with POWER_MANAGER_CALLBACK_AFTER set as callback type parameter. The index of the callback which returned error code during pre-switch notifications is stored and can be obtained by using POWER_SYS_GetErrorCallback(). Any error codes during callbacks re-invocation (recover phase) are ignored. POWER_SYS_SetMode() returns an error code denoting the phase in which a callback failed. It is possible to enter any mode supported by the processor. Refer to the chip reference manual for the list of available power modes. If it is necessary to switch into an intermediate power mode prior to entering the requested mode (for example, when switching from Run into Very low power wait through Very low power run mode), then the intermediate mode is entered without invoking the callback mechanism.

Parameters

in	*powerMode↩ Index*	Requested power mode represented as an index into array of user-defined power mode configurations passed to the POWER_SYS_Init().
in	*policy*	Transaction policy

Returns

> An error code or STATUS_SUCCESS.

Definition at line 338 of file power_manager.c.

16.73   Power Manager Driver

This module covers the device-specific clock_manager functionality implemented for S32K1xx and S32MTV SOC.

Hardware background

System mode controller (SMC) is passing the system into and out of all low-power Stop and Run modes. Controls the power, clocks and memories of the system to achieve the power consumption and functionality of that mode.

Driver consideration

Power mode entry and sleep-on-exit-value are provided at initialization time through the power manager user configuration structure. With platform is S32K14x, the available power mode entries are the following ones: HSRUN, RUN, VLPR, STOP1, STOP2 and VLPS. With platform is S32MTV,S32K11x, the available power mode entries are the following ones: RUN, VLPR, STOP1, STOP2 and VLPS.

This is an example of configuration:

```
power_manager_user_config_t pwrMan1_InitConfig0 = {
    .powerMode = POWER_MANAGER_RUN,
    .sleepOnExitValue = false,
};

power_manager_user_config_t *powerConfigsArr[] = {
    &pwrMan1_InitConfig0
};

power_manager_callback_user_config_t * powerCallbacksConfigsArr[] = {(
    void *)0};

if (STATUS_SUCCESS != POWER_SYS_Init(&powerConfigsArr,1,&powerCallbacksConfigsArr,0)) {
    ...
}
else {
    ...
}

if (STATUS_SUCCESS != POWER_SYS_SetMode(0,
    POWER_MANAGER_POLICY_AGREEMENT)) {
    ...
}
else {
    ...
}
```

16.74 Power_s32k1xx

16.74.1 Detailed Description

Data Structures

- struct power_manager_user_config_t

    Power mode user configuration structure. More...

- struct smc_power_mode_protection_config_t

    Power mode protection configuration. More...

- struct smc_power_mode_config_t

    Power mode control configuration used for calling the SMC_SYS_SetPowerMode API. More...

- struct smc_version_info_t

    SMC module version number. More...

- struct rcm_version_info_t

    RCM module version number. More...

Enumerations

- enum power_manager_modes_t { POWER_MANAGER_RUN, POWER_MANAGER_VLPR, POWER_MA←
  NAGER_VLPS, POWER_MANAGER_MAX }

    Power modes enumeration.

- enum power_mode_stat_t {
  STAT_RUN = 0x01, STAT_STOP = 0x02, STAT_VLPR = 0x04, STAT_VLPW = 0x08,
  STAT_VLPS = 0x10, STAT_HSRUN = 0x80, STAT_INVALID = 0xFF }

    Power Modes in PMSTAT.

- enum power_modes_protect_t { ALLOW_HSRUN, ALLOW_VLP, ALLOW_MAX }

    Power Modes Protection.

- enum smc_run_mode_t { SMC_RUN, SMC_RESERVED_RUN, SMC_VLPR, SMC_HSRUN }

    Run mode definition.

- enum smc_stop_mode_t { SMC_STOP = 0U, SMC_RESERVED_STOP1 = 1U, SMC_VLPS = 2U }

    Stop mode definition.

- enum smc_stop_option_t { SMC_STOP_RESERVED = 0x00, SMC_STOP1 = 0x01, SMC_STOP2 = 0x02 }

    STOP option.

- enum pmc_int_select_t { PMC_INT_LOW_VOLT_DETECT, PMC_INT_LOW_VOLT_WARN }

    Power management control interrupts.

- enum rcm_source_names_t {
  RCM_LOW_VOLT_DETECT = 1U, RCM_LOSS_OF_CLK = 2U, RCM_LOSS_OF_LOCK = 3U, RCM_WA←
  TCH_DOG = 5U,
  RCM_EXTERNAL_PIN = 6U, RCM_POWER_ON = 7U, RCM_SJTAG = 8U, RCM_CORE_LOCKUP = 9U,
  RCM_SOFTWARE = 10U, RCM_SMDM_AP = 11U, RCM_STOP_MODE_ACK_ERR = 13U, RCM_SRC_←
  NAME_MAX }

    System Reset Source Name definitions Implements rcm_source_names_t_Class.

- enum rcm_filter_run_wait_modes_t { RCM_FILTER_DISABLED, RCM_FILTER_BUS_CLK, RCM_FILTE←
  R_LPO_CLK, RCM_FILTER_RESERVED }

    Reset pin filter select in Run and Wait modes.

- enum rcm_reset_delay_time_t { RCM_10LPO_CYCLES_DELAY, RCM_34LPO_CYCLES_DELAY, RCM_←
  130LPO_CYCLES_DELAY, RCM_514LPO_CYCLES_DELAY }

    Reset delay time.

Functions

- status_t [POWER_SYS_DoInit](void)

    This function implementation-specific configuration of power modes.

- status_t [POWER_SYS_DoDeinit](void)

    This function implementation-specific de-initialization of power manager.

- status_t [POWER_SYS_DoSetMode](const [power_manager_user_config_t](#) ∗const configPtr)

    This function configures the power mode.

- bool [POWER_SYS_GetResetSrcStatusCmd](const RCM_Type ∗const baseAddr, const [rcm_source_↩ names_t](#) srcName)

    Gets the reset source status.

16.74.2 Data Structure Documentation

16.74.2.1 struct power_manager_user_config_t

Power mode user configuration structure.

List of power mode configuration structure members depends on power options available for the specific chip. Complete list contains: mode - S32K power mode. List of available modes is chip-specific. See power_manager↩ _modes_t list of modes. sleepOnExitOption - Controls whether the sleep-on-exit option value is used(when set to true) or ignored(when set to false). See sleepOnExitValue. sleepOnExitValue - When set to true, ARM core returns to sleep (S32K wait modes) or deep sleep state (S32K stop modes) after interrupt service finishes. When set to false, core stays woken-up. Implements power_manager_user_config_t_Class

Definition at line 98 of file power_manager_S32K1xx.h.

Data Fields

- [power_manager_modes_t powerMode](#)
- bool [sleepOnExitValue](#)

Field Documentation

16.74.2.1.1 power_manager_modes_t powerMode

Definition at line 100 of file power_manager_S32K1xx.h.

16.74.2.1.2 bool sleepOnExitValue

Definition at line 101 of file power_manager_S32K1xx.h.

16.74.2.2 struct smc_power_mode_protection_config_t

Power mode protection configuration.

Definition at line 167 of file power_manager_S32K1xx.h.

Data Fields

- bool [vlpProt](#)

Field Documentation

16.74.2.2.1 bool vlpProt

VLP protect

Definition at line 169 of file power_manager_S32K1xx.h.

16.74.2.3   struct smc_power_mode_config_t

Power mode control configuration used for calling the SMC_SYS_SetPowerMode API.

Definition at line 179 of file power_manager_S32K1xx.h.

Data Fields

- power_manager_modes_t powerModeName

Field Documentation

16.74.2.3.1   power_manager_modes_t powerModeName

Power mode(enum), see power_manager_modes_t

Definition at line 181 of file power_manager_S32K1xx.h.

16.74.2.4   struct smc_version_info_t

SMC module version number.

Definition at line 195 of file power_manager_S32K1xx.h.

Data Fields

- uint32_t majorNumber
- uint32_t minorNumber
- uint32_t featureNumber

Field Documentation

16.74.2.4.1   uint32_t featureNumber

Feature Specification Number

Definition at line 199 of file power_manager_S32K1xx.h.

16.74.2.4.2   uint32_t majorNumber

Major Version Number

Definition at line 197 of file power_manager_S32K1xx.h.

16.74.2.4.3   uint32_t minorNumber

Minor Version Number

Definition at line 198 of file power_manager_S32K1xx.h.

16.74.2.5   struct rcm_version_info_t

RCM module version number.

Definition at line 264 of file power_manager_S32K1xx.h.

Data Fields

- uint32_t majorNumber
- uint32_t minorNumber
- uint32_t featureNumber

Field Documentation

16.74.2.5.1 uint32_t featureNumber

Feature Specification Number

Definition at line 268 of file power_manager_S32K1xx.h.

16.74.2.5.2 uint32_t majorNumber

Major Version Number

Definition at line 266 of file power_manager_S32K1xx.h.

16.74.2.5.3 uint32_t minorNumber

Minor Version Number

Definition at line 267 of file power_manager_S32K1xx.h.

16.74.3 Enumeration Type Documentation

16.74.3.1 enum pmc_int_select_t

Power management control interrupts.

Enumerator

> ***PMC_INT_LOW_VOLT_DETECT*** Low Voltage Detect Interrupt
> ***PMC_INT_LOW_VOLT_WARN*** Low Voltage Warning Interrupt

Definition at line 206 of file power_manager_S32K1xx.h.

16.74.3.2 enum power_manager_modes_t

Power modes enumeration.

Defines power modes. Used in the power mode configuration structure (power_manager_user_config_t). From ARM core perspective, Power modes can be generally divided into run modes (High speed run, Run and Very low power run), sleep (Wait and Very low power wait) and deep sleep modes (all Stop modes). List of power modes supported by specific chip along with requirements for entering and exiting of these modes can be found in chip documentation. List of all supported power modes:

- POWER_MANAGER_HSRUN - High speed run mode.

- POWER_MANAGER_RUN - Run mode.

- POWER_MANAGER_VLPR - Very low power run mode.

- POWER_MANAGER_WAIT - Wait mode.

- POWER_MANAGER_VLPW - Very low power wait mode.

- POWER_MANAGER_PSTOP1 - Partial stop 1 mode.

- POWER_MANAGER_PSTOP2 - Partial stop 2 mode.

- POWER_MANAGER_PSTOP1 - Stop 1 mode.

- POWER_MANAGER_PSTOP2 - Stop 2 mode.

- POWER_MANAGER_VLPS - Very low power stop mode. Implements power_manager_modes_t_Class

Enumerator

> ***POWER_MANAGER_RUN*** Run mode.

POWER_MANAGER_VLPR Very low power run mode.

POWER_MANAGER_VLPS Very low power stop mode.

POWER_MANAGER_MAX

Definition at line 61 of file power_manager_S32K1xx.h.

16.74.3.3 enum power_mode_stat_t

Power Modes in PMSTAT.

Enumerator

STAT_RUN 0000_0001 - Current power mode is RUN

STAT_STOP 0000_0010 - Current power mode is STOP

STAT_VLPR 0000_0100 - Current power mode is VLPR

STAT_VLPW 0000_1000 - Current power mode is VLPW

STAT_VLPS 0001_0000 - Current power mode is VLPS

STAT_HSRUN 1000_0000 - Current power mode is HSRUN

STAT_INVALID 1111_1111 - Non-existing power mode

Definition at line 108 of file power_manager_S32K1xx.h.

16.74.3.4 enum power_modes_protect_t

Power Modes Protection.

Enumerator

ALLOW_HSRUN Allow High Speed Run mode

ALLOW_VLP Allow Very-Low-Power Modes

ALLOW_MAX

Definition at line 123 of file power_manager_S32K1xx.h.

16.74.3.5 enum rcm_filter_run_wait_modes_t

Reset pin filter select in Run and Wait modes.

Enumerator

RCM_FILTER_DISABLED All filtering disabled

RCM_FILTER_BUS_CLK Bus clock filter enabled

RCM_FILTER_LPO_CLK LPO clock filter enabled

RCM_FILTER_RESERVED Reserved setting

Definition at line 239 of file power_manager_S32K1xx.h.

16.74.3.6 enum rcm_reset_delay_time_t

Reset delay time.

Enumerator

RCM_10LPO_CYCLES_DELAY reset delay time 10 LPO cycles

RCM_34LPO_CYCLES_DELAY reset delay time 34 LPO cycles

RCM_130LPO_CYCLES_DELAY reset delay time 130 LPO cycles

RCM_514LPO_CYCLES_DELAY reset delay time 514 LPO cycles

Definition at line 252 of file power_manager_S32K1xx.h.

16.74.3.7 enum rcm_source_names_t

System Reset Source Name definitions Implements rcm_source_names_t_Class.

Enumerator

> ***RCM_LOW_VOLT_DETECT*** Low voltage detect reset
>
> ***RCM_LOSS_OF_CLK*** Loss of clock reset
>
> ***RCM_LOSS_OF_LOCK*** Loss of lock reset
>
> ***RCM_WATCH_DOG*** Watch dog reset
>
> ***RCM_EXTERNAL_PIN*** External pin reset
>
> ***RCM_POWER_ON*** Power on reset
>
> ***RCM_SJTAG*** JTAG generated reset
>
> ***RCM_CORE_LOCKUP*** core lockup reset
>
> ***RCM_SOFTWARE*** Software reset
>
> ***RCM_SMDM_AP*** MDM-AP system reset
>
> ***RCM_STOP_MODE_ACK_ERR*** Stop mode ack error reset
>
> ***RCM_SRC_NAME_MAX***

Definition at line 216 of file power_manager_S32K1xx.h.

16.74.3.8 enum smc_run_mode_t

Run mode definition.

Enumerator

> ***SMC_RUN*** normal RUN mode
>
> ***SMC_RESERVED_RUN***
>
> ***SMC_VLPR*** Very-Low-Power RUN mode
>
> ***SMC_HSRUN*** High Speed Run mode (HSRUN)

Definition at line 134 of file power_manager_S32K1xx.h.

16.74.3.9 enum smc_stop_mode_t

Stop mode definition.

Enumerator

> ***SMC_STOP*** Normal STOP mode
>
> ***SMC_RESERVED_STOP1*** Reserved
>
> ***SMC_VLPS*** Very-Low-Power STOP mode

Definition at line 145 of file power_manager_S32K1xx.h.

16.74.3.10 enum smc_stop_option_t

STOP option.

Enumerator

> ***SMC_STOP_RESERVED*** Reserved stop mode
>
> ***SMC_STOP1*** Stop with both system and bus clocks disabled
>
> ***SMC_STOP2*** Stop with system clock disabled and bus clock enabled

Definition at line 156 of file power_manager_S32K1xx.h.

16.74.4 Function Documentation

16.74.4.1 status_t POWER_SYS_DoDeinit (void)

This function implementation-specific de-initialization of power manager.

This function performs the actual implementation-specific de-initialization.

Returns

Operation status

- STATUS_SUCCESS: Operation was successful.
- STATUS_ERROR: Operation failed.

Definition at line 178 of file power_manager_S32K1xx.c.

16.74.4.2 status_t POWER_SYS_DoInit (void)

This function implementation-specific configuration of power modes.

This function performs the actual implementation-specific initialization based on the provided power mode configurations.

Returns

Operation status

- STATUS_SUCCESS: Operation was successful.
- STATUS_ERROR: Operation failed.

Definition at line 141 of file power_manager_S32K1xx.c.

16.74.4.3 status_t POWER_SYS_DoSetMode (const power_manager_user_config_t ∗const *configPtr*)

This function configures the power mode.

This function performs the actual implementation-specific logic to switch to one of the defined power modes.

Parameters

configPtr	Pointer to user configuration structure

Returns

Operation status

- STATUS_SUCCESS: Operation was successful.
- STATUS_ERROR: Operation failed.

Definition at line 191 of file power_manager_S32K1xx.c.

16.74.4.4 bool POWER_SYS_GetResetSrcStatusCmd (const RCM_Type ∗const *baseAddr,* const rcm_source_names_t *srcName*)

Gets the reset source status.

This function gets the current reset source status for a specified source.

Parameters

in	*baseAddr*	Register base address of RCM
in	*srcName*	reset source name

Returns

      status True or false for specified reset source

Definition at line 630 of file power_manager_S32K1xx.c.

16.75   Programmable Delay Block (PDB)

16.75.1   Detailed Description

The S32 SDK provides Peripheral Drivers for the Programmable Delay Block (PDB) module of S32 SDK devices.

The PDB is a configurable counter that can generate events (triggers) that can be used by the ADC to start conversions or routed through TRGMUX to other modules in the S32K144.

Modules

- PDB Driver

     Programmable Delay Block Peripheral Driver.

16.76 Raw API

16.76.1 Detailed Description

The raw API is operating on PDU level and it is typically used to gateway PDUs between CAN and LIN.

Usually, a FIFO is used to buffer PDUs in order to handle the different bus speeds.

Functions

- void [ld_put_raw](l_ifc_handle iii, const l_u8 ∗const data)

  Queue the transmission of 8 bytes of data in one frame.

- void [ld_get_raw](l_ifc_handle iii, l_u8 ∗const data)

  Copy the oldest received diagnostic frame data to the memory specified by data.

- l_u8 [ld_raw_tx_status](l_ifc_handle iii)

  Get the status of the raw frame transmission function.

- l_u8 [ld_raw_rx_status](l_ifc_handle iii)

  Get the status of the raw frame receive function.

16.76.2 Function Documentation

16.76.2.1 void ld_get_raw (l_ifc_handle *iii,* l_u8 ∗const *data*)

Copy the oldest received diagnostic frame data to the memory specified by data.

Parameters

in	*iii*	Interface name
in	*data*	Buffer for the data to be transmitted

Returns

> void

Copy the oldest received diagnostic frame data to the memory specified by data. The data returned is received from master request frame for slave node and the slave response frame for master node.

Definition at line 168 of file lin_commontl_api.c.

16.76.2.2 void ld_put_raw (l_ifc_handle *iii,* const l_u8 ∗const *data*)

Queue the transmission of 8 bytes of data in one frame.

Parameters

in	*iii*	Interface name
in	*data*	Buffer for the data to be transmitted

Returns

> void

Queue the transmission of 8 bytes of data in one frame The data is sent in the next suitable frame.

Definition at line 134 of file lin_commontl_api.c.

16.76.2.3 l_u8 ld_raw_rx_status (l_ifc_handle *iii*)

Get the status of the raw frame receive function.

Parameters

in	*iii*	Interface name

Returns

> l_u8

Get the status of the raw frame receive function: LD_NO_DATA The receive queue is empty.(For LIN2.1 and above only) LD_DATA_AVAILABLE The receive queue contains data that can be read. LD_RECEIVE_ERROR LIN protocol errors occurred during the transfer; initialize and redo the transfer.(For LIN2.1 and above only). LD_TRA←
NSFER_ERROR: (For LIN2.0 and J2602 only) LIN protocol errors occurred during the transfer; initialize and redo the transfer.

Definition at line 200 of file lin_commontl_api.c.

16.76.2.4 l_u8 ld_raw_tx_status (l_ifc_handle *iii*)

Get the status of the raw frame transmission function.

Parameters

in	*iii*	Interface name

Returns

> l_u8

Get the status of the raw frame transmission function: This function is available for $<$ br $/ >$ LD_QUEUE_EMPTY : The transmit queue is empty. In case previous calls to $<$ br $/ >$ ld_put_raw, all frames in the queue have been $<$ br $/ >$ transmitted. $<$ br $/ >$ LD_QUEUE_AVAILABLE: The transmit queue contains entries, but is not full. $<$ br $/ >$ (For LIN2.1 and above only). LD_QUEUE_FULL : The transmit queue is full and can not accept further $<$ br $/ >$ frames. $<$ br $/ >$ LD_TRANSMIT_ERROR : (For LIN2.1 and above only) LIN protocol errors occurred during the transfer; initialize and redo the transfer. LD_TRANSFER_ERROR: (For LIN2.0 and J2602 only) LIN protocol errors occurred during the transfer; initialize and redo the transfer.

Definition at line 185 of file lin_commontl_api.c.

16.77   Real Time Clock Driver

16.77.1   Detailed Description

Real Time Clock Driver Peripheral Driver.

Data Structures

- struct rtc_timedate_t

    RTC Time Date structure Implements : rtc_timedate_t_Class. More...

- struct rtc_init_config_t

    RTC Initialization structure Implements : rtc_init_config_t_Class. More...

- struct rtc_alarm_config_t

    RTC alarm configuration Implements : rtc_alarm_config_t_Class. More...

- struct rtc_interrupt_config_t

    RTC interrupt configuration. It is used to configure interrupt other than Time Alarm and Time Seconds interrupt Implements : rtc_interrupt_config_t_Class. More...

- struct rtc_seconds_int_config_t

    RTC Seconds Interrupt Configuration Implements : rtc_seconds_int_config_t_Class. More...

- struct rtc_register_lock_config_t

    RTC Register Lock Configuration Implements : rtc_register_lock_config_t_Class. More...

Macros

- #define SECONDS_IN_A_DAY (86400UL)
- #define SECONDS_IN_A_HOUR (3600U)
- #define SECONDS_IN_A_MIN (60U)
- #define MINS_IN_A_HOUR (60U)
- #define HOURS_IN_A_DAY (24U)
- #define DAYS_IN_A_YEAR (365U)
- #define DAYS_IN_A_LEAP_YEAR (366U)
- #define YEAR_RANGE_START (1970U)
- #define YEAR_RANGE_END (2099U)

Enumerations

- enum rtc_second_int_cfg_t {
  RTC_INT_1HZ = 0x00U, RTC_INT_2HZ = 0x01U, RTC_INT_4HZ = 0x02U, RTC_INT_8HZ = 0x03U,
  RTC_INT_16HZ = 0x04U, RTC_INT_32HZ = 0x05U, RTC_INT_64HZ = 0x06U, RTC_INT_128HZ = 0x07U }

    RTC Seconds interrupt configuration Implements : rtc_second_int_cfg_t_Class.

- enum rtc_clk_out_config_t { RTC_CLKOUT_DISABLED = 0x00U, RTC_CLKOUT_SRC_TSIC = 0x01U, R↩
  TC_CLKOUT_SRC_32KHZ = 0x02U }

    RTC CLKOUT pin configuration Implements : rtc_clk_out_config_t_Class.

- enum rtc_clk_select_t { RTC_CLK_SRC_OSC_32KHZ = 0x00U, RTC_CLK_SRC_LPO_1KHZ = 0x01U }

    RTC clock select Implements : rtc_clk_select_t_Class.

- enum rtc_lock_register_select_t { RTC_LOCK_REG_LOCK = 0x00U, RTC_STATUS_REG_LOCK = 0x01U,
  RTC_CTRL_REG_LOCK = 0x02U, RTC_TCL_REG_LOCK = 0x03U }

    RTC register lock Implements : rtc_lock_register_select_t_Class.

Functions

- status_t RTC_DRV_Init (uint32_t instance, const rtc_init_config_t ∗const rtcUserCfg)

    This function initializes the RTC instance with the settings provided by the user via the rtcUserCfg parameter. The user must ensure that clock is enabled for the RTC instance used. If the Control register is locked then this method returns STATUS_ERROR. In order to clear the CR Lock the user must perform a power-on reset.

- status_t RTC_DRV_Deinit (uint32_t instance)

    This function deinitializes the RTC instance. If the Control register is locked then this method returns STATUS_ER↩ROR.

- void RTC_DRV_GetDefaultConfig (rtc_init_config_t ∗const config)

    This function will set the default configuration values into the structure passed as a parameter.

- status_t RTC_DRV_StartCounter (uint32_t instance)

    Start RTC instance counter. Before calling this function the user should use RTC_DRV_SetTimeDate to configure the start time.

- status_t RTC_DRV_StopCounter (uint32_t instance)

    Disable RTC instance counter.

- status_t RTC_DRV_GetCurrentTimeDate (uint32_t instance, rtc_timedate_t ∗const currentTime)

    Get current time and date from RTC instance.

- status_t RTC_DRV_SetTimeDate (uint32_t instance, const rtc_timedate_t ∗const time)

    Set time and date for RTC instance. The user must stop the counter before using this function. Otherwise it will return an error.

- status_t RTC_DRV_ConfigureRegisterLock (uint32_t instance, const rtc_register_lock_config_t ∗const lockConfig)

    This method configures register lock for the corresponding RTC instance. Remember that all the registers are unlocked only by software reset or power on reset. (Except for CR that is unlocked only by POR).

- void RTC_DRV_GetRegisterLock (uint32_t instance, rtc_register_lock_config_t ∗const lockConfig)

    Get which registers are locked for RTC instance.

- status_t RTC_DRV_ConfigureTimeCompensation (uint32_t instance, uint8_t compInterval, int8_t compensation)

    This method configures time compensation. Data is passed by the compInterval and compensation parameters. For more details regarding coefficient calculation see the Reference Manual.

- void RTC_DRV_GetTimeCompensation (uint32_t instance, uint8_t ∗compInterval, int8_t ∗compensation)

    This retrieves the time compensation coefficients and saves them on the variables referenced by the parameters.

- void RTC_DRV_ConfigureFaultInt (uint32_t instance, rtc_interrupt_config_t ∗const intConfig)

    This method configures fault interrupts such as:

- void RTC_DRV_ConfigureSecondsInt (uint32_t instance, rtc_seconds_int_config_t ∗const intConfig)

    This method configures the Time Seconds Interrupt with the configuration from the intConfig parameter.

- status_t RTC_DRV_ConfigureAlarm (uint32_t instance, rtc_alarm_config_t ∗const alarmConfig)

    This method configures the alarm with the configuration from the alarmConfig parameter.

- void RTC_DRV_GetAlarmConfig (uint32_t instance, rtc_alarm_config_t ∗alarmConfig)

    Get alarm configuration for RTC instance.

- bool RTC_DRV_IsAlarmPending (uint32_t instance)

    Check if alarm is pending.

- void RTC_DRV_ConvertSecondsToTimeDate (const uint32_t ∗seconds, rtc_timedate_t ∗const timeDate)

    Convert seconds to rtc_timedate_t structure.

- void RTC_DRV_ConvertTimeDateToSeconds (const rtc_timedate_t ∗const timeDate, uint32_t ∗const seconds)

    Convert seconds to rtc_timedate_t structure.

- bool RTC_DRV_IsYearLeap (uint16_t year)

    Check if the current year is leap.

- bool RTC_DRV_IsTimeDateCorrectFormat (const rtc_timedate_t ∗const timeDate)

    Check if the date time struct is configured properly.

- status_t RTC_DRV_GetNextAlarmTime (uint32_t instance, rtc_timedate_t ∗const alarmTime)

Gets the next alarm time.

- void RTC_DRV_IRQHandler (uint32_t instance)

    This method is the API's Interrupt handler for generic and alarm IRQ. It will handle the alarm repetition and calls the user callbacks if they are not NULL.

- void RTC_DRV_SecondsIRQHandler (uint32_t instance)

    This method is the API's Interrupt handler for RTC Second interrupt. This ISR will call the user callback if defined.

16.77.2   Data Structure Documentation

16.77.2.1   struct rtc_timedate_t

RTC Time Date structure Implements : rtc_timedate_t_Class.

Definition at line 99 of file rtc_driver.h.

Data Fields

- uint16_t year
- uint16_t month
- uint16_t day
- uint16_t hour
- uint16_t minutes
- uint8_t seconds

Field Documentation

16.77.2.1.1   uint16_t day

Day

Definition at line 103 of file rtc_driver.h.

16.77.2.1.2   uint16_t hour

Hour

Definition at line 104 of file rtc_driver.h.

16.77.2.1.3   uint16_t minutes

Minutes

Definition at line 105 of file rtc_driver.h.

16.77.2.1.4   uint16_t month

Month

Definition at line 102 of file rtc_driver.h.

16.77.2.1.5   uint8_t seconds

Seconds

Definition at line 106 of file rtc_driver.h.

16.77.2.1.6   uint16_t year

Year

Definition at line 101 of file rtc_driver.h.

16.77.2.2   struct rtc_init_config_t

RTC Initialization structure Implements : rtc_init_config_t_Class.

Definition at line 113 of file rtc_driver.h.

Data Fields

- uint8_t compensationInterval
- int8_t compensation
- rtc_clk_select_t clockSelect
- rtc_clk_out_config_t clockOutConfig
- bool updateEnable
- bool nonSupervisorAccessEnable

Field Documentation

16.77.2.2.1   rtc_clk_out_config_t clockOutConfig

RTC Clock Out Source

Definition at line 118 of file rtc_driver.h.

16.77.2.2.2   rtc_clk_select_t clockSelect

RTC Clock Select

Definition at line 117 of file rtc_driver.h.

16.77.2.2.3   int8_t compensation

Compensation Value

Definition at line 116 of file rtc_driver.h.

16.77.2.2.4   uint8_t compensationInterval

Compensation Interval

Definition at line 115 of file rtc_driver.h.

16.77.2.2.5   bool nonSupervisorAccessEnable

Enable writes to the registers in non Supervisor Mode

Definition at line 120 of file rtc_driver.h.

16.77.2.2.6   bool updateEnable

Enable changing the Time Counter Enable bit even if the Status register is locked

Definition at line 119 of file rtc_driver.h.

16.77.2.3   struct rtc_alarm_config_t

RTC alarm configuration Implements : rtc_alarm_config_t_Class.

Definition at line 127 of file rtc_driver.h.

Data Fields

- rtc_timedate_t alarmTime
- uint32_t repetitionInterval
- uint32_t numberOfRepeats
- bool repeatForever

- bool alarmIntEnable
- void(∗ alarmCallback)(void ∗callbackParam)
- void ∗ callbackParams

Field Documentation

16.77.2.3.1   void(∗ alarmCallback) (void ∗callbackParam)

Pointer to the user callback method.

Definition at line 134 of file rtc_driver.h.

16.77.2.3.2   bool alarmIntEnable

Enable alarm interrupt

Definition at line 133 of file rtc_driver.h.

16.77.2.3.3   rtc_timedate_t alarmTime

Alarm time

Definition at line 129 of file rtc_driver.h.

16.77.2.3.4   void∗ callbackParams

Pointer to the callback parameters.

Definition at line 135 of file rtc_driver.h.

16.77.2.3.5   uint32_t numberOfRepeats

Number of alarm repeats

Definition at line 131 of file rtc_driver.h.

16.77.2.3.6   bool repeatForever

Repeat forever if set, discard number of repeats

Definition at line 132 of file rtc_driver.h.

16.77.2.3.7   uint32_t repetitionInterval

Interval of repetition in sec

Definition at line 130 of file rtc_driver.h.

16.77.2.4   struct rtc_interrupt_config_t

RTC interrupt configuration. It is used to configure interrupt other than Time Alarm and Time Seconds interrupt Implements : rtc_interrupt_config_t_Class.

Definition at line 143 of file rtc_driver.h.

Data Fields

- bool overflowIntEnable
- bool timeInvalidIntEnable
- void(∗ rtcCallback)(void ∗callbackParam)
- void ∗ callbackParams

Field Documentation

16.77.2.4.1 void∗ callbackParams

Pointer to the callback parameters.

Definition at line 148 of file rtc_driver.h.

16.77.2.4.2 bool overflowIntEnable

Enable Time Overflow Interrupt

Definition at line 145 of file rtc_driver.h.

16.77.2.4.3 void(∗ rtcCallback) (void ∗callbackParam)

Pointer to the user callback method.

Definition at line 147 of file rtc_driver.h.

16.77.2.4.4 bool timeInvalidIntEnable

Enable Time Invalid Interrupt

Definition at line 146 of file rtc_driver.h.

16.77.2.5 struct rtc_seconds_int_config_t

RTC Seconds Interrupt Configuration Implements : rtc_seconds_int_config_t_Class.

Definition at line 155 of file rtc_driver.h.

Data Fields

- rtc_second_int_cfg_t secondIntConfig
- bool secondIntEnable
- void(∗ rtcSecondsCallback)(void ∗callbackParam)
- void ∗ secondsCallbackParams

Field Documentation

16.77.2.5.1 void(∗ rtcSecondsCallback) (void ∗callbackParam)

Pointer to the user callback method.

Definition at line 159 of file rtc_driver.h.

16.77.2.5.2 rtc_second_int_cfg_t secondIntConfig

Seconds Interrupt frequency

Definition at line 157 of file rtc_driver.h.

16.77.2.5.3 bool secondIntEnable

Seconds Interrupt enable

Definition at line 158 of file rtc_driver.h.

16.77.2.5.4 void∗ secondsCallbackParams

Pointer to the callback parameters.

Definition at line 160 of file rtc_driver.h.

16.77.2.6 struct rtc_register_lock_config_t

RTC Register Lock Configuration Implements : rtc_register_lock_config_t_Class.

Definition at line 167 of file rtc_driver.h.

Data Fields

- bool lockRegisterLock
- bool statusRegisterLock
- bool controlRegisterLock
- bool timeCompensationRegisterLock

Field Documentation

16.77.2.6.1   bool controlRegisterLock

Lock state of the Control Register

Definition at line 171 of file rtc_driver.h.

16.77.2.6.2   bool lockRegisterLock

Lock state of the Lock Register

Definition at line 169 of file rtc_driver.h.

16.77.2.6.3   bool statusRegisterLock

Lock state of the Status Register

Definition at line 170 of file rtc_driver.h.

16.77.2.6.4   bool timeCompensationRegisterLock

Lock state of the Time Compensation Register

Definition at line 172 of file rtc_driver.h.

16.77.3   Macro Definition Documentation

16.77.3.1   #define DAYS_IN_A_LEAP_YEAR (366U)

Definition at line 42 of file rtc_driver.h.

16.77.3.2   #define DAYS_IN_A_YEAR (365U)

Definition at line 41 of file rtc_driver.h.

16.77.3.3   #define HOURS_IN_A_DAY (24U)

Definition at line 40 of file rtc_driver.h.

16.77.3.4   #define MINS_IN_A_HOUR (60U)

Definition at line 39 of file rtc_driver.h.

16.77.3.5   #define SECONDS_IN_A_DAY (86400UL)

Definition at line 36 of file rtc_driver.h.

16.77.3.6   #define SECONDS_IN_A_HOUR (3600U)

Definition at line 37 of file rtc_driver.h.

16.77.3.7 #define SECONDS_IN_A_MIN (60U)

Definition at line 38 of file rtc_driver.h.

16.77.3.8 #define YEAR_RANGE_END (2099U)

Definition at line 44 of file rtc_driver.h.

16.77.3.9 #define YEAR_RANGE_START (1970U)

Definition at line 43 of file rtc_driver.h.

16.77.4 Enumeration Type Documentation

16.77.4.1 enum rtc_clk_out_config_t

RTC CLKOUT pin configuration Implements : rtc_clk_out_config_t_Class.

Enumerator

> *RTC_CLKOUT_DISABLED* Clock out pin is disabled
>
> *RTC_CLKOUT_SRC_TSIC* Output on RTC_CLKOUT as configured on Time seconds interrupt
>
> *RTC_CLKOUT_SRC_32KHZ* Output on RTC_CLKOUT of the 32KHz clock

Definition at line 66 of file rtc_driver.h.

16.77.4.2 enum rtc_clk_select_t

RTC clock select Implements : rtc_clk_select_t_Class.

Enumerator

> *RTC_CLK_SRC_OSC_32KHZ* RTC Prescaler increments using 32 KHz crystal
>
> *RTC_CLK_SRC_LPO_1KHZ* RTC Prescaler increments using 1KHz LPO

Definition at line 77 of file rtc_driver.h.

16.77.4.3 enum rtc_lock_register_select_t

RTC register lock Implements : rtc_lock_register_select_t_Class.

Enumerator

> *RTC_LOCK_REG_LOCK* RTC Lock Register lock
>
> *RTC_STATUS_REG_LOCK* RTC Status Register lock
>
> *RTC_CTRL_REG_LOCK* RTC Control Register lock
>
> *RTC_TCL_REG_LOCK* RTC Time Compensation Reg lock

Definition at line 87 of file rtc_driver.h.

16.77.4.4 enum rtc_second_int_cfg_t

RTC Seconds interrupt configuration Implements : rtc_second_int_cfg_t_Class.

Enumerator

> *RTC_INT_1HZ* RTC seconds interrupt occurs at 1 Hz
>
> *RTC_INT_2HZ* RTC seconds interrupt occurs at 2 Hz

RTC_INT_4HZ   RTC seconds interrupt occurs at 4 Hz

RTC_INT_8HZ   RTC seconds interrupt occurs at 8 Hz

RTC_INT_16HZ   RTC seconds interrupt occurs at 16 Hz

RTC_INT_32HZ   RTC seconds interrupt occurs at 32 Hz

RTC_INT_64HZ   RTC seconds interrupt occurs at 64 Hz

RTC_INT_128HZ   RTC seconds interrupt occurs at 128 Hz

Definition at line 50 of file rtc_driver.h.

16.77.5   Function Documentation

16.77.5.1   status_t RTC_DRV_ConfigureAlarm (uint32_t *instance,* rtc_alarm_config_t ∗const *alarmConfig*)

This method configures the alarm with the configuration from the alarmConfig parameter.

Parameters

`in`	*instance*	The number of the RTC instance used
`in`	*alarmConfig*	Pointer to the structure which holds the alarm configuration

Returns

   STATUS_SUCCESS if the configuration is successful or STATUS_ERROR if the alarm time is invalid.

Definition at line 934 of file rtc_driver.c.

16.77.5.2   void RTC_DRV_ConfigureFaultInt (uint32_t *instance,* rtc_interrupt_config_t ∗const *intConfig*)

This method configures fault interrupts such as:

- Time Overflow Interrupt

- Time Invalid Interrupt with the user provided configuration struct intConfig.

Parameters

`in`	*instance*	The number of the RTC instance used
`in`	*intConfig*	Pointer to the structure which holds the configuration

Returns

   None

Definition at line 877 of file rtc_driver.c.

16.77.5.3   status_t RTC_DRV_ConfigureRegisterLock (uint32_t *instance,* const rtc_register_lock_config_t ∗const *lockConfig*)

This method configures register lock for the corresponding RTC instance. Remember that all the registers are unlocked only by software reset or power on reset. (Except for CR that is unlocked only by POR).

Parameters

in	*instance*	The number of the RTC instance used
in	*lockConfig*	Pointer to the lock configuration structure

Returns

> STATUS_SUCCESS if the operation was successful, STATUS_ERROR if the Lock Register is locked.

Definition at line 427 of file rtc_driver.c.

16.77.5.4   void RTC_DRV_ConfigureSecondsInt (uint32_t *instance,* rtc_seconds_int_config_t ∗const *intConfig*)

This method configures the Time Seconds Interrupt with the configuration from the intConfig parameter.

Parameters

in	*instance*	The number of the RTC instance used
in	*intConfig*	Pointer to the structure which holds the configuration

Returns

> None

Definition at line 904 of file rtc_driver.c.

16.77.5.5   status_t RTC_DRV_ConfigureTimeCompensation (uint32_t *instance,* uint8_t *compInterval,* int8_t *compensation*)

This method configures time compensation. Data is passed by the compInterval and compensation parameters. For more details regarding coefficient calculation see the Reference Manual.

Parameters

in	*instance*	The number of the RTC instance used
in	*compInterval*	Compensation interval
in	*compensation*	Compensation value

Returns

> STATUS_SUCCESS if the operation was successful, STATUS_ERROR if the TC Register is locked.

Definition at line 502 of file rtc_driver.c.

16.77.5.6   void RTC_DRV_ConvertSecondsToTimeDate (const uint32_t ∗ *seconds,* rtc_timedate_t ∗const *timeDate*)

Convert seconds to rtc_timedate_t structure.

Parameters

in	*seconds*	Pointer to the seconds
out	*timeDate*	Pointer to the structure in which to store the result

Returns

> None

Definition at line 552 of file rtc_driver.c.

16.77.5.7   void RTC_DRV_ConvertTimeDateToSeconds (const rtc_timedate_t ∗const *timeDate,* uint32_t ∗const *seconds*)

Convert seconds to rtc_timedate_t structure.

Parameters

in	*timeDate*	Pointer to the source struct
out	*seconds*	Pointer to the variable in which to store the result

Returns

> None

Definition at line 646 of file rtc_driver.c.

16.77.5.8   status_t RTC_DRV_Deinit (uint32_t *instance*)

This function deinitializes the RTC instance. If the Control register is locked then this method returns STATUS_E↩
RROR.

Parameters

in	*instance*	The number of the RTC instance used

Returns

> STATUS_SUCCESS if the operation was successful or STATUS_ERROR if Control register is locked.

Definition at line 159 of file rtc_driver.c.

16.77.5.9   void RTC_DRV_GetAlarmConfig (uint32_t *instance,* rtc_alarm_config_t ∗ *alarmConfig*)

Get alarm configuration for RTC instance.

Parameters

in	*instance*	The number of the RTC instance used
out	*alarmConfig*	Pointer to the structure in which to store the alarm configuration

Returns

> None

Definition at line 988 of file rtc_driver.c.

16.77.5.10   status_t RTC_DRV_GetCurrentTimeDate (uint32_t *instance,* rtc_timedate_t ∗const *currentTime*)

Get current time and date from RTC instance.

Parameters

in	*instance*	The number of the RTC instance used
out	*currentTime*	Pointer to the variable in which to store the result

Returns

> STATUS_SUCCESS if the operation was successful, STATUS_ERROR if there was a problem.

Definition at line 328 of file rtc_driver.c.

16.77.5.11   void RTC_DRV_GetDefaultConfig (rtc_init_config_t ∗const *config*)

This function will set the default configuration values into the structure passed as a parameter.

Parameters

out		*config*	Pointer to the structure in which the configuration will be saved.

Returns

> None

Definition at line 195 of file rtc_driver.c.

16.77.5.12   status_t RTC_DRV_GetNextAlarmTime (uint32_t *instance,* rtc_timedate_t *∗const *alarmTime*)

Gets the next alarm time.

Parameters

in	*instance*	The number of the RTC instance used
out	*alarmTime*	Pointer to the variable in which to store the data

Returns

> STATUS_SUCCESS if the next alarm time is valid, STATUS_ERROR if there is no new alarm or alarm
> configuration specified.

Definition at line 1020 of file rtc_driver.c.

16.77.5.13   void RTC_DRV_GetRegisterLock (uint32_t *instance,* rtc_register_lock_config_t *∗const *lockConfig*)

Get which registers are locked for RTC instance.

Parameters

in	*instance*	The number of the RTC instance used
out	*lockConfig*	Pointer to the lock configuration structure in which to save the data

Returns

> None

Definition at line 474 of file rtc_driver.c.

16.77.5.14   void RTC_DRV_GetTimeCompensation (uint32_t *instance,* uint8_t ∗ *compInterval,* int8_t ∗ *compensation*)

This retrieves the time compensation coefficients and saves them on the variables referenced by the parameters.

Parameters

in	*instance*	The number of the RTC instance used
out	*compInterval*	Pointer to the variable in which to save the compensation interval
out	*compensation*	Pointer to the variable in which to save the compensation value

Returns

> None

Definition at line 535 of file rtc_driver.c.

16.77.5.15   status_t RTC_DRV_Init (uint32_t *instance,* const rtc_init_config_t ∗const *rtcUserCfg*)

This function initializes the RTC instance with the settings provided by the user via the rtcUserCfg parameter. The user must ensure that clock is enabled for the RTC instance used. If the Control register is locked then this method returns STATUS_ERROR. In order to clear the CR Lock the user must perform a power-on reset.

Parameters

in	*instance*	The number of the RTC instance used
in	*rtcUserCfg*	Pointer to the user's configuration structure

Returns

STATUS_SUCCESS if the operation was successful, STATUS_ERROR if Control is locked.

Definition at line 100 of file rtc_driver.c.

16.77.5.16   void RTC_DRV_IRQHandler (uint32_t *instance*)

This method is the API's Interrupt handler for generic and alarm IRQ. It will handle the alarm repetition and calls the user callbacks if they are not NULL.

Parameters

in	*instance*	RTC instance used

Returns

None

Definition at line 774 of file rtc_driver.c.

16.77.5.17   bool RTC_DRV_IsAlarmPending (uint32_t *instance*)

Check if alarm is pending.

Parameters

in	*instance*	The number of the RTC instance used

Returns

True if the alarm has occurred, false if not

Definition at line 1003 of file rtc_driver.c.

16.77.5.18   bool RTC_DRV_IsTimeDateCorrectFormat (const rtc_timedate_t ∗const *timeDate*)

Check if the date time struct is configured properly.

Parameters

in	*timeDate*	Structure to check to check

Returns

True if the time date is in the correct format, false if not

Definition at line 696 of file rtc_driver.c.

16.77.5.19   bool RTC_DRV_IsYearLeap (uint16_t *year*)

Check if the current year is leap.

Parameters

in	*year*	Year to check

Returns

True if the year is leap, false if not

Definition at line 738 of file rtc_driver.c.

16.77.5.20   void RTC_DRV_SecondsIRQHandler (uint32_t *instance*)

This method is the API's Interrupt handler for RTC Second interrupt. This ISR will call the user callback if defined.

Parameters

in	*instance*	RTC instance used

Returns

None

Definition at line 851 of file rtc_driver.c.

16.77.5.21   status_t RTC_DRV_SetTimeDate (uint32_t *instance,* const rtc_timedate_t ∗const *time*)

Set time and date for RTC instance. The user must stop the counter before using this function. Otherwise it will return an error.

Parameters

in	*instance*	The number of the RTC instance used
in	*time*	Pointer to the variable in which the time is stored

Returns

STATUS_SUCCESS if the operation was successful, STATUS_ERROR if the time provided was invalid or if the counter was not stopped.

Definition at line 383 of file rtc_driver.c.

16.77.5.22   status_t RTC_DRV_StartCounter (uint32_t *instance*)

Start RTC instance counter. Before calling this function the user should use RTC_DRV_SetTimeDate to configure the start time.

Parameters

in	*instance*	The number of the RTC instance used

Returns

STATUS_SUCCESS if the operation was successful, STATUS_ERROR if the counter cannot be enabled or is already enabled.

Definition at line 266 of file rtc_driver.c.

16.77.5.23   status_t RTC_DRV_StopCounter (uint32_t *instance*)

Disable RTC instance counter.

Parameters

in	*instance*	The number of the RTC instance used

Returns

 STATUS_SUCCESS if the operation was successful, STATUS_ERROR if the counter could not be stopped.

Definition at line 297 of file rtc_driver.c.

16.78 Real Time Clock Driver (RTC)

16.78.1 Detailed Description

The S32 SDK provides the Peripheral Driver for the Real Time Clock (RTC) module of S32 SDK devices.
.

Hardware background

The Real Time Clock Module is a independent timer that keeps track of the exact date and time with no software overhead, with low power usage.

Features of the RTC module include:

- 32-bit seconds counter with roll-over protection and 32-bit alarm

- 16-bit prescaler with compensation that can correct errors between 0.12 ppm and 3906 ppm

- Option to increment prescaler using the LPO (prescaler increments by 32 every clock edge)

- Register write protection

- Lock register requires POR or software reset to enable write access

- Configurable 1, 2, 4, 8, 16, 32, 64 or 128 Hz square wave output with optional interrupt

- Alarm interrupt configured by the driver automatically refreshes alarm time configured by the user

- User interrupt handlers can be configured for all interrupts

How to use the RTC driver in your application

In order to be able to use the RTC in your application, the first thing to do is initializing it with the desired configuration. This is done by calling the **RTC_DRV_Init** function. One of the arguments passed to this function is the configuration which will be used for the RTC instance, specified by the **rtc_init_config_t** structure.

The **rtc_init_config_t** structure allows you to configure the following:

- RTC clock source (32 KHz clock or 1 KHz LPO clock)

- Clock Out pin configuration (Clock OUT pin source)

- Compensation (Interval and value)

- Update enable - this allows updates to Time Counter Enable bit if the Status Register under limited conditions

- Enable non supervisor writes to the registers

The **rtc_seconds_int_config_t** structure configures the **time seconds interrupt**. To setup an interrupt every seconds you have to configure the structure mentioned with the following parameters:

- Frequency of the interrupt

- Interrupt Handler

- If needed - interrupt handler parameters

An alarm is configured with **rtc_alarm_config_t** structure, which is described by the following parameters:

- Alarm time in date-time format

- Interval of alarm repeat in seconds

- Number of alarm repeats (use 0 if the alarm is not recursive)

- Repeat forever field (if set, the number of repeats field will be ignored)

- Alarm interrupt enable

- Alarm interrupt handler

- Alarm interrupt handler parameters

Note If the alarm interrupt is not enabled, the user must make the updates of the alarm time manually.

After the RTC_DRV_Init call and, if needed, alarm and other configurations the RTC counter is started by calling RTC_DRV_Enable, with start time as parameter in **rtc_timedate_t** format.

To get the current time and date you can call RTC_DRV_GetCurrentTimeDate function, this method will get the seconds from the Time Seconds Register and will convert into human readable format as rtc_timedate_t.

Example

```
void rtcAlarmCallback(void)
{
    rtc_timedate_t currentTime;
    RTC_DRV_GetCurrentTimeDate(0U, &currentTime);

    /* Do something with the time and date */
}

int main()
{
    /* rtcTimer1 configuration structure */
    const rtc_init_config_t rtcTimer1_Config0 =
    {
        /* Time compensation interval */
        .compensationInterval    =   0U,
        /* Time compensation value */
        .compensation            =   0,
        /* RTC Clock Source is 32 KHz crystal */
        .clockSelect             =   RTC_CLK_SRC_OSC_32KHZ,
        /* RTC Clock Out is 32 KHz clock */
        .clockOutConfig          =   RTC_CLKOUT_SRC_32KHZ,
        /* Update of the TCE bit is not allowed */
        .updateEnable            =   false,
        /* Non-supervisor mode write accesses are not supported and generate
         * a bus error.
         */
        .nonSupervisorAccessEnable =   false
    };

    /* RTC Initial Time and Date */
    rtc_timedate_t      rtcStartTime =
    {
        /* Year */
        .year      =   2016U,
        /* Month */
        .month     =   01U,
        /* Day */
        .day       =   01U,
        /* Hour */
        .hour      =   00U,
        /* Minutes */
        .minutes   =   00U,
        /* Seconds */
        .seconds   =   00U
    };

    /* rtcTimer1 Alarm configuration 0 */
    rtc_alarm_config_t  alarmConfig0 =
    {
        /* Alarm Date */
        .alarmTime          =
            {
                /* Year    */
                .year      =   2016U,
                /* Month   */
                .month     =   01U,
                /* Day     */
                .day       =   01U,
                /* Hour    */
                .hour      =   00U,
                /* Minutes */
                .minutes   =   00U,
                /* Seconds */
                .seconds   =   03U,
```

```
        },

    /* Alarm repeat interval */
    .repetitionInterval  =       3UL,

    /* Number of alarm repeats */
    .numberOfRepeats     =       0UL,

    /* Repeat alarm forever */
    .repeatForever       =       true,

    /* Alarm interrupt disabled */
    .alarmIntEnable      =       true,

    /* Alarm interrupt handler */
    .alarmCallback       =       (void *)rtcAlarmCallback,

    /* Alarm interrupt handler parameters */
    .callbackParams      =       (void *)NULL
  };

    /* Call the init function */
    RTC_DRV_Init(0UL, &rtcInitConfig);

    /* Set the time and date */
    RTC_DRV_SetTimeDate(0UL, &rtcStartTime);

    /* Start RTC counter */
    RTC_DRV_StartCounter(0UL);

    /* Configure an alarm every 3 seconds */
    RTC_DRV_ConfigureAlarm(0UL, &rtcAlarmConfig0);

    while(1);
}
```

Important Notes

- Before using the RTC driver the module clock must be configured

- The driver enables the interrupts for the corresponding RTC module, but any interrupt priority must be done by the application

- The board specific configurations must be done prior to driver calls; the driver has no influence on the functionality of the clockout pin - they must be configured by application

- If Non-supervisor mode write accesses are supported you need to set APIS to allow usermode access to RTC Memory Space

Modules

- Real Time Clock Driver

    Real Time Clock Driver Peripheral Driver.

16.79   S32K116 SoC Header file

16.79.1   Detailed Description

This module covers the S32K116 SoC Header file.

Modules

- Backward Compatibility Symbols for S32K116

    This module covers backward compatibility symbols.
- Interrupt vector numbers for S32K116

    This module covers interrupt number allocation.
- Peripheral access layer for S32K116

    This module covers all memory mapped register available on SoC.

16.80 S32K116 System Files

This module covers the SoC support file for S32K116.

SystemInit method is called automatically from start-up code to do the minimum setup of the SoC. It disables the watchdog, enables FPU and the power mode protection if the corresponding feature macro is enabled.

SystemCoreClockUpdate method can be used at any time to update SystemCoreClock. It evaluates the clock register settings and calculates the current core clock.

SystemSoftwareReset method initiates a system reset.

16.81   Schedule management

16.81.1   Detailed Description

This group contains APIs that help users manage schedule tables in master node only.

Functions

- l_u8 l_sch_tick (l_ifc_handle iii)

  This function follows a schedule. When a frame becomes due, its transmission is initiated. When the end of the current schedule is reached, this function starts again at the beginning of the schedule.

- void l_sch_set (l_ifc_handle iii, l_schedule_handle schedule_iii, l_u8 entry)

  Set up the next schedule to be followed by the l_sch_tick function for a certain interface. The new schedule will be activated as soon as the current schedule reaches its next schedule entry point.

16.81.2   Function Documentation

16.81.2.1   void l_sch_set (l_ifc_handle *iii,* l_schedule_handle *schedule_iii,* l_u8 *entry*)

Set up the next schedule to be followed by the l_sch_tick function for a certain interface. The new schedule will be activated as soon as the current schedule reaches its next schedule entry point.

Parameters

in	*iii*	Interface name
in	*schedule_iii*	Schedule table for interface
in	*entry*	Entry to be set

Returns

    void

Definition at line 76 of file lin_common_api.c.

16.81.2.2   l_u8 l_sch_tick (l_ifc_handle *iii*)

This function follows a schedule. When a frame becomes due, its transmission is initiated. When the end of the current schedule is reached, this function starts again at the beginning of the schedule.

Parameters

in	*Interface*	name

Returns

    Operation status

- Zero: if the next call of l_sch_tick will not start transmission of a frame.
- Non-Zero: if the next call of l_sch_tick will start transmission of a frame. The return value will in this case be the next schedule table entry's number (counted from the beginning of the schedule table) in the schedule table. The return value will be in range 1 to N if the schedule table has N entries.

Definition at line 243 of file lin_common_api.c.

16.82 Signal interaction

This group contains APIs that help users interract with signals of LIN node.

16.83 SoC Header file (SoC Header)

16.83.1 Detailed Description

This module covers SoC Header file.

This section describes the functionality supported by the header file. For usage please see soc_header_usage

Modules

- S32K116 SoC Header file

    This module covers the S32K116 SoC Header file.

16.84 SoC Support

16.84.1 Detailed Description

This module covers SoC support files.

This section describes the files that are used for supporting various SoCs.

The support files are:

1. Linker files

2. Start-up files

3. SVD file

4. Header files

Linker files

Linker files are used to control the linkage part of the project compilation and contain details regarding the following:

1. memory areas definition (type and ranges)

2. data and code segments definition and their mapping to the memory areas.

linker configuration files are provided for all supported linkers. Please see Build Tools for details.

Start-up files

Start-up files are used to control the SoC bring-up part and contain:

1. interrupt vector allocation

2. start-up code and routines

Start-up files are provided for all supported compilers. Please see Build Tools for details.

SVD file

SVD file contains details about registers and can be used with an IDE to allow mapping of memory location to the register definition and information.

Header file

For each SoC there are two header files provided in the SDK:

1. <SoC_name>.h

2. <SoC_name>_features.h

The <SoC_name>.h file contains information related to registers that is used by the SDK drivers and code. The <SoC_name>_features.h contains information related to the integration of modules in the SoC.

Modules

- S32K116 System Files

    This module covers the SoC support file for S32K116.

16.85   System Basis Chip Driver (SBC) - UJA1169 Family

16.85.1   Detailed Description

System Basis Chip driver is a middleware driver for SBC settings and control.

Hardware background

The UJA1169 is a mini high-speed CAN System Basis Chip (SBC) containing an ISO 11898-2:201x (upcoming merged ISO 11898-2/5/6) compliant HS-CAN transceiver and an integrated 5 V or 3.3 V 250 mA scalable supply (V1) for a microcontroller and/or other loads. It also features a watchdog and a Serial Peripheral Interface (SPI). The UJA1169 can be operated in very low-current Standby and Sleep modes with bus and local wake-up capability. The UJA1169 comes in six variants. The UJA1169TK, UJA1169TK/F, UJA1169TK/X and UJA1169TK/X/F contain a 5 V regulator (V1). V1 is a 3.3 V regulator in the UJA1169TK/3 and the UJA1169TK/F/3. The UJA1169TK, UJ↩A1169TK/F, UJA1169TK/3 and UJA1169TK/F/3 variants feature a second on-board 5 V regulator (V2) that supplies the internal CAN transceiver and can also be used to supply additional on-board hardware. The UJA1169TK/X and UJA1169TK/X/F are equipped with a 5 V supply (VEXT) for off-board components. VEXT is short-circuit proof to the battery, ground and negative voltages. The integrated CAN transceiver is supplied internally via V1, in parallel with the microcontroller. The UJA1169xx/F variants support ISO 11898-6:2013 and ISO 11898-2:201x compliant CAN partial networking with a selective wake-up function incorporating CAN FD-passive. CAN FD-passive is a feature that allows CAN FD bus traffic to be ignored in Sleep/Standby mode. CAN FD-passive partial networking is the perfect fit for networks that support both CAN FD and classic CAN communications. It allows normal CA↩N controllers that do not need to communicate CAN FD messages to remain in partial networking Sleep/Standby mode during CAN FD communication without generating bus errors. The UJA1169 implements the standard C↩AN physical layer as defined in the current ISO11898 standard (-2:2003, -5:2007, -6:2013). Pending the release of the upcoming version of ISO11898-2:201x including CAN FD, additional timing parameters defining loop delay symmetry are included. This implementation enables reliable communication in the CAN FD fast phase at data rates up to 2 Mbit/s. A dedicated LIMP output pin is provided to flag system failures. A number of configuration settings are stored in non-volatile memory. This arrangement makes it possible to configure the power-on and limp-home behavior of the UJA1169 to meet the requirements of different applications.

How to use SBC driver in your application

In order to set up SBC device the user needs to configure sbc_int_config_t structure in which are included following structures: sbc_regulator_ctr_t, sbc_wtdog_ctr_t, sbc_mode_mc_t, sbc_fail_safe_lhc_t, sbc_sys_evnt_t, sbc_↩lock_t, sbc_can_conf_t, sbc_wake_t These nested structures correspond to individual registers. The sbc_int_↩config_t structure is passed as a parameter to Init function to initialize SBC device. The rest of the functions are related to individual registers.

Initialization

The initialization function is responsible for setting up the UJA1169, according to user configuration data which is passed as parameter. The initialization function takes another parameter which is an instance of SPI used for communication with UJA1169. The initialization function configures all SBC registers except factories configuration set up in non volatile memory, (Start up control and SBC configuration register.)

Mode transition

SBC_SetMode performs software transition from one mode to another. The transition is achieved by writing to mode control register. The event capture registers are cleared before device is moved to sleep mode.

Writing to registers

In order to write to registers, there are several methods dedicated to some specific registers. These methods (names starting with SBC_Set) take a value or a pointer to structure containing values to be written to particular registers as a parameter. Besides these methods there is also a method SBC_DataTransfer which is common to reading and writing to all registers. It takes three parameters. The first one is an address of a register to be written. Addresses of registers are defined in sbc_register_ t enum. The second argument is pointer to a value which should be sent to a register. The last argument is used for register reading only and its value is unused in this case. NULL

pointer is used when parameter is unused.

Reading from registers

Content of a register is read by method SBC_DataTransfer, which provides both reading and writing to all registers. This method has three arguments. The first one is an address of a register to be read from, the third one is a pointer to a variable where the content of a register will be stored. Second argument is used for the register writing only and it should be NULL in this case. Addresses of registers are defined in enum sbc_register_t. Several methods to reading specific control and status register are available similarly to the register writing. Their names start with SBC_Get.

Reading status registers

Content of status register can be read by method SBC_DataTransfer or using appropriate function which starts with SBC_Get and finishes with Status. Event capture registers must be cleared using SBC_CleanEvents by setting to 1 appropriate status. For clear all events set all statuses to 1 or reading all event capture statuses using SBC_↩ GetEventsStatus before.

There are several functions which read status and store it to structure. The Table 3 sumarize which function reads appropriate status register.

Function name	Status register
SBC_GetMainStatus	Main status, Watchdog status
SBC_GetSupplyStatus	V2/VEXT status, V1 status
SBC_GetCanStatus	CAN transceiver status, CAN partial networking error, CAN partial networking status, CAN oscillator status, CAN-bus silence status, VCAN status, CAN failure status
SBC_GetWakeStatus	WAKE pin status
SBC_GetEventsStatus	Global event status, System event status, Supply event status, Transceiver event status, WAKE pin event status
SBC_GetAllStatus	Read all statuses from this table

Reading and writing non-volatile SBC configuration

The UJA1169 contains Multiple Time Programmable Non-Volatile (MTPNV) memory cells that allow some of the default device settings to be reconfigured. This non-volatile memory has limited write access. Programming of the NVM registers is performed in two steps. First, the required values are written. In the second step, reprogramming is confirmed by writing the correct CRC value to the MTPNV CRC control register. This memory is accessed by SBC_GetFactoriesSettings and SBC_ChangeFactoriesSettings methods. The only parameter is a pointer to sbc↩ _factories_conf_t data structure, which should be written to NTPNVM or where should be stored data read out from the NTPNVM. If the device has been programmed previously, the factory presets may need to be restored before reprogramming can begin. When the factory presets have been restored successfully, a system reset is generated automatically and UJA1169 switches back to Forced Normal mode. If SBC_ChangeFactoriesSettings method returns an error "SBC_UJA_NVN_ERROR" it means device was preconfigured from default settings and it is not possible to write to non-volatile memory. Restore factory preset values is needed. Factory preset values are restored if the following conditions apply continuously for at least td(MTPNV) during battery power-up: • pin RSTN is held LOW • CANH is pulled up to VBAT • CANL is pulled down to GND Now SBC_ChangeFactoriesSettings can be used for change factory preset values to custom configuration.

Error tracking

If an error during the R/W operations to UJA1169 registers occurs, the driver keeps track of it. If a method returns status different from SBC_UJA_STAT_SUCCESS the status represents the type of error from sbc_status_t enum.

Example code snippets (for FRDM PK144-Q100 freedom board).

Write to Regulator control registers example

This example source code snippet shows how to configure Regulator control register. Power distribution control (PDC), V2/VEXT configuration (V2C/ VETXT), V1 reset threshold can be configured by writing to Regulator Control register. Note (V2 can be set for models: UJA1169TK, UJA1169TK/3, UJA1169TK/F and UJA1169TK/F/3): (VEXT can be set for models UJA1169TK/X and UJA1169TK/X/F). For more info read function description.

```c
int main(void)
{
    ...

    sbc_status_t status = SBC_UJA_STAT_SUCCESS;
    sbc_regulator_ctr_t regulator;
    regulator.regulator.pdc = SBC_UJA_REGULATOR_PDC_HV;
    regulator.regulator.v2c = SBC_UJA_REGULATOR_V2C_OFF;
    regulator.regulator.v1rtc = SBC_UJA_REGULATOR_V1RTC_80;

    regulator.supplyEvnt.v2oe = SBC_UJA_SUPPLY_EVNT_V2OE_EN;
    regulator.supplyEvnt.v2ue = SBC_UJA_SUPPLY_EVNT_V2UE_EN;
    regulator.supplyEvnt.v1ue = SBC_UJA_SUPPLY_EVNT_V1UE_DIS;

    status = SBC_SetVreg(&regulator);

    if(status != SBC_UJA_STAT_SUCCESS)
    {
        /* Do something here. */
    }

    ...
}
```

Read from Regulator control registers example

This example source code snippet shows how to read from Regulator control registers. Reading Regulator control register gives information about Power distribution control (PDC), V2/VEXT configuration (V2C/ VETXT), V1 reset threshold current configuration. Using this method can be useful for check if the Regulator control register is configured correctly. For more info read function description.

```c
int main(void)
{
    ...

    sbc_status_t status = SBC_UJA_STAT_SUCCESS;
    sbc_regulator_ctr_t regulator;

    status = SBC_GetVreg(&regulator);

    if(status == SBC_UJA_STAT_SUCCESS)
    {
        if(regulator.supplyEvnt.v2oe ==
        SBC_UJA_SUPPLY_EVNT_V2OE_EN)
        {
            /* Do something here. */
        }
    }

    ...
}
```

Reading all device status example

This example source code snippet shows how to read all SBC device statuses in one function. Variable allStatuses contains these registers: Main status register, Watchdog status register, Supply voltage status register, Transceiver status register, WAKE pin status register, Event capture registers. For more info read function description.

```c
int main(void)
{
    ...

    sbc_status_t status = SBC_UJA_STAT_SUCCESS;
    sbc_status_group_t allStatuses;

    while(1){

        status = SBC_GetAllStatus(&allStatuses);
```

```
        if(status == SBC_UJA_STAT_SUCCESS)
        {
            if(allStatuses.trans.cbss == SBC_UJA_TRANS_STAT_CBSS_ACT)
            {
                /* Do something here. */
            }

            if(allStatuses.supply.v1s == SBC_UJA_SUPPLY_STAT_V1S_VAB)
            {
                /* Do something here. */
            }

            ...

            /* Periodically feed watchdog (anytime in watchdog period in case of timeout watchdog mode). */
            SBC_FeedWatchdog();
        }
    }
}
```

Reading Transceiver device status example

This example source code snippet shows how to read Transceiver device status from SBC. It contains CA←
N transceiver status, CAN partial networking error, CAN partial networking status, CAN oscillator status, CAN-bus
silence status, VCAN status, CAN failure status. For more info read function description. Note similar approach can
be used for reading other status using different SBC_Get∗Status.

```
int main(void)
{
    ...

    sbc_status_t status = SBC_UJA_STAT_SUCCESS;
    sbc_trans_stat_t transStatus;

    while(1){

        status = SBC_GetCanStatus(&transStatus);

        if(status == SBC_UJA_STAT_SUCCESS)
        {
            if(transStatus.cbss == SBC_UJA_TRANS_STAT_CBSS_ACT)
            {
                /* Do something here. */
            }

            ...

            /* Periodically feed watchdog (anytime in watchdog period in case of timeout watchdog mode). */
            SBC_FeedWatchdog();
        }
    }
}
```

Change factories settings

This example source code snippet shows how to change factory preset value od non-volatile memmory. Device
must be set to fatory preset. For more info read function description.

```
int main(void)
{
    ...

    sbc_status_t status = SBC_UJA_STAT_SUCCESS;
    sbc_factories_conf_t factories;

    status = SBC_GetFactoriesSettings(&factories);

    factories.control.fnmc = SBC_UJA_SBC_SDMC_EN;
    factories.control.sdmc = SBC_UJA_SBC_SDMC_DIS;
    factories.startUp.rlc = SBC_UJA_START_UP_RLC_20_25p0;

    if(status == SBC_UJA_STAT_SUCCESS)
    {
        status = SBC_ChangeFactoriesSettings(&factories);
    }

    if(status != SBC_UJA_STAT_SUCCESS)
```

```
    {
        /* Do something here. */
    }
}
```

Modules

- UJA1169 SBC Driver

16.86 TRGMUX Driver

16.86.1 Detailed Description

Trigger MUX Control Peripheral Driver. The TRGMUX introduces an extremely flexible methodology for connecting various trigger sources to multiple pins/peripherals.

The S32 SDK provides Peripheral Drivers for the Trigger MUX Control (TRGMUX) module of S32 SDK devices.

Overview

This section describes the programing interface of the TRGMUX driver. The TRGMUX driver configures the TR↩GMUX (Trigger Mux Control). The Trigger MUX module allows software to configure the trigger inputs for various peripherals.

TRGMUX Driver model building

TRGMUX can be seen as a collection of muxes, each mux allowing to select one output from a list of input signals that are common to all muxes. The TRGMUX registers are identical as structure and all bitfields can be read/written using the TRGMUX driver API.

TRGMUX Initialization

The **TRGMUX_DRV_Init()** function is used to initialize the TRGMUX IP. The function receives as parameter a pointer to the trgmux_user_config_t structure. This structure contains a variable number of mappings between a trgmux trigger source and a trgmux target modules.

TRGMUX API

After initialization, the driver allows the reconfiguration of the source trigger for a given target module using **TR↩GMUX_DRV_SetTrigSourceForTargetModule()**. Also, by using **TRGMUX_DRV_SetLockForTargetModule()**, a given target module can be locked, such that it cannot be updated until a reset.

Data Structures

- struct trgmux_inout_mapping_config_t

    Configuration structure for pairing source triggers with target modules. More...
- struct trgmux_user_config_t

    User configuration structure for the TRGMUX driver. More...

Functions

- status_t TRGMUX_DRV_Init (const uint32_t instance, const trgmux_user_config_t ∗const trgmuxUserConfig)

    Initialize a TRGMUX instance for operation.
- status_t TRGMUX_DRV_Deinit (const uint32_t instance)

    Reset to default values the source triggers corresponding to all target modules, if none of the target modules is locked.
- status_t TRGMUX_DRV_SetTrigSourceForTargetModule (const uint32_t instance, const trgmux_trigger_↩source_t triggerSource, const trgmux_target_module_t targetModule)

    Configure a source trigger for a selected target module.
- trgmux_trigger_source_t TRGMUX_DRV_GetTrigSourceForTargetModule (const uint32_t instance, const trgmux_target_module_t targetModule)

    Get the source trigger configured for a target module.
- void TRGMUX_DRV_SetLockForTargetModule (const uint32_t instance, const trgmux_target_module_↩t targetModule)

    Locks the TRGMUX register of a target module.
- bool TRGMUX_DRV_GetLockForTargetModule (const uint32_t instance, const trgmux_target_module_↩t targetModule)

Get the Lock bit status of the TRGMUX register of a target module.

16.86.2   Data Structure Documentation

16.86.2.1   struct trgmux_inout_mapping_config_t

Configuration structure for pairing source triggers with target modules.

Use an instance of this structure to define a TRGMUX link between a trigger source and a target module. This structure is used by the user configuration structure.

Implements : trgmux_inout_mapping_config_t_Class

Definition at line 69 of file trgmux_driver.h.

Data Fields

- trgmux_trigger_source_t triggerSource
- trgmux_target_module_t targetModule
- bool lockTargetModuleReg

Field Documentation

16.86.2.1.1   bool lockTargetModuleReg

if true, the LOCK bit of the target module register will be set by TRGMUX_DRV_INIT(), after the current mapping is configured

Definition at line 73 of file trgmux_driver.h.

16.86.2.1.2   trgmux_target_module_t targetModule

selects one of the TRGMUX target modules

Definition at line 72 of file trgmux_driver.h.

16.86.2.1.3   trgmux_trigger_source_t triggerSource

selects one of the TRGMUX trigger sources

Definition at line 71 of file trgmux_driver.h.

16.86.2.2   struct trgmux_user_config_t

User configuration structure for the TRGMUX driver.

Use an instance of this structure with the TRGMUX_DRV_Init() function. This enables configuration of TRGMUX with the user defined mappings between inputs (source triggers) and outputs (target modules), via a single function call.

Implements : trgmux_user_config_t_Class

Definition at line 85 of file trgmux_driver.h.

Data Fields

- uint8_t numInOutMappingConfigs
- const trgmux_inout_mapping_config_t ∗ inOutMappingConfig

Field Documentation

16.86.2.2.1   const trgmux_inout_mapping_config_t∗ inOutMappingConfig

pointer to array of in-out mapping structures

Definition at line 88 of file trgmux_driver.h.

16.86.2.2.2   uint8_t numInOutMappingConfigs

number of in-out mappings defined in TRGMUX configuration

Definition at line 87 of file trgmux_driver.h.

16.86.3   Function Documentation

16.86.3.1   status_t TRGMUX_DRV_Deinit (const uint32_t *instance*)

Reset to default values the source triggers corresponding to all target modules, if none of the target modules is locked.

Parameters

in	*instance*	The TRGMUX instance number.

Returns

> Execution status:
> STATUS_SUCCESS
> STATUS_ERROR - if at least one of the target module register is locked.

Definition at line 118 of file trgmux_driver.c.

16.86.3.2   bool TRGMUX_DRV_GetLockForTargetModule (const uint32_t *instance,* const trgmux_target_module_t *targetModule*)

Get the Lock bit status of the TRGMUX register of a target module.

This function gets the value of the LK bit from the TRGMUX register corresponding to the selected target module.

Parameters

in	*instance*	The TRGMUX instance number.
in	*targetModule*	One of the values in the trgmux_target_module_t enumeration

Returns

> true - if the selected targetModule register is locked
> false - if the selected targetModule register is not locked

Definition at line 207 of file trgmux_driver.c.

16.86.3.3   trgmux_trigger_source_t TRGMUX_DRV_GetTrigSourceForTargetModule (const uint32_t *instance,* const trgmux_target_module_t *targetModule*)

Get the source trigger configured for a target module.

This function returns the TRGMUX source trigger linked to a selected target module.

Parameters

in	*instance*	The TRGMUX instance number.
in	*targetModule*	One of the values in the trgmux_target_module_t enumeration.

Returns

> Enum value corresponding to the trigger source configured for the selected target module.

Definition at line 172 of file trgmux_driver.c.

16.86.3.4   status_t TRGMUX_DRV_Init (const uint32_t *instance,* const trgmux_user_config_t ∗const *trgmuxUserConfig*)

Initialize a TRGMUX instance for operation.

This function first resets the source triggers of all TRGMUX target modules to their default values, then configures the TRGMUX with all the user defined in-out mappings. If at least one of the target modules is locked, the function will not change any of the TRGMUX target modules and return error code. This example shows how to set up the trgmux_user_config_t parameters and how to call the TRGMUX_DRV_Init() function with the required parameters:

```
1 trgmux_user_config_t          trgmuxConfig;
2 trgmux_inout_mapping_config_t    trgmuxInoutMappingConfig[] =
3 {
4     {TRGMUX_TRIG_SOURCE_TRGMUX_IN9,     TRGMUX_TARGET_MODULE_DMA_CH0,     false},
5     {TRGMUX_TRIG_SOURCE_FTM1_EXT_TRIG,  TRGMUX_TARGET_MODULE_TRGMUX_OUT4, true}
6 };
7
8 trgmuxConfig.numInOutMappingConfigs = 2;
9 trgmuxConfig.inOutMappingConfig     = trgmuxInoutMappingConfig;
10
11 TRGMUX_DRV_Init(instance, &trgmuxConfig);
```

Parameters

in	*instance*	The TRGMUX instance number.
in	*trgmuxUser↩ Config*	Pointer to the user configuration structure.

Returns

    Execution status:
    STATUS_SUCCESS
    STATUS_ERROR - if at least one of the target module register is locked.

Definition at line 75 of file trgmux_driver.c.

16.86.3.5   void TRGMUX_DRV_SetLockForTargetModule (const uint32_t *instance,* const trgmux_target_module_t *targetModule*)

Locks the TRGMUX register of a target module.

This function sets the LK bit of the TRGMUX register corresponding to the selected target module. Please note that some TRGMUX registers can contain up to 4 SEL bitfields, meaning that these registers can be used to configure up to 4 target modules independently. Because the LK bit is only one per register, the configuration of all target modules referred from that register will be locked.

Parameters

in	*instance*	The TRGMUX instance number.
in	*targetModule*	One of the values in the trgmux_target_module_t enumeration

Definition at line 189 of file trgmux_driver.c.

16.86.3.6   status_t TRGMUX_DRV_SetTrigSourceForTargetModule (const uint32_t *instance,* const trgmux_trigger_source_t *triggerSource,* const trgmux_target_module_t *targetModule*)

Configure a source trigger for a selected target module.

This function configures a TRGMUX link between a source trigger and a target module, if the requested target module is not locked.

Parameters

in	*instance*	The TRGMUX instance number.
in	*triggerSource*	One of the values in the trgmux_trigger_source_t enumeration
in	*targetModule*	One of the values in the trgmux_target_module_t enumeration

Returns

Execution status:
STATUS_SUCCESS
STATUS_ERROR - if requested target module is locked

Definition at line 139 of file trgmux_driver.c.

16.87   Transport layer API

16.87.1   Detailed Description

Transport layer stands between the application layer and the core API layer.

This layer consists the implementation of data transportation which contains one or more LIN frames. It is situated between the application layer and the core API layer including LIN2.1 TL API and LIN TL J2602. This layer provides APIs for the transport protocol, node configuration and diagnostic services. For LIN 2.1, all components will be extended from LIN 2.0 specification. The node configuration for J2602 implements only some functions of LIN 2.0 specification.

Modules

- Common Transport Layer API

    Contains Transport Layer APIs that used for both protocols LIN 2.1 and J2602.
- J2602 Transport Layer specific API

    Contains Transport Layer APIs that only used for J2602 protocol.

16.88 UJA1169 SBC Driver

16.88.1 Detailed Description

Data Structures

- struct sbc_wtdog_ctr_t

    Watchdog control register structure. Watchdog configuration structure. More...

- struct sbc_sbc_t

    SBC configuration control register structure. Two operating modes have a major impact on the operation of the watchdog: Forced Normal mode and Software Development mode (Software Development mode is provided for test and development purposes only and is not a dedicated SBC operating mode; the UJA1169 can be in any functional operating mode with Software Development mode enabled). These modes are enabled and disabled via bits FNMC and SDMC respectively in the SBC configuration control register. Note that this register is located in the non-volatile memory area. The watchdog is disabled in Forced Normal mode (FNM). In Software Development mode (SDM), the watchdog can be disabled or activated for test and software debugging purposes. More...

- struct sbc_start_up_t

    Start-up control register structure. This structure contains settings of RSTN output reset pulse width and V2/VEXT start-up control. More...

- struct sbc_regulator_t

    Regulator control register structure. This structure set power distribution control, V2/VEXT configuration, set V1 reset threshold. More...

- struct sbc_supply_evnt_t

    Supply event capture enable register structure. This structure enables or disables detection of V2/VEXT overvoltage, undervoltage and V1 undervoltage enable. More...

- struct sbc_sys_evnt_t

    System event capture enable register structure. This structure enables or disables overtemperature warning, SPI failure enable. More...

- struct sbc_can_ctr_t

    CAN control register structure. This structure configure CAN peripheral behavior. More...

- struct sbc_trans_evnt_t

    Transceiver event capture enable register structure. Can bus silence, Can failure and Can wake-up settings. More...

- struct sbc_frame_t

    Frame control register structure. The wake-up frame format, standard (11-bit) or extended (29-bit) identifier, is selected via bit IDE in the Frame control register. More...

- struct sbc_can_conf_t

    CAN configuration group structure. This structure configure CAN peripheral behavior. More...

- struct sbc_wake_t

    WAKE pin event capture enable register structure. Local wake-up is enabled via bits WPRE and WPFE in the WAKE pin event capture enable register. A wake-up event is triggered by a LOW-to-HIGH (if WPRE = 1) and/or a HIGH-to-LOW (if WPFE = 1) transition on the WAKE pin. This arrangement allows for maximum flexibility when designing a local wake-up circuit. In applications that do not use the local wake-up facility, local wake-up should be disabled and the WAKE pin connected to GND. More...

- struct sbc_regulator_ctr_t

    Regulator control register group. This structure is group of regulator settings. More...

- struct sbc_int_config_t

    Init configuration structure. This structure is used for initialization of sbc. More...

- struct sbc_factories_conf_t

    Factory configuration structure. It contains Start-up control register and SBC configuration control register. This is non-volatile memory with limited write access. The MTPNV cells can be reprogrammed a maximum of 200 times (Ncy(W)MTP; Bit NVMPS in the MTPNV status register indicates whether the non-volatile cells can be reprogrammed. This register also contains a write counter, WRCNTS, that is incremented each time the MTPNV cells are reprogrammed (up to a maximum value of 111111; there is no overflow; performing a factory reset also increments the counter). This counter is provided for information purposes only; reprogramming will not be rejected when it reaches its maximum value. Factory preset values are restored if the following conditions apply continuously for at least td(MTPNV) during battery power-up: pin RSTN is held LOW, CANH is pulled up to VBAT, CANL is pulled down to

GND After the factory preset values have been restored, the SBC performs a system reset and enters Forced normal Mode. Since the CAN-bus is clamped dominant, pin RXDC is forced LOW. Pin RXD is forced HIGH during the factory preset restore process (td(MTPNV)). A falling edge on RXD caused by bit PO being set after power-on indicates that the factory preset process has been completed. Note that the write counter, WRCNTS, in the MTPNV status register is incremented every time the factory presets are restored. More...

- struct sbc_main_status_t

  Main status register structure. The Main status register can be accessed to monitor the status of the overtemperature warning flag and to determine whether the UJA1169 has entered Normal mode after initial power-up. It also indicates the source of the most recent reset event. More...

- struct sbc_wtdog_status_t

  Watchdog status register structure. Information on the status of the watchdog is available from the Watchdog status register. This register also indicates whether Forced Normal and Software Development modes are active. More...

- struct sbc_supply_status_t

  Supply voltage status register structure. V2/VEXT and V1 undervoltage and overvoltage status. More...

- struct sbc_trans_stat_t

  Transceiver status register structure. There are stored CAN transceiver statuses. More...

- struct sbc_gl_evnt_stat_t

  Global event status register. The microcontroller can monitor events via the event status registers. An extra status register, the Global event status register, is provided to help speed up software polling routines. By polling the Global event status register, the microcontroller can quickly determine the type of event captured (system, supply, transceiver or WAKE pin) and then query the relevant event status register. More...

- struct sbc_sys_evnt_stat_t

  System event status register. Wake-up and interrupt event diagnosis in the UJA1169 is intended to provide the microcontroller with information on the status of a range of features and functions. This information is stored in the event status registers and is signaled on pin RXD, if enabled. More...

- struct sbc_sup_evnt_stat_t

  Supply event status register. More...

- struct sbc_trans_evnt_stat_t

  Transceiver event status register. More...

- struct sbc_wake_evnt_stat_t

  WAKE pin event status register. More...

- struct sbc_evn_capt_t

  Event capture registers structure. This structure contains Global event status, System event status, Supply event status, Transceiver event status, WAKE pin event status. More...

- struct sbc_mtpnv_stat_t

  MTPNV status register. The MTPNV cells can be reprogrammed a maximum of 200 times (Ncy(W)MTP). Bit N$\leftarrow$ VMPS in the MTPNV status register indicates whether the non-volatile cells can be reprogrammed. This register also contains a write counter, WRCNTS, that is incremented each time the MTPNV cells are reprogrammed (up to a maximum value of 111111; there is no overflow; performing a factory reset also increments the counter). This counter is provided for information purposes only; reprogramming will not be rejected when it reaches its maximum value. More...

- struct sbc_status_group_t

  Status group structure. All statuses of SBC are stored in this structure. More...

Macros

- #define SBC_UJA_TIMEOUT 1000U
- #define SBC_UJA_COUNT_ID_REG 4U
- #define SBC_UJA_COUNT_MASK 4U
- #define SBC_UJA_COUNT_DMASK 8U

Typedefs

- typedef uint8_t sbc_fail_safe_rcc_t

  Fail-safe control register, reset counter control (0x02). incremented every time the SBC enters Reset mode while FNMC = 0; RCC overflows from 11 to 00; default at power-on is 00.

- typedef uint8_t sbc_identifier_t

  ID registers, identifier format (0x27 to 0x2A). A valid WUF identifier is defined and stored in the ID registers. An ID mask can be defined to allow a group of identifiers to be recognized as valid by an individual node.

- typedef uint8_t sbc_identif_mask_t

  ID mask registers (0x2B to 0x2E). The identifier mask is defined in the ID mask registers, where a 1 means dont care.

- typedef uint8_t sbc_frame_ctr_dlc_t

  Frame control register, number of data bytes expected in a CAN frame (0x2F).

- typedef uint8_t sbc_data_mask_t

  Data mask registers. The data field indicates the nodes to be woken up. Within the data field, groups of nodes can be predefined and associated with bits in a data mask. By comparing the incoming data field with the data mask, multiple groups of nodes can be woken up simultaneously with a single wake-up message.

- typedef uint8_t sbc_mtpnv_stat_wrcnts_t

  MTPNV status register, write counter status (0x70). 6-bits - contains the number of times the MTPNV cells were reprogrammed.

Enumerations

- enum sbc_register_t {
  SBC_UJA_WTDOG_CTR = 0x00U, SBC_UJA_MODE = 0x01U, SBC_UJA_FAIL_SAFE = 0x02U, SBC_U↩
  JA_MAIN = 0x03U,
  SBC_UJA_SYSTEM_EVNT = 0x04U, SBC_UJA_WTDOG_STAT = 0x05U, SBC_UJA_MEMORY_0 =
  0x06U, SBC_UJA_MEMORY_1 = 0x07U,
  SBC_UJA_MEMORY_2 = 0x08U, SBC_UJA_MEMORY_3 = 0x09U, SBC_UJA_LOCK = 0x0AU, SBC_UJ↩
  A_REGULATOR = 0x10U,
  SBC_UJA_SUPPLY_STAT = 0x1BU, SBC_UJA_SUPPLY_EVNT = 0x1CU, SBC_UJA_CAN = 0x20U, SB↩
  C_UJA_TRANS_STAT = 0x22U,
  SBC_UJA_TRANS_EVNT = 0x23U, SBC_UJA_DAT_RATE = 0x26U, SBC_UJA_IDENTIF_0 = 0x27U, S↩
  BC_UJA_IDENTIF_1 = 0x28U,
  SBC_UJA_IDENTIF_2 = 0x29U, SBC_UJA_IDENTIF_3 = 0x2AU, SBC_UJA_MASK_0 = 0x2BU, SBC_UJ↩
  A_MASK_1 = 0x2CU,
  SBC_UJA_MASK_2 = 0x2DU, SBC_UJA_MASK_3 = 0x2EU, SBC_UJA_FRAME_CTR = 0x2FU, SBC_U↩
  JA_DAT_MASK_0 = 0x68U,
  SBC_UJA_DAT_MASK_1 = 0x69U, SBC_UJA_DAT_MASK_2 = 0x6AU, SBC_UJA_DAT_MASK_3 = 0x6↩
  BU, SBC_UJA_DAT_MASK_4 = 0x6CU,
  SBC_UJA_DAT_MASK_5 = 0x6DU, SBC_UJA_DAT_MASK_6 = 0x6EU, SBC_UJA_DAT_MASK_7 = 0x6↩
  FU, SBC_UJA_WAKE_STAT = 0x4BU,
  SBC_UJA_WAKE_EN = 0x4CU, SBC_UJA_GL_EVNT_STAT = 0x60U, SBC_UJA_SYS_EVNT_STAT =
  0x61U, SBC_UJA_SUP_EVNT_STAT = 0x62U,
  SBC_UJA_TRANS_EVNT_STAT = 0x63U, SBC_UJA_WAKE_EVNT_STAT = 0x64U, SBC_UJA_MTPNV↩
  _STAT = 0x70U, SBC_UJA_START_UP = 0x73U,
  SBC_UJA_SBC = 0x74U, SBC_UJA_MTPNV_CRC = 0x75U, SBC_UJA_IDENTIF = 0x7EU }

  Register map.

- enum sbc_wtdog_ctr_wmc_t { SBC_UJA_WTDOG_CTR_WMC_AUTO = SBC_UJA_WTDOG_CTR_WM↩
  C_F(1U), SBC_UJA_WTDOG_CTR_WMC_TIME = SBC_UJA_WTDOG_CTR_WMC_F(2U), SBC_UJA_↩
  WTDOG_CTR_WMC_WIND = SBC_UJA_WTDOG_CTR_WMC_F(4U) }

  Watchdog control register, watchdog mode control (0x00). The UJA1169 contains a watchdog that supports three operating modes: Window, Timeout and Autonomous. In Window mode (available only in SBC Normal mode), a watchdog trigger event within a defined watchdog window triggers and resets the watchdog timer. In Timeout mode, the watchdog runs continuously and can be triggered and reset at any time within the watchdog period by a watchdog trigger. Watchdog time-out mode can also be used for cyclic wake-up of the microcontroller. In Autonomous mode, the watchdog can be off or autonomously in Timeout mode, depending on the selected SBC mode. The watchdog mode

is selected via bits WMC in the Watchdog control register. The SBC must be in Standby mode when the watchdog mode is changed.

- enum sbc_wtdog_ctr_nwp_t {
  SBC_UJA_WTDOG_CTR_NWP_8 = 0x08U, SBC_UJA_WTDOG_CTR_NWP_16 = 0x01U, SBC_UJA_W↩
  TDOG_CTR_NWP_32 = 0x02U, SBC_UJA_WTDOG_CTR_NWP_64 = 0x0BU,
  SBC_UJA_WTDOG_CTR_NWP_128 = 0x04U, SBC_UJA_WTDOG_CTR_NWP_256 = 0x0DU, SBC_UJA↩
  _WTDOG_CTR_NWP_1024 = 0x0EU, SBC_UJA_WTDOG_CTR_NWP_4096 = 0x07U }

  Watchdog control register, nominal watchdog period (0x00). Eight watchdog periods are supported, from 8 ms to 4096 ms. The watchdog period is programmed via bits NWP. The selected period is valid for both Window and Timeout modes. The default watchdog period is 128 ms. A watchdog trigger event resets the watchdog timer. A watchdog trigger event is any valid write access to the Watchdog control register. If the watchdog mode or the watchdog period have changed as a result of the write access, the new values are immediately valid.

- enum sbc_mode_mc_t { SBC_UJA_MODE_MC_SLEEP = 0x01U, SBC_UJA_MODE_MC_STANDBY = 0x04U, SBC_UJA_MODE_MC_NORMAL = 0x07U }

  Mode control register, mode control (0x01)

- enum sbc_fail_safe_lhc_t { SBC_UJA_FAIL_SAFE_LHC_FLOAT = SBC_UJA_FAIL_SAFE_LHC_F(0U), S↩
  BC_UJA_FAIL_SAFE_LHC_LOW = SBC_UJA_FAIL_SAFE_LHC_F(1U) }

  Fail-safe control register, LIMP home control (0x02). The dedicated LIMP pin can be used to enable so called limp home hardware in the event of a serious ECU failure. Detectable failure conditions include SBC overtemperature events, loss of watchdog service, short-circuits on pins RSTN or V1 and user-initiated or external reset events. The LIMP pin is a battery-robust, active-LOW, open-drain output. The LIMP pin can also be forced LOW by setting bit LHC in the Fail-safe control register.

- enum sbc_main_otws_t { SBC_UJA_MAIN_OTWS_BELOW = SBC_UJA_MAIN_OTWS_F(0U), SBC_UJ↩
  A_MAIN_OTWS_ABOVE = SBC_UJA_MAIN_OTWS_F(1U) }

  Main status register, Overtemperature warning status (0x03).

- enum sbc_main_nms_t { SBC_UJA_MAIN_NMS_NORMAL = SBC_UJA_MAIN_NMS_F(0U), SBC_UJA_↩
  MAIN_NMS_PWR_UP = SBC_UJA_MAIN_NMS_F(1U) }

  Main status register, normal mode status (0x03).

- enum sbc_main_rss_t {
  SBC_UJA_MAIN_RSS_OFF_MODE = 0x00U, SBC_UJA_MAIN_RSS_CAN_WAKEUP = 0x01U, SBC_UJ↩
  A_MAIN_RSS_SLP_WAKEUP = 0x04U, SBC_UJA_MAIN_RSS_OVF_SLP = 0x0CU,
  SBC_UJA_MAIN_RSS_DIAG_WAKEUP = 0x0DU, SBC_UJA_MAIN_RSS_WATCH_TRIG = 0x0EU, SBC↩
  _UJA_MAIN_RSS_WATCH_OVF = 0x0FU, SBC_UJA_MAIN_RSS_ILLEG_WATCH = 0x10U,
  SBC_UJA_MAIN_RSS_RSTN_PULDW = 0x11U, SBC_UJA_MAIN_RSS_LFT_OVERTM = 0x12U, SBC_↩
  UJA_MAIN_RSS_V1_UNDERV = 0x13U, SBC_UJA_MAIN_RSS_ILLEG_SLP = 0x14U,
  SBC_UJA_MAIN_RSS_WAKE_SLP = 0x16U }

  Main status register, Reset source status (0x03).

- enum sbc_sys_evnt_otwe_t { SBC_UJA_SYS_EVNT_OTWE_DIS = SBC_UJA_SYS_EVNT_OTWE_F(0U), SBC_UJA_SYS_EVNT_OTWE_EN = SBC_UJA_SYS_EVNT_OTWE_F(1U) }

  System event capture enable, overtemperature warning enable (0x04).

- enum sbc_sys_evnt_spife_t { SBC_UJA_SYS_EVNT_SPIFE_DIS = SBC_UJA_SYS_EVNT_SPIFE_F(0U), SBC_UJA_SYS_EVNT_SPIFE_EN = SBC_UJA_SYS_EVNT_SPIFE_F(1U) }

  System event capture enable, SPI failure enable (0x04).

- enum sbc_wtdog_stat_fnms_t { SBC_UJA_WTDOG_STAT_FNMS_N_NORMAL = SBC_UJA_WTDOG_↩
  STAT_FNMS_F(0U), SBC_UJA_WTDOG_STAT_FNMS_NORMAL = SBC_UJA_WTDOG_STAT_FNMS↩
  _F(1U) }

  Watchdog status register, forced Normal mode status (0x05).

- enum sbc_wtdog_stat_sdms_t { SBC_UJA_WTDOG_STAT_SDMS_N_NORMAL = SBC_UJA_WTDOG_↩
  STAT_SDMS_F(0U), SBC_UJA_WTDOG_STAT_SDMS_NORMAL = SBC_UJA_WTDOG_STAT_SDMS↩
  _F(1U) }

  Watchdog status register, Software Development mode status (0x05).

- enum sbc_wtdog_stat_wds_t { SBC_UJA_WTDOG_STAT_WDS_OFF = SBC_UJA_WTDOG_STAT_WD↩
  S_F(0U), SBC_UJA_WTDOG_STAT_WDS_FIH = SBC_UJA_WTDOG_STAT_WDS_F(1U), SBC_UJA_↩
  WTDOG_STAT_WDS_SEH = SBC_UJA_WTDOG_STAT_WDS_F(2U) }

  Watchdog status register, watchdog status (0x05).

- enum sbc_lock_t {
  LK0C = SBC_UJA_LOCK_LK0C_MASK, LK1C = SBC_UJA_LOCK_LK1C_MASK, LK2C = SBC_UJA_LO↩
  CK_LK2C_MASK, LK3C = SBC_UJA_LOCK_LK3C_MASK,
  LK4C = SBC_UJA_LOCK_LK4C_MASK, LK5C = SBC_UJA_LOCK_LK5C_MASK, LK6C = SBC_UJA_LO↩
  CK_LK6C_MASK, LKAC = SBC_UJA_LOCK_LKNC_MASK }

  Lock control(0x0A). Sections of the register address area can be write-protected to protect against unintended mod-
  ifications. This facility only protects locked bits from being modified via the SPI and will not prevent the UJA1169
  updating status registers etc.

- enum sbc_regulator_pdc_t { SBC_UJA_REGULATOR_PDC_HV = SBC_UJA_REGULATOR_PDC_F(0U),
  SBC_UJA_REGULATOR_PDC_LV = SBC_UJA_REGULATOR_PDC_F(1U) }

  Regulator control register, power distribution control (0x10).

- enum sbc_regulator_v2c_t { SBC_UJA_REGULATOR_V2C_OFF = SBC_UJA_REGULATOR_V2C_F(0U),
  SBC_UJA_REGULATOR_V2C_N = SBC_UJA_REGULATOR_V2C_F(1U), SBC_UJA_REGULATOR_V2↩
  C_N_S_R = SBC_UJA_REGULATOR_V2C_F(2U), SBC_UJA_REGULATOR_V2C_N_S_S_R = SBC_UJ↩
  A_REGULATOR_V2C_F(3U) }

  Regulator control register, V2/VEXT configuration (0x10).

- enum sbc_regulator_v1rtc_t { SBC_UJA_REGULATOR_V1RTC_90 = SBC_UJA_REGULATOR_V1RTC↩
  _F(0U), SBC_UJA_REGULATOR_V1RTC_80 = SBC_UJA_REGULATOR_V1RTC_F(1U), SBC_UJA_RE↩
  GULATOR_V1RTC_70 = SBC_UJA_REGULATOR_V1RTC_F(2U), SBC_UJA_REGULATOR_V1RTC_60 =
  SBC_UJA_REGULATOR_V1RTC_F(3U) }

  Regulator control register, set V1 reset threshold (0x10).

- enum sbc_supply_stat_v2s_t { SBC_UJA_SUPPLY_STAT_V2S_VOK = SBC_UJA_SUPPLY_STAT_V2S↩
  _F(0U), SBC_UJA_SUPPLY_STAT_V2S_VBE = SBC_UJA_SUPPLY_STAT_V2S_F(1U), SBC_UJA_SU↩
  PPLY_STAT_V2S_VAB = SBC_UJA_SUPPLY_STAT_V2S_F(2U), SBC_UJA_SUPPLY_STAT_V2S_DIS =
  SBC_UJA_SUPPLY_STAT_V2S_F(3U) }

  Supply voltage status register, V2/VEXT status (0x1B).

- enum sbc_supply_stat_v1s_t { SBC_UJA_SUPPLY_STAT_V1S_VAB = SBC_UJA_SUPPLY_STAT_V1S↩
  _F(0U), SBC_UJA_SUPPLY_STAT_V1S_VBE = SBC_UJA_SUPPLY_STAT_V1S_F(1U) }

  Supply voltage status register, V1 status (0x1B).

- enum sbc_supply_evnt_v2oe_t { SBC_UJA_SUPPLY_EVNT_V2OE_DIS = SBC_UJA_SUPPLY_EVNT_↩
  V2OE_F(0U), SBC_UJA_SUPPLY_EVNT_V2OE_EN = SBC_UJA_SUPPLY_EVNT_V2OE_F(1U) }

  Supply event capture enable register, V2/VEXT overvoltage enable (0x1C).

- enum sbc_supply_evnt_v2ue_t { SBC_UJA_SUPPLY_EVNT_V2UE_DIS = SBC_UJA_SUPPLY_EVNT_↩
  V2UE_F(0U), SBC_UJA_SUPPLY_EVNT_V2UE_EN = SBC_UJA_SUPPLY_EVNT_V2UE_F(1U) }

  Supply event capture enable register, V2/VEXT undervoltage enable (0x1C).

- enum sbc_supply_evnt_v1ue_t { SBC_UJA_SUPPLY_EVNT_V1UE_DIS = SBC_UJA_SUPPLY_EVNT_↩
  V1UE_F(0U), SBC_UJA_SUPPLY_EVNT_V1UE_EN = SBC_UJA_SUPPLY_EVNT_V1UE_F(1U) }

  Supply event capture enable register, V1 undervoltage enable (0x1C).

- enum sbc_can_cfdc_t { SBC_UJA_CAN_CFDC_DIS = SBC_UJA_CAN_CFDC_F(0U), SBC_UJA_CAN_↩
  CFDC_EN = SBC_UJA_CAN_CFDC_F(1U) }

  CAN control register, CAN FD control (0x20).

- enum sbc_can_pncok_t { SBC_UJA_CAN_PNCOK_DIS = SBC_UJA_CAN_PNCOK_F(0U), SBC_UJA_C↩
  AN_PNCOK_EN = SBC_UJA_CAN_PNCOK_F(1U) }

  CAN control register, CAN partial networking configuration OK (0x20).

- enum sbc_can_cpnc_t { SBC_UJA_CAN_CPNC_DIS = SBC_UJA_CAN_CPNC_F(0U), SBC_UJA_CAN_↩
  CPNC_EN = SBC_UJA_CAN_CPNC_F(1U) }

  CAN control register, CAN partial networking control (0x20).

- enum sbc_can_cmc_t { SBC_UJA_CAN_CMC_OFMODE = SBC_UJA_CAN_CMC_F(0U), SBC_UJA_CA↩
  N_CMC_ACMODE_DA = SBC_UJA_CAN_CMC_F(1U), SBC_UJA_CAN_CMC_ACMODE_DD = SBC_U↩
  JA_CAN_CMC_F(2U), SBC_UJA_CAN_CMC_LISTEN = SBC_UJA_CAN_CMC_F(3U) }

  CAN control register, CAN mode control (0x20).

- enum sbc_trans_stat_cts_t { SBC_UJA_TRANS_STAT_CTS_INACT = SBC_UJA_TRANS_STAT_CTS_↩
  F(0U), SBC_UJA_TRANS_STAT_CTS_ACT = SBC_UJA_TRANS_STAT_CTS_F(1U) }

  Transceiver status register, CAN transceiver status (0x22).

- enum sbc_trans_stat_cpnerr_t { SBC_UJA_TRANS_STAT_CPNERR_NO_DET = SBC_UJA_TRANS_ST↩
AT_CPNERR_F(0U), SBC_UJA_TRANS_STAT_CPNERR_DET = SBC_UJA_TRANS_STAT_CPNERR_↩
F(1U) }

    Transceiver status register, CAN partial networking error (0x22).
- enum sbc_trans_stat_cpns_t { SBC_UJA_TRANS_STAT_CPNS_ERR = SBC_UJA_TRANS_STAT_CPN↩
S_F(0U), SBC_UJA_TRANS_STAT_CPNS_OK = SBC_UJA_TRANS_STAT_CPNS_F(1U) }

    Transceiver status register, CAN partial networking status (0x22).
- enum sbc_trans_stat_coscs_t { SBC_UJA_TRANS_STAT_COSCS_NRUN = SBC_UJA_TRANS_STAT_↩
COSCS_F(0U), SBC_UJA_TRANS_STAT_COSCS_RUN = SBC_UJA_TRANS_STAT_COSCS_F(1U) }

    Transceiver status register, CAN oscillator status (0x22).
- enum sbc_trans_stat_cbss_t { SBC_UJA_TRANS_STAT_CBSS_ACT = SBC_UJA_TRANS_STAT_CBSS↩
_F(0U), SBC_UJA_TRANS_STAT_CBSS_INACT = SBC_UJA_TRANS_STAT_CBSS_F(1U) }

    Transceiver status register, CAN-bus silence status (0x22).
- enum sbc_trans_stat_vcs_t { SBC_UJA_TRANS_STAT_VCS_AB = SBC_UJA_TRANS_STAT_VCS_F(0U),
SBC_UJA_TRANS_STAT_VCS_BE = SBC_UJA_TRANS_STAT_VCS_F(1U) }

    Transceiver status register, VCAN status (0x22).
- enum sbc_trans_stat_cfs_t { SBC_UJA_TRANS_STAT_CFS_NO_TXD = SBC_UJA_TRANS_STAT_CFS↩
_F(0U), SBC_UJA_TRANS_STAT_CFS_TXD = SBC_UJA_TRANS_STAT_CFS_F(1U) }

    Transceiver status register, CAN failure status (0x22).
- enum sbc_trans_evnt_cbse_t { SBC_UJA_TRANS_EVNT_CBSE_DIS = SBC_UJA_TRANS_EVNT_CBS↩
E_F(0U), SBC_UJA_TRANS_EVNT_CBSE_EN = SBC_UJA_TRANS_EVNT_CBSE_F(1U) }

    Transceiver event capture enable register, CAN-bus silence enable (0x23).
- enum sbc_trans_evnt_cfe_t { SBC_UJA_TRANS_EVNT_CFE_DIS = SBC_UJA_TRANS_EVNT_CFE_F(0↩
U), SBC_UJA_TRANS_EVNT_CFE_EN = SBC_UJA_TRANS_EVNT_CFE_F(1U) }

    Transceiver event capture enable register, CAN failure enable (0x23).
- enum sbc_trans_evnt_cwe_t { SBC_UJA_TRANS_EVNT_CWE_DIS = SBC_UJA_TRANS_EVNT_CWE_↩
F(0U), SBC_UJA_TRANS_EVNT_CWE_EN = SBC_UJA_TRANS_EVNT_CWE_F(1U) }

    Transceiver event capture enable register, CAN wake-up enable (0x23).
- enum sbc_dat_rate_t {
SBC_UJA_DAT_RATE_CDR_50KB = SBC_UJA_DAT_RATE_CDR_F(0U), SBC_UJA_DAT_RATE_CDR↩
_100KB = SBC_UJA_DAT_RATE_CDR_F(1U), SBC_UJA_DAT_RATE_CDR_125KB = SBC_UJA_DAT_↩
RATE_CDR_F(2U), SBC_UJA_DAT_RATE_CDR_250KB = SBC_UJA_DAT_RATE_CDR_F(3U),
SBC_UJA_DAT_RATE_CDR_500KB = SBC_UJA_DAT_RATE_CDR_F(5U), SBC_UJA_DAT_RATE_CD↩
R_1000KB = SBC_UJA_DAT_RATE_CDR_F(7U) }

    Data rate register, CAN data rate selection (0x26). CAN partial networking configuration registers. Dedicated registers are provided for configuring CAN partial networking.
- enum sbc_frame_ctr_ide_t { SBC_UJA_FRAME_CTR_IDE_11B = SBC_UJA_FRAME_CTR_IDE_F(0U),
SBC_UJA_FRAME_CTR_IDE_29B = SBC_UJA_FRAME_CTR_IDE_F(1U) }

    Frame control register, identifier format (0x2F). The wake-up frame format, standard (11-bit) or extended (29-bit) identifier, is selected via bit IDE in the Frame control register.
- enum sbc_frame_ctr_pndm_t { SBC_UJA_FRAME_CTR_PNDM_DCARE = SBC_UJA_FRAME_CTR_PN↩
DM_F(0U), SBC_UJA_FRAME_CTR_PNDM_EVAL = SBC_UJA_FRAME_CTR_PNDM_F(1U) }

    Frame control register, partial networking data mask (0x2F).
- enum sbc_wake_stat_wpvs_t { SBC_UJA_WAKE_STAT_WPVS_BE = SBC_UJA_WAKE_STAT_WPVS_↩
F(0U), SBC_UJA_WAKE_STAT_WPVS_AB = SBC_UJA_WAKE_STAT_WPVS_F(1U) }

    WAKE pin status register, WAKE pin status (0x4B).
- enum sbc_wake_en_wpre_t { SBC_UJA_WAKE_EN_WPRE_DIS = SBC_UJA_WAKE_EN_WPRE_F(0U),
SBC_UJA_WAKE_EN_WPRE_EN = SBC_UJA_WAKE_EN_WPRE_F(1U) }

    WAKE pin event capture enable register, WAKE pin rising-edge enable (0x4C).
- enum sbc_wake_en_wpfe_t { SBC_UJA_WAKE_EN_WPFE_DIS = SBC_UJA_WAKE_EN_WPFE_F(0U),
SBC_UJA_WAKE_EN_WPFE_EN = SBC_UJA_WAKE_EN_WPFE_F(1U) }

    WAKE pin event capture enable register, WAKE pin falling-edge enable (0x4C).
- enum sbc_gl_evnt_stat_wpe_t { SBC_UJA_GL_EVNT_STAT_WPE_NO = SBC_UJA_GL_EVNT_STAT_↩
WPE_F(0U), SBC_UJA_GL_EVNT_STAT_WPE = SBC_UJA_GL_EVNT_STAT_WPE_F(1U) }

WAKE pin event status register, WAKE pin falling edge (0x64).

- enum sbc_mtpnv_stat_eccs_t { SBC_UJA_MTPNV_STAT_ECCS_NO = SBC_UJA_MTPNV_STAT_ECC↩
S_F(0U), SBC_UJA_MTPNV_STAT_ECCS = SBC_UJA_MTPNV_STAT_ECCS_F(1U) }

    MTPNV status register, error correction code status (0x70).

- enum sbc_mtpnv_stat_nvmps_t { SBC_UJA_MTPNV_STAT_NVMPS_NO = SBC_UJA_MTPNV_STAT_N↩
VMPS_F(0U), SBC_UJA_MTPNV_STAT_NVMPS = SBC_UJA_MTPNV_STAT_NVMPS_F(1U) }

    MTPNV status register, non-volatile memory programming status (0x70).

- enum sbc_start_up_rlc_t { SBC_UJA_START_UP_RLC_20_25p0 = SBC_UJA_START_UP_RLC_F(0U),
SBC_UJA_START_UP_RLC_10_12p5 = SBC_UJA_START_UP_RLC_F(1U), SBC_UJA_START_UP_R↩
LC_03p6_05 = SBC_UJA_START_UP_RLC_F(2U), SBC_UJA_START_UP_RLC_01_01p5 = SBC_UJA_↩
START_UP_RLC_F(3U) }

    Start-up control register, RSTN output reset pulse width macros (0x73).

- enum sbc_start_up_v2suc_t { SBC_UJA_START_UP_V2SUC_00 = SBC_UJA_START_UP_V2SUC_F(0U),
SBC_UJA_START_UP_V2SUC_11 = SBC_UJA_START_UP_V2SUC_F(1U) }

    Start-up control register, V2/VEXT start-up control (0x73).

- enum sbc_sbc_v1rtsuc_t { SBC_UJA_SBC_V1RTSUC_90 = SBC_UJA_SBC_V1RTSUC_F(0U), SBC_U↩
JA_SBC_V1RTSUC_80 = SBC_UJA_SBC_V1RTSUC_F(1U), SBC_UJA_SBC_V1RTSUC_70 = SBC_UJ↩
A_SBC_V1RTSUC_F(2U), SBC_UJA_SBC_V1RTSUC_60 = SBC_UJA_SBC_V1RTSUC_F(3U) }

    SBC configuration control register, V1 undervoltage threshold (defined by bit V1RTC) at start-up (0x74).

- enum sbc_sbc_fnmc_t { SBC_UJA_SBC_FNMC_DIS = SBC_UJA_SBC_FNMC_F(0U), SBC_UJA_SBC_↩
FNMC_EN = SBC_UJA_SBC_FNMC_F(1U) }

    SBC configuration control register, Forced Normal mode control (0x74).

- enum sbc_sbc_sdmc_t { SBC_UJA_SBC_SDMC_DIS = SBC_UJA_SBC_SDMC_F(0U), SBC_UJA_SBC↩
_SDMC_EN = SBC_UJA_SBC_SDMC_F(1U) }

    SBC configuration control register, Software Development mode control (0x74).

- enum sbc_sbc_slpc_t { SBC_UJA_SBC_SLPC_AC = SBC_UJA_SBC_SLPC_F(0U), SBC_UJA_SBC_SL↩
PC_IG = SBC_UJA_SBC_SLPC_F(1U) }

    SBC configuration control register, Sleep control (0x74).

16.88.2   Data Structure Documentation

16.88.2.1   struct sbc_wtdog_ctr_t

Watchdog control register structure. Watchdog configuration structure.

Implements : sbc_wtdog_ctr_t_Class

Definition at line 1087 of file sbc_uja1169_driver.h.

Data Fields

- sbc_wtdog_ctr_wmc_t modeControl
- sbc_wtdog_ctr_nwp_t nominalPeriod

Field Documentation

16.88.2.1.1   sbc_wtdog_ctr_wmc_t modeControl

Watchdog mode control.

Definition at line 1088 of file sbc_uja1169_driver.h.

16.88.2.1.2   sbc_wtdog_ctr_nwp_t nominalPeriod

Nominal watchdog period.

Definition at line 1089 of file sbc_uja1169_driver.h.

16.88.2.2   struct sbc_sbc_t

SBC configuration control register structure. Two operating modes have a major impact on the operation of the watchdog: Forced Normal mode and Software Development mode (Software Development mode is provided for test and development purposes only and is not a dedicated SBC operating mode; the UJA1169 can be in any functional operating mode with Software Development mode enabled). These modes are enabled and disabled via bits FNMC and SDMC respectively in the SBC configuration control register. Note that this register is located in the non-volatile memory area. The watchdog is disabled in Forced Normal mode (FNM). In Software Development mode (SDM), the watchdog can be disabled or activated for test and software debugging purposes.

Implements : sbc_sbc_t_Class

Definition at line 1108 of file sbc_uja1169_driver.h.

Data Fields

- sbc_sbc_v1rtsuc_t v1rtsuc
- sbc_sbc_fnmc_t fnmc
- sbc_sbc_sdmc_t sdmc
- sbc_sbc_slpc_t slpc

Field Documentation

16.88.2.2.1   sbc_sbc_fnmc_t fnmc

Forced Normal mode control.

Definition at line 1111 of file sbc_uja1169_driver.h.

16.88.2.2.2   sbc_sbc_sdmc_t sdmc

Software Development mode control.

Definition at line 1112 of file sbc_uja1169_driver.h.

16.88.2.2.3   sbc_sbc_slpc_t slpc

Sleep control.

Definition at line 1114 of file sbc_uja1169_driver.h.

16.88.2.2.4   sbc_sbc_v1rtsuc_t v1rtsuc

V1 undervoltage threshold (defined by bit V1RTC) at start-up (0x74).

Definition at line 1109 of file sbc_uja1169_driver.h.

16.88.2.3   struct sbc_start_up_t

Start-up control register structure. This structure contains settings of RSTN output reset pulse width and V2/VEXT start-up control.

Implements : sbc_start_up_t_Class

Definition at line 1124 of file sbc_uja1169_driver.h.

Data Fields

- sbc_start_up_rlc_t rlc
- sbc_start_up_v2suc_t v2suc

Field Documentation

16.88.2.3.1 sbc_start_up_rlc_t rlc

RSTN output reset pulse width macros.

Definition at line 1125 of file sbc_uja1169_driver.h.

16.88.2.3.2 sbc_start_up_v2suc_t v2suc

V2/VEXT start-up control.

Definition at line 1127 of file sbc_uja1169_driver.h.

16.88.2.4 struct sbc_regulator_t

Regulator control register structure. This structure set power distribution control, V2/VEXT configuration, set V1 reset threshold.

Implements : sbc_regulator_t_Class

Definition at line 1137 of file sbc_uja1169_driver.h.

Data Fields

- sbc_regulator_pdc_t pdc
- sbc_regulator_v2c_t v2c
- sbc_regulator_v1rtc_t v1rtc

Field Documentation

16.88.2.4.1 sbc_regulator_pdc_t pdc

Power distribution control.

Definition at line 1138 of file sbc_uja1169_driver.h.

16.88.2.4.2 sbc_regulator_v1rtc_t v1rtc

Set V1 reset threshold.

Definition at line 1140 of file sbc_uja1169_driver.h.

16.88.2.4.3 sbc_regulator_v2c_t v2c

V2/VEXT configuration.

Definition at line 1139 of file sbc_uja1169_driver.h.

16.88.2.5 struct sbc_supply_evnt_t

Supply event capture enable register structure. This structure enables or disables detection of V2/VEXT overvoltage, undervoltage and V1 undervoltage enable.

Implements : sbc_supply_evnt_t_Class

Definition at line 1150 of file sbc_uja1169_driver.h.

Data Fields

- sbc_supply_evnt_v2oe_t v2oe
- sbc_supply_evnt_v2ue_t v2ue
- sbc_supply_evnt_v1ue_t v1ue

Field Documentation

16.88.2.5.1   sbc_supply_evnt_v1ue_t v1ue

SV1 undervoltage enable.

Definition at line 1153 of file sbc_uja1169_driver.h.

16.88.2.5.2   sbc_supply_evnt_v2oe_t v2oe

V2/VEXT overvoltage enable.

Definition at line 1151 of file sbc_uja1169_driver.h.

16.88.2.5.3   sbc_supply_evnt_v2ue_t v2ue

V2/VEXT undervoltage enable.

Definition at line 1152 of file sbc_uja1169_driver.h.

16.88.2.6   struct sbc_sys_evnt_t

System event capture enable register structure. This structure enables or disables overtemperature warning, SPI failure enable.

Implements : sbc_sys_evnt_t_Class

Definition at line 1163 of file sbc_uja1169_driver.h.

Data Fields

- sbc_sys_evnt_otwe_t owte
- sbc_sys_evnt_spife_t spife

Field Documentation

16.88.2.6.1   sbc_sys_evnt_otwe_t owte

Overtemperature warning enable.

Definition at line 1164 of file sbc_uja1169_driver.h.

16.88.2.6.2   sbc_sys_evnt_spife_t spife

SPI failure enable.

Definition at line 1165 of file sbc_uja1169_driver.h.

16.88.2.7   struct sbc_can_ctr_t

CAN control register structure. This structure configure CAN peripheral behavior.

Implements : sbc_can_ctr_t_Class

Definition at line 1174 of file sbc_uja1169_driver.h.

Data Fields

- sbc_can_cfdc_t cfdc
- sbc_can_pncok_t pncok
- sbc_can_cpnc_t cpnc
- sbc_can_cmc_t cmc

Field Documentation

16.88.2.7.1   sbc_can_cfdc_t cfdc

CAN FD control.

Definition at line 1175 of file sbc_uja1169_driver.h.

16.88.2.7.2   sbc_can_cmc_t cmc

CAN mode control.

Definition at line 1180 of file sbc_uja1169_driver.h.

16.88.2.7.3   sbc_can_cpnc_t cpnc

CAN partial. networking control.

Definition at line 1178 of file sbc_uja1169_driver.h.

16.88.2.7.4   sbc_can_pncok_t pncok

CAN partial networking. configuration OK.

Definition at line 1176 of file sbc_uja1169_driver.h.

16.88.2.8   struct sbc_trans_evnt_t

Transceiver event capture enable register structure. Can bus silence, Can failure and Can wake-up settings.

Implements : sbc_trans_evnt_t_Class

Definition at line 1189 of file sbc_uja1169_driver.h.

Data Fields

- sbc_trans_evnt_cbse_t cbse
- sbc_trans_evnt_cfe_t cfe
- sbc_trans_evnt_cwe_t cwe

Field Documentation

16.88.2.8.1   sbc_trans_evnt_cbse_t cbse

CAN-bus silence enable.

Definition at line 1190 of file sbc_uja1169_driver.h.

16.88.2.8.2   sbc_trans_evnt_cfe_t cfe

CAN failure enable.

Definition at line 1191 of file sbc_uja1169_driver.h.

16.88.2.8.3   sbc_trans_evnt_cwe_t cwe

CAN wake-up enable.

Definition at line 1192 of file sbc_uja1169_driver.h.

16.88.2.9   struct sbc_frame_t

Frame control register structure. The wake-up frame format, standard (11-bit) or extended (29-bit) identifier, is selected via bit IDE in the Frame control register.

Implements : sbc_frame_t_Class

Definition at line 1202 of file sbc_uja1169_driver.h.

Data Fields

- sbc_frame_ctr_ide_t ide

- sbc_frame_ctr_pndm_t pndm
- sbc_frame_ctr_dlc_t dlc

Field Documentation

16.88.2.9.1 sbc_frame_ctr_dlc_t dlc

Number of data bytes expected.

Definition at line 1205 of file sbc_uja1169_driver.h.

16.88.2.9.2 sbc_frame_ctr_ide_t ide

Identifier format.

Definition at line 1203 of file sbc_uja1169_driver.h.

16.88.2.9.3 sbc_frame_ctr_pndm_t pndm

Partial networking data mask.

Definition at line 1204 of file sbc_uja1169_driver.h.

16.88.2.10 struct sbc_can_conf_t

CAN configuration group structure. This structure configure CAN peripheral behavior.

Implements : sbc_can_conf_t_Class

Definition at line 1214 of file sbc_uja1169_driver.h.

Data Fields

- sbc_can_ctr_t canConf
- sbc_trans_evnt_t canTransEvnt
- sbc_dat_rate_t datRate
- sbc_identifier_t identif [SBC_UJA_COUNT_ID_REG]
- sbc_identif_mask_t mask [SBC_UJA_COUNT_MASK]
- sbc_frame_t frame
- sbc_data_mask_t dataMask [SBC_UJA_COUNT_DMASK]

Field Documentation

16.88.2.10.1 sbc_can_ctr_t canConf

CAN control register.

Definition at line 1215 of file sbc_uja1169_driver.h.

16.88.2.10.2 sbc_trans_evnt_t canTransEvnt

Transceiver event capture enable register.

Definition at line 1216 of file sbc_uja1169_driver.h.

16.88.2.10.3 sbc_data_mask_t dataMask[SBC_UJA_COUNT_DMASK]

Data mask 0 - 7 configuration.

Definition at line 1222 of file sbc_uja1169_driver.h.

16.88.2.10.4 sbc_dat_rate_t datRate

CAN data rate selection.

Definition at line 1218 of file sbc_uja1169_driver.h.

16.88.2.10.5   sbc_frame_t frame

Frame control register.

Definition at line 1221 of file sbc_uja1169_driver.h.

16.88.2.10.6   sbc_identifier_t identif[SBC_UJA_COUNT_ID_REG]

ID registers.

Definition at line 1219 of file sbc_uja1169_driver.h.

16.88.2.10.7   sbc_identif_mask_t mask[SBC_UJA_COUNT_MASK]

ID mask registers.

Definition at line 1220 of file sbc_uja1169_driver.h.

16.88.2.11   struct sbc_wake_t

WAKE pin event capture enable register structure. Local wake-up is enabled via bits WPRE and WPFE in the WAKE pin event capture enable register. A wake-up event is triggered by a LOW-to-HIGH (ifWPRE = 1) and/or a HIGH-to-LOW (if WPFE = 1) transition on the WAKE pin. This arrangement allows for maximum flexibility when designing a local wake-up circuit. In applications that do not use the local wake-up facility, local wake-up should be disabled and the WAKE pin connected to GND.

Implements : sbc_wake_t_Class

Definition at line 1237 of file sbc_uja1169_driver.h.

Data Fields

- sbc_wake_en_wpre_t wpre
- sbc_wake_en_wpfe_t wpfe

Field Documentation

16.88.2.11.1   sbc_wake_en_wpfe_t wpfe

WAKE pin falling-edge enable.

Definition at line 1239 of file sbc_uja1169_driver.h.

16.88.2.11.2   sbc_wake_en_wpre_t wpre

WAKE pin rising-edge enable.

Definition at line 1238 of file sbc_uja1169_driver.h.

16.88.2.12   struct sbc_regulator_ctr_t

Regulator control register group. This structure is group of regulator settings.

Implements : sbc_regulator_ctr_t_Class

Definition at line 1248 of file sbc_uja1169_driver.h.

Data Fields

- sbc_regulator_t regulator
- sbc_supply_evnt_t supplyEvnt

Field Documentation

16.88.2.12.1 sbc_regulator_t regulator

Regulator control register.

Definition at line 1249 of file sbc_uja1169_driver.h.

16.88.2.12.2 sbc_supply_evnt_t supplyEvnt

Supply event capture enable register.

Definition at line 1250 of file sbc_uja1169_driver.h.

16.88.2.13 struct sbc_int_config_t

Init configuration structure. This structure is used for initialization of sbc.

Implements : sbc_int_config_t_Class

Definition at line 1260 of file sbc_uja1169_driver.h.

Data Fields

- sbc_regulator_ctr_t regulatorCtr
- sbc_wtdog_ctr_t watchdog
- sbc_mode_mc_t mode
- sbc_fail_safe_lhc_t lhc
- sbc_sys_evnt_t sysEvnt
- sbc_lock_t lockMask
- sbc_can_conf_t can
- sbc_wake_t wakePin

Field Documentation

16.88.2.13.1 sbc_can_conf_t can

CAN configuration group.

Definition at line 1268 of file sbc_uja1169_driver.h.

16.88.2.13.2 sbc_fail_safe_lhc_t lhc

LIMP home control.

Definition at line 1264 of file sbc_uja1169_driver.h.

16.88.2.13.3 sbc_lock_t lockMask

Lock control register.

Definition at line 1267 of file sbc_uja1169_driver.h.

16.88.2.13.4 sbc_mode_mc_t mode

Mode control register.

Definition at line 1263 of file sbc_uja1169_driver.h.

16.88.2.13.5 sbc_regulator_ctr_t regulatorCtr

Regulator control register group.

Definition at line 1261 of file sbc_uja1169_driver.h.

16.88.2.13.6 sbc_sys_evnt_t sysEvnt

System event capture enable registers.

Definition at line 1265 of file sbc_uja1169_driver.h.

16.88.2.13.7   sbc_wake_t wakePin

WAKE pin event capture enable register.

Definition at line 1269 of file sbc_uja1169_driver.h.

16.88.2.13.8   sbc_wtdog_ctr_t watchdog

Watchdog control register.

Definition at line 1262 of file sbc_uja1169_driver.h.

16.88.2.14   struct sbc_factories_conf_t

Factory configuration structure. It contains Start-up control register and SBC configuration control register. This is non-volatile memory with limited write access. The MTPNV cells can be reprogrammed a maximum of 200 times (Ncy(W)MTP; Bit NVMPS in the MTPNV status register indicates whether the non-volatile cells can be reprogrammed. This register also contains a write counter, WRCNTS, that is incremented each time the MTPNV cells are reprogrammed (up to a maximum value of 111111; there is no overflow; performing a factory reset also increments the counter). This counter is provided for information purposes only; reprogramming will not be rejected when it reaches its maximum value. Factory preset values are restored if the following conditions apply continuously for at least td(MTPNV) during battery power-up: pin RSTN is held LOW, CANH is pulled up to VBAT, CANL is pulled down to GND After the factory preset values have been restored, the SBC performs a system reset and enters Forced normal Mode. Since the CAN-bus is clamped dominant, pin RXDC is forced LOW. Pin RXD is forced HIGH during the factory preset restore process (td(MTPNV)). A falling edge on RXD caused by bit PO being set after power-on indicates that the factory preset process has been completed. Note that the write counter, WRCNTS, in the MTPNV status register is incremented every time the factory presets are restored.

Implements : sbc_factories_conf_t_Class

Definition at line 1299 of file sbc_uja1169_driver.h.

Data Fields

- sbc_start_up_t startUp

- sbc_sbc_t control

Field Documentation

16.88.2.14.1   sbc_sbc_t control

SBC configuration control register. Note that this register is located in the non-volatile memory area.

Definition at line 1301 of file sbc_uja1169_driver.h.

16.88.2.14.2   sbc_start_up_t startUp

Start-up control register.

Definition at line 1300 of file sbc_uja1169_driver.h.

16.88.2.15   struct sbc_main_status_t

Main status register structure. The Main status register can be accessed to monitor the status of the overtemperature warning flag and to determine whether the UJA1169 has entered Normal mode after initial power-up. It also indicates the source of the most recent reset event.

Implements : sbc_main_status_t_Class

Definition at line 1315 of file sbc_uja1169_driver.h.

Data Fields

- sbc_main_otws_t otws
- sbc_main_nms_t nms
- sbc_main_rss_t rss

Field Documentation

16.88.2.15.1 sbc_main_nms_t nms

Normal mode status.

Definition at line 1317 of file sbc_uja1169_driver.h.

16.88.2.15.2 sbc_main_otws_t otws

Overtemperature warning status.

Definition at line 1316 of file sbc_uja1169_driver.h.

16.88.2.15.3 sbc_main_rss_t rss

Reset source status.

Definition at line 1318 of file sbc_uja1169_driver.h.

16.88.2.16 struct sbc_wtdog_status_t

Watchdog status register structure. Information on the status of the watchdog is available from the Watchdog status register. This register also indicates whether Forced Normal and Software Development modes are active.

Implements : sbc_wtdog_status_t_Class

Definition at line 1329 of file sbc_uja1169_driver.h.

Data Fields

- sbc_wtdog_stat_fnms_t fnms
- sbc_wtdog_stat_sdms_t sdms
- sbc_wtdog_stat_wds_t wds

Field Documentation

16.88.2.16.1 sbc_wtdog_stat_fnms_t fnms

Forced Normal mode status.

Definition at line 1330 of file sbc_uja1169_driver.h.

16.88.2.16.2 sbc_wtdog_stat_sdms_t sdms

Software Development mode status.

Definition at line 1331 of file sbc_uja1169_driver.h.

16.88.2.16.3 sbc_wtdog_stat_wds_t wds

Watchdog status.

Definition at line 1332 of file sbc_uja1169_driver.h.

16.88.2.17 struct sbc_supply_status_t

Supply voltage status register structure. V2/VEXT and V1 undervoltage and overvoltage status.

Implements : sbc_supply_status_t_Class

Definition at line 1341 of file sbc_uja1169_driver.h.

Data Fields

- sbc_supply_stat_v2s_t v2s
- sbc_supply_stat_v1s_t v1s

Field Documentation

16.88.2.17.1   sbc_supply_stat_v1s_t v1s

V1 status.

Definition at line 1343 of file sbc_uja1169_driver.h.

16.88.2.17.2   sbc_supply_stat_v2s_t v2s

V2/VEXT status.

Definition at line 1342 of file sbc_uja1169_driver.h.

16.88.2.18   struct sbc_trans_stat_t

Transceiver status register structure. There are stored CAN transceiver statuses.

Implements : sbc_trans_stat_t_Class

Definition at line 1352 of file sbc_uja1169_driver.h.

Data Fields

- sbc_trans_stat_cts_t cts
- sbc_trans_stat_cpnerr_t cpnerr
- sbc_trans_stat_cpns_t cpns
- sbc_trans_stat_coscs_t coscs
- sbc_trans_stat_cbss_t cbss
- sbc_trans_stat_vcs_t vcs
- sbc_trans_stat_cfs_t cfs

Field Documentation

16.88.2.18.1   sbc_trans_stat_cbss_t cbss

CAN-bus silence status.

Definition at line 1357 of file sbc_uja1169_driver.h.

16.88.2.18.2   sbc_trans_stat_cfs_t cfs

CAN failure status.

Definition at line 1359 of file sbc_uja1169_driver.h.

16.88.2.18.3   sbc_trans_stat_coscs_t coscs

CAN oscillator status.

Definition at line 1356 of file sbc_uja1169_driver.h.

16.88.2.18.4   sbc_trans_stat_cpnerr_t cpnerr

CAN partial networking error.

Definition at line 1354 of file sbc_uja1169_driver.h.

16.88.2.18.5   sbc_trans_stat_cpns_t cpns

CAN partial networking status.

Definition at line 1355 of file sbc_uja1169_driver.h.

16.88.2.18.6   sbc_trans_stat_cts_t cts

CAN transceiver status.

Definition at line 1353 of file sbc_uja1169_driver.h.

16.88.2.18.7   sbc_trans_stat_vcs_t vcs

VCAN status.

Definition at line 1358 of file sbc_uja1169_driver.h.

16.88.2.19   struct sbc_gl_evnt_stat_t

Global event status register. The microcontroller can monitor events via the event status registers. An extra status register, the Global event status register, is provided to help speed up software polling routines. By polling the Global event status register, the microcontroller can quickly determine the type of event captured (system, supply, transceiver or WAKE pin) and then query the relevant event status register.

Implements : sbc_gl_evnt_stat_t_Class

Definition at line 1373 of file sbc_uja1169_driver.h.

Data Fields

- sbc_gl_evnt_stat_wpe_t wpe
- sbc_gl_evnt_stat_trxe_t trxe
- sbc_gl_evnt_stat_supe_t supe
- sbc_gl_evnt_stat_syse_t syse

Field Documentation

16.88.2.19.1   sbc_gl_evnt_stat_supe_t supe

Supply event.

Definition at line 1376 of file sbc_uja1169_driver.h.

16.88.2.19.2   sbc_gl_evnt_stat_syse_t syse

System event.

Definition at line 1377 of file sbc_uja1169_driver.h.

16.88.2.19.3   sbc_gl_evnt_stat_trxe_t trxe

Transceiver event.

Definition at line 1375 of file sbc_uja1169_driver.h.

16.88.2.19.4   sbc_gl_evnt_stat_wpe_t wpe

WAKE pin event.

Definition at line 1374 of file sbc_uja1169_driver.h.

16.88.2.20   struct sbc_sys_evnt_stat_t

System event status register. Wake-up and interrupt event diagnosis in the UJA1169 is intended to provide the microcontroller with information on the status of a range of features and functions. This information is stored in the

event status registers and is signaled on pin RXD, if enabled.

Implements : sbc_sys_evnt_stat_t_Class

Definition at line 1389 of file sbc_uja1169_driver.h.

Data Fields

- sbc_sys_evnt_stat_po_t po
- sbc_sys_evnt_stat_otw_t otw
- sbc_sys_evnt_stat_spif_t spif
- sbc_sys_evnt_stat_wdf_t wdf

Field Documentation

16.88.2.20.1   sbc_sys_evnt_stat_otw_t otw

Transceiver event, overtemperature warning

Definition at line 1391 of file sbc_uja1169_driver.h.

16.88.2.20.2   sbc_sys_evnt_stat_po_t po

Power-on.

Definition at line 1390 of file sbc_uja1169_driver.h.

16.88.2.20.3   sbc_sys_evnt_stat_spif_t spif

SPI failure.

Definition at line 1393 of file sbc_uja1169_driver.h.

16.88.2.20.4   sbc_sys_evnt_stat_wdf_t wdf

Watchdog failure.

Definition at line 1394 of file sbc_uja1169_driver.h.

16.88.2.21   struct sbc_sup_evnt_stat_t

Supply event status register.

Implements : sbc_sup_evnt_stat_t_Class

Definition at line 1402 of file sbc_uja1169_driver.h.

Data Fields

- sbc_sup_evnt_stat_v2o_t v2o
- sbc_sup_evnt_stat_v2u_t v2u
- sbc_sup_evnt_stat_v1u_t v1u

Field Documentation

16.88.2.21.1   sbc_sup_evnt_stat_v1u_t v1u

V1 undervoltage.

Definition at line 1405 of file sbc_uja1169_driver.h.

16.88.2.21.2   sbc_sup_evnt_stat_v2o_t v2o

V2/VEXT overvoltage.

Definition at line 1403 of file sbc_uja1169_driver.h.

16.88.2.21.3   sbc_sup_evnt_stat_v2u_t v2u

V2/VEXT undervoltage.

Definition at line 1404 of file sbc_uja1169_driver.h.

16.88.2.22   struct sbc_trans_evnt_stat_t

Transceiver event status register.

Implements : sbc_trans_evnt_stat_t_Class

Definition at line 1413 of file sbc_uja1169_driver.h.

Data Fields

- sbc_trans_evnt_stat_pnfde_t pnfde
- sbc_trans_evnt_stat_cbs_t cbs
- sbc_trans_evnt_stat_cf_t cf
- sbc_trans_evnt_stat_cw_t cw

Field Documentation

16.88.2.22.1   sbc_trans_evnt_stat_cbs_t cbs

CAN-bus status.

Definition at line 1416 of file sbc_uja1169_driver.h.

16.88.2.22.2   sbc_trans_evnt_stat_cf_t cf

CAN failure.

Definition at line 1417 of file sbc_uja1169_driver.h.

16.88.2.22.3   sbc_trans_evnt_stat_cw_t cw

CAN wake-up.

Definition at line 1418 of file sbc_uja1169_driver.h.

16.88.2.22.4   sbc_trans_evnt_stat_pnfde_t pnfde

Partial networking frame detection error.

Definition at line 1414 of file sbc_uja1169_driver.h.

16.88.2.23   struct sbc_wake_evnt_stat_t

WAKE pin event status register.

Implements : sbc_wake_evnt_stat_t_Class

Definition at line 1426 of file sbc_uja1169_driver.h.

Data Fields

- sbc_wake_evnt_stat_wpr_t wpr
- sbc_wake_evnt_stat_wpf_t wpf

Field Documentation

16.88.2.23.1   sbc_wake_evnt_stat_wpf_t wpf

WAKE pin falling edge.

Definition at line 1428 of file sbc_uja1169_driver.h.

16.88.2.23.2   sbc_wake_evnt_stat_wpr_t wpr

WAKE pin rising edge.

Definition at line 1427 of file sbc_uja1169_driver.h.

16.88.2.24   struct sbc_evn_capt_t

Event capture registers structure. This structure contains Global event status, System event status, Supply event status, Transceiver event status, WAKE pin event status.

Implements : sbc_evn_capt_t_Class

Definition at line 1438 of file sbc_uja1169_driver.h.

Data Fields

- sbc_gl_evnt_stat_t glEvnt
- sbc_sys_evnt_stat_t sysEvnt
- sbc_sup_evnt_stat_t supEvnt
- sbc_trans_evnt_stat_t transEvnt
- sbc_wake_evnt_stat_t wakePinEvnt

Field Documentation

16.88.2.24.1   sbc_gl_evnt_stat_t glEvnt

Global event status.

Definition at line 1439 of file sbc_uja1169_driver.h.

16.88.2.24.2   sbc_sup_evnt_stat_t supEvnt

Supply event status.

Definition at line 1441 of file sbc_uja1169_driver.h.

16.88.2.24.3   sbc_sys_evnt_stat_t sysEvnt

System event status.

Definition at line 1440 of file sbc_uja1169_driver.h.

16.88.2.24.4   sbc_trans_evnt_stat_t transEvnt

Transceiver event status.

Definition at line 1442 of file sbc_uja1169_driver.h.

16.88.2.24.5   sbc_wake_evnt_stat_t wakePinEvnt

WAKE pin event status.

Definition at line 1443 of file sbc_uja1169_driver.h.

16.88.2.25   struct sbc_mtpnv_stat_t

MTPNV status register. The MTPNV cells can be reprogrammed a maximum of 200 times (Ncy(W)MTP). Bit N←↩
VMPS in the MTPNV status register indicates whether the non-volatile cells can be reprogrammed. This register also contains a write counter, WRCNTS, that is incremented each time the MTPNV cells are reprogrammed (up to a maximum value of 111111; there is no overflow; performing a factory reset also increments the counter). This counter is provided for information purposes only; reprogramming will not be rejected when it reaches its maximum value.

Implements : sbc_mtpnv_stat_t_Class

Definition at line 1459 of file sbc_uja1169_driver.h.

Data Fields

- sbc_mtpnv_stat_wrcnts_t wrcnts
- sbc_mtpnv_stat_eccs_t eccs
- sbc_mtpnv_stat_nvmps_t nvmps

Field Documentation

16.88.2.25.1   sbc_mtpnv_stat_eccs_t eccs

Error correction code status.

Definition at line 1461 of file sbc_uja1169_driver.h.

16.88.2.25.2   sbc_mtpnv_stat_nvmps_t nvmps

Non-volatile memory programming status.

Definition at line 1462 of file sbc_uja1169_driver.h.

16.88.2.25.3   sbc_mtpnv_stat_wrcnts_t wrcnts

Write counter status.

Definition at line 1460 of file sbc_uja1169_driver.h.

16.88.2.26   struct sbc_status_group_t

Status group structure. All statuses of SBC are stored in this structure.

Implements : sbc_status_group_t_Class

Definition at line 1473 of file sbc_uja1169_driver.h.

Data Fields

- sbc_main_status_t mainS
- sbc_wtdog_status_t wtdog
- sbc_supply_status_t supply
- sbc_trans_stat_t trans
- sbc_wake_stat_wpvs_t wakePin
- sbc_evn_capt_t events

Field Documentation

16.88.2.26.1   sbc_evn_capt_t events

Event capture registers.

Definition at line 1479 of file sbc_uja1169_driver.h.

16.88.2.26.2   sbc_main_status_t mainS

Main status.

Definition at line 1474 of file sbc_uja1169_driver.h.

16.88.2.26.3   sbc_supply_status_t supply

Supply voltage status.

Definition at line 1476 of file sbc_uja1169_driver.h.

16.88.2.26.4 sbc_trans_stat_t trans

Transceiver status.

Definition at line 1477 of file sbc_uja1169_driver.h.

16.88.2.26.5 sbc_wake_stat_wpvs_t wakePin

WAKE pin status.

Definition at line 1478 of file sbc_uja1169_driver.h.

16.88.2.26.6 sbc_wtdog_status_t wtdog

Watchdog status.

Definition at line 1475 of file sbc_uja1169_driver.h.

16.88.3 Macro Definition Documentation

16.88.3.1 #define SBC_UJA_COUNT_DMASK 8U

Definition at line 44 of file sbc_uja1169_driver.h.

16.88.3.2 #define SBC_UJA_COUNT_ID_REG 4U

Definition at line 42 of file sbc_uja1169_driver.h.

16.88.3.3 #define SBC_UJA_COUNT_MASK 4U

Definition at line 43 of file sbc_uja1169_driver.h.

16.88.3.4 #define SBC_UJA_TIMEOUT 1000U

Timeout for the transfer in milliseconds. If the transfer takes longer than this time, the transfer is aborted and LPSPI_STATUS_SBC_UJA_TIMEOUT error is reported.

Definition at line 36 of file sbc_uja1169_driver.h.

16.88.4 Typedef Documentation

16.88.4.1 typedef uint8_t sbc_data_mask_t

Data mask registers. The data field indicates the nodes to be woken up. Within the data field, groups of nodes can be predefined and associated with bits in a data mask. By comparing the incoming data field with the data mask, multiple groups of nodes can be woken up simultaneously with a single wake-up message.

Implements : sbc_data_mask_t_Class

Definition at line 707 of file sbc_uja1169_driver.h.

16.88.4.2 typedef uint8_t sbc_fail_safe_rcc_t

Fail-safe control register, reset counter control (0x02). incremented every time the SBC enters Reset mode while FNMC = 0; RCC overflows from 11 to 00; default at power-on is 00.

Implements : sbc_fail_safe_rcc_t_Class

Definition at line 198 of file sbc_uja1169_driver.h.

16.88.4.3 typedef uint8_t sbc_frame_ctr_dlc_t

Frame control register, number of data bytes expected in a CAN frame (0x2F).

Implements : sbc_frame_ctr_dlc_t_Class

Definition at line 696 of file sbc_uja1169_driver.h.

16.88.4.4 typedef uint8_t sbc_identif_mask_t

ID mask registers (0x2B to 0x2E). The identifier mask is defined in the ID mask registers, where a 1 means dont care.

Implements : sbc_identif_mask_t_Class

Definition at line 662 of file sbc_uja1169_driver.h.

16.88.4.5 typedef uint8_t sbc_identifier_t

ID registers, identifier format (0x27 to 0x2A). A valid WUF identifier is defined and stored in the ID registers. An ID mask can be defined to allow a group of identifiers to be recognized as valid by an individual node.

Implements : sbc_identifier_t_Class

Definition at line 653 of file sbc_uja1169_driver.h.

16.88.4.6 typedef uint8_t sbc_mtpnv_stat_wrcnts_t

MTPNV status register, write counter status (0x70). 6-bits - contains the number of times the MTPNV cells were reprogrammed.

Implements : sbc_mtpnv_stat_wrcnts_t_Class

Definition at line 968 of file sbc_uja1169_driver.h.

16.88.5 Enumeration Type Documentation

16.88.5.1 enum sbc_can_cfdc_t

CAN control register, CAN FD control (0x20).

Implements : sbc_can_cfdc_t_Class

Enumerator

> ***SBC_UJA_CAN_CFDC_DIS*** CAN FD tolerance disabled.
>
> ***SBC_UJA_CAN_CFDC_EN*** CAN FD tolerance enabled.

Definition at line 461 of file sbc_uja1169_driver.h.

16.88.5.2 enum sbc_can_cmc_t

CAN control register, CAN mode control (0x20).

Implements : sbc_can_cmc_t_Class

Enumerator

> ***SBC_UJA_CAN_CMC_OFMODE*** Offline mode.
>
> ***SBC_UJA_CAN_CMC_ACMODE_DA*** Active mode (when the SBC is in Normal mode); CAN supply under-voltage detection active.
>
> ***SBC_UJA_CAN_CMC_ACMODE_DD*** Active mode (when the SBC is in Normal mode); CAN supply under-voltage detection disabled.
>
> ***SBC_UJA_CAN_CMC_LISTEN*** Listen-only mode.

Definition at line 497 of file sbc_uja1169_driver.h.

16.88.5.3   enum sbc_can_cpnc_t

CAN control register, CAN partial networking control (0x20).

Implements : sbc_can_cpnc_t_Class

Enumerator

> **SBC_UJA_CAN_CPNC_DIS**   Disable CAN selective wake-up.
>
> **SBC_UJA_CAN_CPNC_EN**   Enable CAN selective wake-up.

Definition at line 485 of file sbc_uja1169_driver.h.

16.88.5.4   enum sbc_can_pncok_t

CAN control register, CAN partial networking configuration OK (0x20).

Implements : sbc_can_pncok_t_Class

Enumerator

> **SBC_UJA_CAN_PNCOK_DIS**   Partial networking register configuration invalid (wake-up via standard wake-up pattern only).
>
> **SBC_UJA_CAN_PNCOK_EN**   Partial networking registers configured successfully.

Definition at line 473 of file sbc_uja1169_driver.h.

16.88.5.5   enum sbc_dat_rate_t

Data rate register, CAN data rate selection (0x26). CAN partial networking configuration registers. Dedicated registers are provided for configuring CAN partial networking.

Implements : sbc_dat_rate_t_Class

Enumerator

> **SBC_UJA_DAT_RATE_CDR_50KB**   50 kbit/s.
>
> **SBC_UJA_DAT_RATE_CDR_100KB**   100 kbit/s.
>
> **SBC_UJA_DAT_RATE_CDR_125KB**   125 kbit/s.
>
> **SBC_UJA_DAT_RATE_CDR_250KB**   250 kbit/s.
>
> **SBC_UJA_DAT_RATE_CDR_500KB**   500 kbit/s.
>
> **SBC_UJA_DAT_RATE_CDR_1000KB**   1000 kbit/s.

Definition at line 636 of file sbc_uja1169_driver.h.

16.88.5.6   enum sbc_fail_safe_lhc_t

Fail-safe control register, LIMP home control (0x02). The dedicated LIMP pin can be used to enable so called limp home hardware in the event of a serious ECU failure. Detectable failure conditions include SBC overtemperature events, loss of watchdog service, short-circuits on pins RSTN or V1 and user-initiated or external reset events. The LIMP pin is a battery-robust, active-LOW, open-drain output. The LIMP pin can also be forced LOW by setting bit LHC in the Fail-safe control register.

Implements : sbc_fail_safe_lhc_t_Class

Enumerator

> **SBC_UJA_FAIL_SAFE_LHC_FLOAT**   LIMP pin is floating.
>
> **SBC_UJA_FAIL_SAFE_LHC_LOW**   LIMP pin is driven LOW.

Definition at line 186 of file sbc_uja1169_driver.h.

16.88.5.7 enum sbc_frame_ctr_ide_t

Frame control register, identifier format (0x2F). The wake-up frame format, standard (11-bit) or extended (29-bit) identifier, is selected via bit IDE in the Frame control register.

Implements : sbc_frame_ctr_ide_t_Class

Enumerator

> ***SBC_UJA_FRAME_CTR_IDE_11B*** Standard frame format (11-bit).
> ***SBC_UJA_FRAME_CTR_IDE_29B*** Extended frame format (29-bit).

Definition at line 671 of file sbc_uja1169_driver.h.

16.88.5.8 enum sbc_frame_ctr_pndm_t

Frame control register, partial networking data mask (0x2F).

Implements : sbc_frame_ctr_pndm_t_Class

Enumerator

> ***SBC_UJA_FRAME_CTR_PNDM_DCARE*** Data length code and data field are do not care for wake-up.
> ***SBC_UJA_FRAME_CTR_PNDM_EVAL*** Data length code and data field are evaluated at wake-up.

Definition at line 683 of file sbc_uja1169_driver.h.

16.88.5.9 enum sbc_gl_evnt_stat_supe_t

Global event status register, supply event (0x60).

Implements : sbc_gl_evnt_stat_supe_t_Class

Enumerator

> ***SBC_UJA_GL_EVNT_STAT_SUPE_NO*** No pending supply event.
> ***SBC_UJA_GL_EVNT_STAT_SUPE*** Supply event pending at address 0x62 .

Definition at line 774 of file sbc_uja1169_driver.h.

16.88.5.10 enum sbc_gl_evnt_stat_syse_t

Global event status register, system event (0x60).

Implements : sbc_gl_evnt_stat_syse_t_Class

Enumerator

> ***SBC_UJA_GL_EVNT_STAT_SYSE_NO*** No pending system event.
> ***SBC_UJA_GL_EVNT_STAT_SYSE*** System event pending at address 0x61.

Definition at line 786 of file sbc_uja1169_driver.h.

16.88.5.11 enum sbc_gl_evnt_stat_trxe_t

Global event status register, transceiver event (0x60).

Implements : sbc_gl_evnt_stat_trxe_t_Class

Enumerator

> ***SBC_UJA_GL_EVNT_STAT_TRXE_NO*** No pending transceiver event.
> ***SBC_UJA_GL_EVNT_STAT_TRXE*** Transceiver event pending at address 0x63.

Definition at line 762 of file sbc_uja1169_driver.h.

16.88.5.12   enum sbc_gl_evnt_stat_wpe_t

Global event status register, WAKE pin event (0x60).

Implements : sbc_gl_evnt_stat_wpe_t_Class

Enumerator

> ***SBC_UJA_GL_EVNT_STAT_WPE_NO***   No pending WAKE pin event.
> ***SBC_UJA_GL_EVNT_STAT_WPE***   WAKE pin event pending at address 0x64.

Definition at line 750 of file sbc_uja1169_driver.h.

16.88.5.13   enum sbc_lock_t

Lock control(0x0A). Sections of the register address area can be write-protected to protect against unintended modifications. This facility only protects locked bits from being modified via the SPI and will not prevent the UJ←
A1169 updating status registers etc.

Implements : sbc_lock_t_Class

Enumerator

> ***LK0C***   Lock control 0: address area 0x06 to 0x09 - general-purpose memory macros. Lock control 1: address area 0x10 to 0x1F - regulator control macros.
> ***LK1C***   Lock control 2: address area 0x20 to 0x2F - transceiver control macros.
> ***LK2C***   Lock control 3: address area 0x30 to 0x3F - unused register range macros.
> ***LK3C***   Lock control 4: address area 0x40 to 0x4F - WAKE pin control macros.
> ***LK4C***   Lock control 5: address area 0x50 to 0x5F.
> ***LK5C***   Lock control 6: address area 0x68 to 0x6F macros.
> ***LK6C***   Lock control All: address area 0x10 to 0x6F macros.
> ***LKAC***

Definition at line 320 of file sbc_uja1169_driver.h.

16.88.5.14   enum sbc_main_nms_t

Main status register, normal mode status (0x03).

Implements : sbc_main_nms_t_Class

Enumerator

> ***SBC_UJA_MAIN_NMS_NORMAL***   UJA1169 has entered Normal mode (after power-up)
> ***SBC_UJA_MAIN_NMS_PWR_UP***   UJA1169 has powered up but has not yet switched to Normal mode.

Definition at line 217 of file sbc_uja1169_driver.h.

16.88.5.15   enum sbc_main_otws_t

Main status register, Overtemperature warning status (0x03).

Implements : sbc_main_otws_t_Class

Enumerator

> ***SBC_UJA_MAIN_OTWS_BELOW***   IC temperature below overtemperature warning threshold.
> ***SBC_UJA_MAIN_OTWS_ABOVE***   IC temperature above overtemperature warning threshold.

Definition at line 205 of file sbc_uja1169_driver.h.

16.88.5.16 enum sbc_main_rss_t

Main status register, Reset source status (0x03).

Implements : sbc_main_rss_t_Class

Enumerator

> **SBC_UJA_MAIN_RSS_OFF_MODE** Left Off mode (power-on).
> **SBC_UJA_MAIN_RSS_CAN_WAKEUP** CAN wake-up in Sleep mode.
> **SBC_UJA_MAIN_RSS_SLP_WAKEUP** Wake-up via WAKE pin in Sleep mode.
> **SBC_UJA_MAIN_RSS_OVF_SLP** Watchdog overflow in Sleep mode (Timeout mode).
> **SBC_UJA_MAIN_RSS_DIAG_WAKEUP** Diagnostic wake-up in Sleep mode
> **SBC_UJA_MAIN_RSS_WATCH_TRIG** Watchdog triggered too early (Window mode).
> **SBC_UJA_MAIN_RSS_WATCH_OVF** Watchdog overflow (Window mode or Timeout mode with WDF = 1)
> **SBC_UJA_MAIN_RSS_ILLEG_WATCH** Illegal watchdog mode control access.
> **SBC_UJA_MAIN_RSS_RSTN_PULDW** RSTN pulled down externally.
> **SBC_UJA_MAIN_RSS_LFT_OVERTM** Left Overtemp mode.
> **SBC_UJA_MAIN_RSS_V1_UNDERV** V1 undervoltage.
> **SBC_UJA_MAIN_RSS_ILLEG_SLP** Illegal Sleep mode command received.
> **SBC_UJA_MAIN_RSS_WAKE_SLP** Wake-up from Sleep mode due to a frame detect error

Definition at line 229 of file sbc_uja1169_driver.h.

16.88.5.17 enum sbc_mode_mc_t

Mode control register, mode control (0x01)

Implements : sbc_mode_mc_t_Class

Enumerator

> **SBC_UJA_MODE_MC_SLEEP** Sleep mode.
> **SBC_UJA_MODE_MC_STANDBY** Standby mode.
> **SBC_UJA_MODE_MC_NORMAL** Normal mode.

Definition at line 168 of file sbc_uja1169_driver.h.

16.88.5.18 enum sbc_mtpnv_stat_eccs_t

MTPNV status register, error correction code status (0x70).

Implements : sbc_mtpnv_stat_eccs_t_Class

Enumerator

> **SBC_UJA_MTPNV_STAT_ECCS_NO** No bit failure detected in non-volatile memory.
> **SBC_UJA_MTPNV_STAT_ECCS** Bit failure detected and corrected in non-volatile memory.

Definition at line 975 of file sbc_uja1169_driver.h.

16.88.5.19 enum sbc_mtpnv_stat_nvmps_t

MTPNV status register, non-volatile memory programming status (0x70).

Implements : sbc_mtpnv_stat_nvmps_t_Class

Enumerator

> **SBC_UJA_MTPNV_STAT_NVMPS_NO** MTPNV memory cannot be overwritten.
> **SBC_UJA_MTPNV_STAT_NVMPS** MTPNV memory is ready to be reprogrammed.

Definition at line 987 of file sbc_uja1169_driver.h.

16.88.5.20   enum sbc_register_t

Register map.

Implements : sbc_register_t_Class

Enumerator

 SBC_UJA_WTDOG_CTR
 SBC_UJA_MODE
 SBC_UJA_FAIL_SAFE
 SBC_UJA_MAIN
 SBC_UJA_SYSTEM_EVNT
 SBC_UJA_WTDOG_STAT
 SBC_UJA_MEMORY_0
 SBC_UJA_MEMORY_1
 SBC_UJA_MEMORY_2
 SBC_UJA_MEMORY_3
 SBC_UJA_LOCK
 SBC_UJA_REGULATOR
 SBC_UJA_SUPPLY_STAT
 SBC_UJA_SUPPLY_EVNT
 SBC_UJA_CAN
 SBC_UJA_TRANS_STAT
 SBC_UJA_TRANS_EVNT
 SBC_UJA_DAT_RATE
 SBC_UJA_IDENTIF_0
 SBC_UJA_IDENTIF_1
 SBC_UJA_IDENTIF_2
 SBC_UJA_IDENTIF_3
 SBC_UJA_MASK_0
 SBC_UJA_MASK_1
 SBC_UJA_MASK_2
 SBC_UJA_MASK_3
 SBC_UJA_FRAME_CTR
 SBC_UJA_DAT_MASK_0
 SBC_UJA_DAT_MASK_1
 SBC_UJA_DAT_MASK_2
 SBC_UJA_DAT_MASK_3
 SBC_UJA_DAT_MASK_4
 SBC_UJA_DAT_MASK_5
 SBC_UJA_DAT_MASK_6
 SBC_UJA_DAT_MASK_7
 SBC_UJA_WAKE_STAT
 SBC_UJA_WAKE_EN
 SBC_UJA_GL_EVNT_STAT
 SBC_UJA_SYS_EVNT_STAT
 SBC_UJA_SUP_EVNT_STAT

SBC_UJA_TRANS_EVNT_STAT

SBC_UJA_WAKE_EVNT_STAT

SBC_UJA_MTPNV_STAT

SBC_UJA_START_UP

SBC_UJA_SBC

SBC_UJA_MTPNV_CRC

SBC_UJA_IDENTIF

Definition at line 54 of file sbc_uja1169_driver.h.

16.88.5.21 enum sbc_regulator_pdc_t

Regulator control register, power distribution control (0x10).

Implements : sbc_regulator_pdc_t_Class

Enumerator

SBC_UJA_REGULATOR_PDC_HV V1 threshold current for activating the external PNP transistor, load current rising; Ith(act)PNP (higher value) V1 threshold current for deactivating the external PNP transistor, load current falling; Ith(deact)PNP (higher value).

SBC_UJA_REGULATOR_PDC_LV V1 threshold current for activating the external PNP transistor; load current rising; Ith(act)PNP (lower value) V1 threshold current for deactivating the external PNP transistor; load current falling; Ith(deact)PNP (lower value).

Definition at line 345 of file sbc_uja1169_driver.h.

16.88.5.22 enum sbc_regulator_v1rtc_t

Regulator control register, set V1 reset threshold (0x10).

Implements : sbc_regulator_v1rtc_t_Class

Enumerator

SBC_UJA_REGULATOR_V1RTC_90 Reset threshold set to 90 % of V1 nominal output voltage.

SBC_UJA_REGULATOR_V1RTC_80 Reset threshold set to 80 % of V1 nominal output voltage.

SBC_UJA_REGULATOR_V1RTC_70 Reset threshold set to 70 % of V1 nominal output voltage.

SBC_UJA_REGULATOR_V1RTC_60 Reset threshold set to 60 % of V1 nominal output voltage.

Definition at line 379 of file sbc_uja1169_driver.h.

16.88.5.23 enum sbc_regulator_v2c_t

Regulator control register, V2/VEXT configuration (0x10).

Implements : sbc_regulator_v2c_t_Class

Enumerator

SBC_UJA_REGULATOR_V2C_OFF V2/VEXT off in all modes.

SBC_UJA_REGULATOR_V2C_N V2/VEXT on in Normal mode.

SBC_UJA_REGULATOR_V2C_N_S_R V2/VEXT on in Normal, Standby and Reset modes.

SBC_UJA_REGULATOR_V2C_N_S_S_R V2/VEXT on in Normal, Standby, Sleep and Reset modes.

Definition at line 363 of file sbc_uja1169_driver.h.

16.88.5.24   enum sbc_sbc_fnmc_t

SBC configuration control register, Forced Normal mode control (0x74).

Implements : sbc_sbc_fnmc_t_Class

Enumerator

> **SBC_UJA_SBC_FNMC_DIS**   Forced Normal mode disabled.
> **SBC_UJA_SBC_FNMC_EN**   Forced Normal mode enabled.

Definition at line 1044 of file sbc_uja1169_driver.h.

16.88.5.25   enum sbc_sbc_sdmc_t

SBC configuration control register, Software Development mode control (0x74).

Implements : sbc_sbc_sdmc_t_Class

Enumerator

> **SBC_UJA_SBC_SDMC_DIS**   Software Development mode disabled.
> **SBC_UJA_SBC_SDMC_EN**   Software Development mode enabled.

Definition at line 1057 of file sbc_uja1169_driver.h.

16.88.5.26   enum sbc_sbc_slpc_t

SBC configuration control register, Sleep control (0x74).

Implements : sbc_sbc_slpc_t_Class

Enumerator

> **SBC_UJA_SBC_SLPC_AC**   Sleep mode commands accepted. Factory preset value.
> **SBC_UJA_SBC_SLPC_IG**   Sleep mode commands ignored.

Definition at line 1070 of file sbc_uja1169_driver.h.

16.88.5.27   enum sbc_sbc_v1rtsuc_t

SBC configuration control register, V1 undervoltage threshold (defined by bit V1RTC) at start-up (0x74).

Implements : sbc_sbc_v1rtsuc_t_Class

Enumerator

> **SBC_UJA_SBC_V1RTSUC_90**   V1 undervoltage detection at 90 % of nominal value at start-up (V1RTC = 00).
> **SBC_UJA_SBC_V1RTSUC_80**   V1 undervoltage detection at 80 % of nominal value at start-up (V1RTC = 01).
> **SBC_UJA_SBC_V1RTSUC_70**   V1 undervoltage detection at 70 % of nominal value at start-up V1RTC = 10).

> **SBC_UJA_SBC_V1RTSUC_60**   V1 undervoltage detection at 60 % of nominal value at start-up (V1RTC = 11).

Definition at line 1028 of file sbc_uja1169_driver.h.

16.88.5.28   enum sbc_start_up_rlc_t

Start-up control register, RSTN output reset pulse width macros (0x73).

Implements : sbc_start_up_rlc_t_Class

Enumerator

> ***SBC_UJA_START_UP_RLC_20_25p0*** Tw(rst) = 20 ms to 25 ms.
>
> ***SBC_UJA_START_UP_RLC_10_12p5*** Tw(rst) = 10 ms to 12.5 ms.
>
> ***SBC_UJA_START_UP_RLC_03p6_05*** Tw(rst) = 3.6 ms to 5 ms.
>
> ***SBC_UJA_START_UP_RLC_01_01p5*** Tw(rst) = 1 ms to 1.5 ms.

Definition at line 999 of file sbc_uja1169_driver.h.

16.88.5.29   enum sbc_start_up_v2suc_t

Start-up control register, V2/VEXT start-up control (0x73).

Implements : sbc_start_up_v2suc_t_Class

Enumerator

> ***SBC_UJA_START_UP_V2SUC_00*** bits V2C/VEXTC set to 00 at power-up.
>
> ***SBC_UJA_START_UP_V2SUC_11*** bits V2C/VEXTC set to 11 at power-up.

Definition at line 1015 of file sbc_uja1169_driver.h.

16.88.5.30   enum sbc_sup_evnt_stat_v1u_t

Supply event status register, V1 undervoltage (0x62).

Implements : sbc_sup_evnt_stat_v1u_t_Class

Enumerator

> ***SBC_UJA_SUP_EVNT_STAT_V1U_NO*** no V1 undervoltage event captured.
>
> ***SBC_UJA_SUP_EVNT_STAT_V1U*** voltage on V1 has dropped below the 90 % undervoltage threshold while V1 is active (event is not captured in Sleep mode because V1 is off); V1U event capture is independent of the setting of bits V1RTC.

Definition at line 877 of file sbc_uja1169_driver.h.

16.88.5.31   enum sbc_sup_evnt_stat_v2o_t

Supply event status register, V2/VEXT overvoltage (0x62).

Implements : sbc_sup_evnt_stat_v2o_t_Class

Enumerator

> ***SBC_UJA_SUP_EVNT_STAT_V2O_NO*** No V2/VEXT overvoltage event captured.
>
> ***SBC_UJA_SUP_EVNT_STAT_V2O*** V2/VEXT overvoltage event captured.

Definition at line 853 of file sbc_uja1169_driver.h.

16.88.5.32   enum sbc_sup_evnt_stat_v2u_t

Supply event status register, V2/VEXT undervoltage (0x62).

Implements : sbc_sup_evnt_stat_v2u_t_Class

Enumerator

> ***SBC_UJA_SUP_EVNT_STAT_V2U_NO*** No V2/VEXT undervoltage event captured.
>
> ***SBC_UJA_SUP_EVNT_STAT_V2U*** V2/VEXT undervoltage event captured.

Definition at line 865 of file sbc_uja1169_driver.h.

16.88.5.33   enum sbc_supply_evnt_v1ue_t

Supply event capture enable register, V1 undervoltage enable (0x1C).

Implements : sbc_supply_evnt_v1ue_t_Class

Enumerator

> ***SBC_UJA_SUPPLY_EVNT_V1UE_DIS***   V1 undervoltage detection disabled.
> ***SBC_UJA_SUPPLY_EVNT_V1UE_EN***   V1 undervoltage detection enabled.

Definition at line 449 of file sbc_uja1169_driver.h.

16.88.5.34   enum sbc_supply_evnt_v2oe_t

Supply event capture enable register, V2/VEXT overvoltage enable (0x1C).

Implements : sbc_supply_evnt_v2oe_t_Class

Enumerator

> ***SBC_UJA_SUPPLY_EVNT_V2OE_DIS***   V2/VEXT overvoltage detection disabled.
> ***SBC_UJA_SUPPLY_EVNT_V2OE_EN***   V2/VEXT overvoltage detection enabled.

Definition at line 424 of file sbc_uja1169_driver.h.

16.88.5.35   enum sbc_supply_evnt_v2ue_t

Supply event capture enable register, V2/VEXT undervoltage enable (0x1C).

Implements : sbc_supply_evnt_v2ue_t_Class

Enumerator

> ***SBC_UJA_SUPPLY_EVNT_V2UE_DIS***   V2/VEXT undervoltage detection disabled.
> ***SBC_UJA_SUPPLY_EVNT_V2UE_EN***   V2/VEXT undervoltage detection enabled.

Definition at line 437 of file sbc_uja1169_driver.h.

16.88.5.36   enum sbc_supply_stat_v1s_t

Supply voltage status register, V1 status (0x1B).

Implements : sbc_supply_stat_v1s_t_Class

Enumerator

> ***SBC_UJA_SUPPLY_STAT_V1S_VAB***   V1 output voltage above 90 % undervoltage threshold.
> ***SBC_UJA_SUPPLY_STAT_V1S_VBE***   V1 output voltage below 90 % undervoltage threshold.

Definition at line 411 of file sbc_uja1169_driver.h.

16.88.5.37   enum sbc_supply_stat_v2s_t

Supply voltage status register, V2/VEXT status (0x1B).

Implements : sbc_supply_stat_v2s_t_Class

Enumerator

> ***SBC_UJA_SUPPLY_STAT_V2S_VOK***   V2/VEXT voltage ok.
> ***SBC_UJA_SUPPLY_STAT_V2S_VBE***   V2/VEXT output voltage below undervoltage threshold
> ***SBC_UJA_SUPPLY_STAT_V2S_VAB***   V2/VEXT output voltage above overvoltage threshold
> ***SBC_UJA_SUPPLY_STAT_V2S_DIS***   V2/VEXT disabled

Definition at line 395 of file sbc_uja1169_driver.h.

16.88.5.38 enum sbc_sys_evnt_otwe_t

System event capture enable, overtemperature warning enable (0x04).

Implements : sbc_sys_evnt_otwe_t_Class

Enumerator

SBC_UJA_SYS_EVNT_OTWE_DIS Overtemperature warning disabled.

SBC_UJA_SYS_EVNT_OTWE_EN Overtemperature warning enabled.

Definition at line 254 of file sbc_uja1169_driver.h.

16.88.5.39 enum sbc_sys_evnt_spife_t

System event capture enable, SPI failure enable (0x04).

Implements : sbc_sys_evnt_spife_t_Class

Enumerator

SBC_UJA_SYS_EVNT_SPIFE_DIS SPI failure detection disabled.

SBC_UJA_SYS_EVNT_SPIFE_EN SPI failure detection enabled.

Definition at line 266 of file sbc_uja1169_driver.h.

16.88.5.40 enum sbc_sys_evnt_stat_otw_t

System event status register, overtemperature warning (0x61).

Implements : sbc_sys_evnt_stat_otw_t_Class

Enumerator

SBC_UJA_SYS_EVNT_STAT_OTW_NO Overtemperature not detected.

SBC_UJA_SYS_EVNT_STAT_OTW The global chip temperature has exceeded the overtemperature warning threshold, Tth(warn)otp (not in Sleep mode).

Definition at line 810 of file sbc_uja1169_driver.h.

16.88.5.41 enum sbc_sys_evnt_stat_po_t

System event status register, power-on (0x61).

Implements : sbc_sys_evnt_stat_po_t_Class

Enumerator

SBC_UJA_SYS_EVNT_STAT_PO_NO No recent battery power-on.

SBC_UJA_SYS_EVNT_STAT_PO The UJA1169 has left Off mode after battery power-on.

Definition at line 798 of file sbc_uja1169_driver.h.

16.88.5.42 enum sbc_sys_evnt_stat_spif_t

System event status register, SPI failure (0x61).

Implements : sbc_sys_evnt_stat_spif_t_Class

Enumerator

SBC_UJA_SYS_EVNT_STAT_SPIF_NO No SPI failure detected

SBC_UJA_SYS_EVNT_STAT_SPIF SPI clock count error (only 16-, 24- and 32-bit commands are valid), illegal WMC, NWP or MC code or attempted write access to locked register (not in Sleep mode)

Definition at line 823 of file sbc_uja1169_driver.h.

16.88.5.43   enum sbc_sys_evnt_stat_wdf_t

System event status register, watchdog failure (0x61).

Implements : sbc_sys_evnt_stat_wdf_t_Class

Enumerator

> ***SBC_UJA_SYS_EVNT_STAT_WDF_NO***   No watchdog failure event captured
>
> ***SBC_UJA_SYS_EVNT_STAT_WDF***   Watchdog overflow in Window or Timeout mode or watchdog triggered too early in Window mode; a system reset is triggered immediately in response to a watchdog failure in Window mode; when the watchdog overflows in Timeout mode, a system reset is only performed if a WDF is already pending (WDF = 1).

Definition at line 837 of file sbc_uja1169_driver.h.

16.88.5.44   enum sbc_trans_evnt_cbse_t

Transceiver event capture enable register, CAN-bus silence enable (0x23).

Implements : sbc_trans_evnt_cbse_t_Class

Enumerator

> ***SBC_UJA_TRANS_EVNT_CBSE_DIS***   CAN-bus silence detection disabled.
>
> ***SBC_UJA_TRANS_EVNT_CBSE_EN***   CAN-bus silence detection enabled.

Definition at line 598 of file sbc_uja1169_driver.h.

16.88.5.45   enum sbc_trans_evnt_cfe_t

Transceiver event capture enable register, CAN failure enable (0x23).

Implements : sbc_trans_evnt_cfe_t_Class

Enumerator

> ***SBC_UJA_TRANS_EVNT_CFE_DIS***   CAN failure detection disabled.
>
> ***SBC_UJA_TRANS_EVNT_CFE_EN***   CAN failure detection enabled.

Definition at line 610 of file sbc_uja1169_driver.h.

16.88.5.46   enum sbc_trans_evnt_cwe_t

Transceiver event capture enable register, CAN wake-up enable (0x23).

Implements : sbc_trans_evnt_cwe_t_Class

Enumerator

> ***SBC_UJA_TRANS_EVNT_CWE_DIS***   CAN wake-up detection disabled.
>
> ***SBC_UJA_TRANS_EVNT_CWE_EN***   CAN wake-up detection enabled.

Definition at line 622 of file sbc_uja1169_driver.h.

16.88.5.47   enum sbc_trans_evnt_stat_cbs_t

Transceiver event status register, CAN-bus status (0x63).

Implements : sbc_trans_evnt_stat_cbs_t_Class

Enumerator

> ***SBC_UJA_TRANS_EVNT_STAT_CBS_NO***   CAN-bus active.

> **SBC_UJA_TRANS_EVNT_STAT_CBS** No activity on CAN-bus for tto(silence) (detected only when CBSE = 1 while bus active).

Definition at line 904 of file sbc_uja1169_driver.h.

16.88.5.48 enum sbc_trans_evnt_stat_cf_t

Transceiver event status register, CAN failure (0x63).

Implements : sbc_trans_evnt_stat_cf_t_Class

Enumerator

> **SBC_UJA_TRANS_EVNT_STAT_CF_NO** No CAN failure detected.
>
> **SBC_UJA_TRANS_EVNT_STAT_CF** CAN transceiver deactivated due to VCAN undervoltage OR dominant clamped TXD (not in Sleep mode)

Definition at line 917 of file sbc_uja1169_driver.h.

16.88.5.49 enum sbc_trans_evnt_stat_cw_t

Transceiver event status register, CAN wake-up (0x63).

Implements : sbc_trans_evnt_stat_cw_t_Class

Enumerator

> **SBC_UJA_TRANS_EVNT_STAT_CW_NO** No CAN wake-up event detected.
>
> **SBC_UJA_TRANS_EVNT_STAT_CW** CAN wake-up event detected while the transceiver is in CAN Offline Mode.

Definition at line 930 of file sbc_uja1169_driver.h.

16.88.5.50 enum sbc_trans_evnt_stat_pnfde_t

Transceiver event status register,partial networking frame detection error (0x63).

Implements : sbc_trans_evnt_stat_pnfde_t_Class

Enumerator

> **SBC_UJA_TRANS_EVNT_STAT_PNFDE_NO** No partial networking frame detection error detected.
>
> **SBC_UJA_TRANS_EVNT_STAT_PNFDE** Partial networking frame detection error detected.

Definition at line 892 of file sbc_uja1169_driver.h.

16.88.5.51 enum sbc_trans_stat_cbss_t

Transceiver status register, CAN-bus silence status (0x22).

Implements : sbc_trans_stat_cbss_t_Class

Enumerator

> **SBC_UJA_TRANS_STAT_CBSS_ACT** CAN-bus active (communication detected on bus)
>
> **SBC_UJA_TRANS_STAT_CBSS_INACT** CAN-bus inactive (for longer than t_to(silence)).

Definition at line 562 of file sbc_uja1169_driver.h.

16.88.5.52 enum sbc_trans_stat_cfs_t

Transceiver status register, CAN failure status (0x22).

Implements : sbc_trans_stat_cfs_t_Class

Enumerator

  SBC_UJA_TRANS_STAT_CFS_NO_TXD   No TXD dominant time-out event detected.

  SBC_UJA_TRANS_STAT_CFS_TXD   CAN transmitter disabled due to a TXD dominant time-out event.

Definition at line 586 of file sbc_uja1169_driver.h.

16.88.5.53   enum sbc_trans_stat_coscs_t

Transceiver status register, CAN oscillator status (0x22).

Implements : sbc_trans_stat_coscs_t_Class

Enumerator

  SBC_UJA_TRANS_STAT_COSCS_NRUN   CAN partial networking oscillator not running at target frequency.

  SBC_UJA_TRANS_STAT_COSCS_RUN   CAN partial networking oscillator running at target.

Definition at line 550 of file sbc_uja1169_driver.h.

16.88.5.54   enum sbc_trans_stat_cpnerr_t

Transceiver status register, CAN partial networking error (0x22).

Implements : sbc_trans_stat_cpnerr_t_Class

Enumerator

  SBC_UJA_TRANS_STAT_CPNERR_NO_DET   no CAN partial networking error detected (PNFDE = 0 AND PNCOK = 1).

  SBC_UJA_TRANS_STAT_CPNERR_DET   CAN partial networking error detected (PNFDE = 1OR PNCOK = 0; wake-up via standard wake-up pattern only).

Definition at line 525 of file sbc_uja1169_driver.h.

16.88.5.55   enum sbc_trans_stat_cpns_t

Transceiver status register, CAN partial networking status (0x22).

Implements : sbc_trans_stat_cpns_t_Class

Enumerator

  SBC_UJA_TRANS_STAT_CPNS_ERR   CAN partial networking configuration error detected(PNCOK = 0).

  SBC_UJA_TRANS_STAT_CPNS_OK   CAN partial networking configuration ok (PNCOK = 1).

Definition at line 538 of file sbc_uja1169_driver.h.

16.88.5.56   enum sbc_trans_stat_cts_t

Transceiver status register, CAN transceiver status (0x22).

Implements : sbc_trans_stat_cts_t_Class

Enumerator

  SBC_UJA_TRANS_STAT_CTS_INACT   CAN transceiver not in Active mode.

  SBC_UJA_TRANS_STAT_CTS_ACT   CAN transceiver in Active mode.

Definition at line 513 of file sbc_uja1169_driver.h.

16.88.5.57 enum sbc_trans_stat_vcs_t

Transceiver status register, VCAN status (0x22).

Implements : sbc_trans_stat_vcs_t_Class

Enumerator

> ***SBC_UJA_TRANS_STAT_VCS_AB*** CAN supply voltage is above the 90 % threshold.
> ***SBC_UJA_TRANS_STAT_VCS_BE*** CAN supply voltage is below the 90 % threshold

Definition at line 574 of file sbc_uja1169_driver.h.

16.88.5.58 enum sbc_wake_en_wpfe_t

WAKE pin event capture enable register, WAKE pin falling-edge enable (0x4C).

Implements : sbc_wake_en_wpfe_t_Class

Enumerator

> ***SBC_UJA_WAKE_EN_WPFE_DIS*** Falling-edge detection on WAKE pin disabled.
> ***SBC_UJA_WAKE_EN_WPFE_EN*** Falling-edge detection on WAKE pin enabled.

Definition at line 738 of file sbc_uja1169_driver.h.

16.88.5.59 enum sbc_wake_en_wpre_t

WAKE pin event capture enable register, WAKE pin rising-edge enable (0x4C).

Implements : sbc_wake_en_wpre_t_Class

Enumerator

> ***SBC_UJA_WAKE_EN_WPRE_DIS*** Rising-edge detection on WAKE pin disabled.
> ***SBC_UJA_WAKE_EN_WPRE_EN*** Rising-edge detection on WAKE pin enabled.

Definition at line 726 of file sbc_uja1169_driver.h.

16.88.5.60 enum sbc_wake_evnt_stat_wpf_t

WAKE pin event status register, WAKE pin falling edge (0x64).

Implements : sbc_wake_evnt_stat_wpf_t_Class

Enumerator

> ***SBC_UJA_WAKE_EVNT_STAT_WPF_NO*** No falling edge detected on WAKE pin.
> ***SBC_UJA_WAKE_EVNT_STAT_WPF*** Falling edge detected on WAKE pin.

Definition at line 954 of file sbc_uja1169_driver.h.

16.88.5.61 enum sbc_wake_evnt_stat_wpr_t

WAKE pin event status register, WAKE pin rising edge (0x64).

Implements : sbc_wake_evnt_stat_wpr_t_Class

Enumerator

> ***SBC_UJA_WAKE_EVNT_STAT_WPR_NO*** No rising edge detected on WAKE pin.
> ***SBC_UJA_WAKE_EVNT_STAT_WPR*** Rising edge detected on WAKE pin.

Definition at line 942 of file sbc_uja1169_driver.h.

16.88.5.62   enum sbc_wake_stat_wpvs_t

WAKE pin status register, WAKE pin status (0x4B).

Implements : sbc_wake_stat_wpvs_t_Class

Enumerator

> ***SBC_UJA_WAKE_STAT_WPVS_BE***   Voltage on WAKE pin below switching threshold (Vth(sw)).
>
> ***SBC_UJA_WAKE_STAT_WPVS_AB***   voltage on WAKE pin above switching threshold (Vth(sw)).

Definition at line 714 of file sbc_uja1169_driver.h.

16.88.5.63   enum sbc_wtdog_ctr_nwp_t

Watchdog control register, nominal watchdog period (0x00). Eight watchdog periods are supported, from 8 ms to 4096 ms. The watchdog period is programmed via bits NWP. The selected period is valid for both Window and Timeout modes. The default watchdog period is 128 ms. A watchdog trigger event resets the watchdog timer. A watchdog trigger event is any valid write access to the Watchdog control register. If the watchdog mode or the watchdog period have changed as a result of the write access, the new values are immediately valid.

Implements : sbc_wtdog_ctr_nwp_t_Class

Enumerator

> ***SBC_UJA_WTDOG_CTR_NWP_8***   8 ms.
>
> ***SBC_UJA_WTDOG_CTR_NWP_16***   16 ms.
>
> ***SBC_UJA_WTDOG_CTR_NWP_32***   32 ms.
>
> ***SBC_UJA_WTDOG_CTR_NWP_64***   64 ms.
>
> ***SBC_UJA_WTDOG_CTR_NWP_128***   128 ms.
>
> ***SBC_UJA_WTDOG_CTR_NWP_256***   256 ms.
>
> ***SBC_UJA_WTDOG_CTR_NWP_1024***   1024 ms.
>
> ***SBC_UJA_WTDOG_CTR_NWP_4096***   4096 ms.

Definition at line 152 of file sbc_uja1169_driver.h.

16.88.5.64   enum sbc_wtdog_ctr_wmc_t

Watchdog control register, watchdog mode control (0x00). The UJA1169 contains a watchdog that supports three operating modes: Window, Timeout and Autonomous. In Window mode (available only in SBC Normal mode), a watchdog trigger event within a defined watchdog window triggers and resets the watchdog timer. In Timeout mode, the watchdog runs continuously and can be triggered and reset at any time within the watchdog period by a watchdog trigger. Watchdog time-out mode can also be used for cyclic wake-up of the microcontroller. In Autonomous mode, the watchdog can be off or autonomously in Timeout mode, depending on the selected SBC mode. The watchdog mode is selected via bits WMC in the Watchdog control register. The SBC must be in Standby mode when the watchdog mode is changed.

Implements : sbc_wtdog_ctr_wmc_t_Class

Enumerator

> ***SBC_UJA_WTDOG_CTR_WMC_AUTO***   Autonomous mode.
>
> ***SBC_UJA_WTDOG_CTR_WMC_TIME***   Timeout mode.
>
> ***SBC_UJA_WTDOG_CTR_WMC_WIND***   Window mode (available only in SBC Normal mode).

Definition at line 131 of file sbc_uja1169_driver.h.

16.88.5.65 enum sbc_wtdog_stat_fnms_t

Watchdog status register, forced Normal mode status (0x05).

Implements : sbc_wtdog_stat_fnms_t_Class

Enumerator

    SBC_UJA_WTDOG_STAT_FNMS_N_NORMAL   SBC is not in Forced Normal mode.

    SBC_UJA_WTDOG_STAT_FNMS_NORMAL   SBC is in Forced Normal mode.

Definition at line 278 of file sbc_uja1169_driver.h.

16.88.5.66 enum sbc_wtdog_stat_sdms_t

Watchdog status register, Software Development mode status (0x05).

Implements : sbc_wtdog_stat_sdms_t_Class

Enumerator

    SBC_UJA_WTDOG_STAT_SDMS_N_NORMAL   SBC is not in Software Development mode.

    SBC_UJA_WTDOG_STAT_SDMS_NORMAL   SBC is in Software Development mode.

Definition at line 290 of file sbc_uja1169_driver.h.

16.88.5.67 enum sbc_wtdog_stat_wds_t

Watchdog status register, watchdog status (0x05).

Implements : sbc_wtdog_stat_wds_t_Class

Enumerator

    SBC_UJA_WTDOG_STAT_WDS_OFF   Watchdog is off.

    SBC_UJA_WTDOG_STAT_WDS_FIH   Watchdog is in first half of the nominal period.

    SBC_UJA_WTDOG_STAT_WDS_SEH   Watchdog is in second half of the nominal period.

Definition at line 302 of file sbc_uja1169_driver.h.

16.89 User provided call-outs

16.89.1 Detailed Description

This group contains APIs which may be called from within the LIN module in order to enable/disable LIN communication interrupts.

Functions

- l_u16 l_sys_irq_disable (l_ifc_handle iii)

    Disable LIN related IRQ.
- void l_sys_irq_restore (l_ifc_handle iii)

    Enable LIN related IRQ.

16.89.2 Function Documentation

16.89.2.1 l_u16 l_sys_irq_disable (l_ifc_handle *iii*)

Disable LIN related IRQ.

Parameters

in	*iii*	Interface name

Returns

> l_u16

Definition at line 558 of file lin_common_api.c.

16.89.2.2 void l_sys_irq_restore (l_ifc_handle *iii*)

Enable LIN related IRQ.

Parameters

in	*iii*	Interface name

Returns

> void

Definition at line 572 of file lin_common_api.c.

16.90 WDOG Driver

16.90.1 Detailed Description

Watchdog Timer Peripheral Driver.

How to use the WDOG driver in your application

In order to be able to use the Watchdog in your application, the first thing to do is initializing it with the desired configuration. This is done by calling the **WDOG_DRV_Init** function. One of the arguments passed to this function is the configuration which will be used for the Watchdog, specified by the **wdog_user_config_t** structure.

The **wdog_user_config_t** structure allows you to configure the following:

- the clock source of the Watchdog;

- the prescaler (a fixed 256 pre-scaling of the Watchdog counter reference clock may be enabled);

- the operation modes in which the Watchdog is functional (by default, the Watchdog is not functional in Debug mode, Wait mode or Stop mode);

- the timeout value to which the Watchdog counter is compared;

- the window mode option for the refresh mechanism (by default, the window mode is disabled, but it may be enabled and a window value may be set);

- the Watchdog timeout interrupt (if enabled, after a reset-triggering event, the Watchdog first generates an interrupt request; next, the Watchdog delays 128 bus clocks before forcing a reset, to allow the interrupt service routine to perform tasks (like analyzing the stack to debug code));

- the update mechanism (by default, the Watchdog reconfiguration is enabled, but updates can be disabled)

Please note that if the updates are disabled the Watchdog cannot be later modified without forcing a reset (this implies that further calls of the **WDOG_DRV_Init**, **WDOG_DRV_Deinit** or **WDOG_DRV_SetInt** functions will lead to a reset).

As mentioned before, a timeout interrupt may be enabled by specifying it at the module initialization. The **WDO←G_DRV_Init** only allows enabling/disabling the interrupt, and it does not set up the ISR to be used for the interrupt request. In order to set up a function to be called after a reset-triggering event (and also enable/disable the interrupt), the **WDOG_DRV_SetInt** function may be used. **Please note** that, due to the 128 bus clocks delay before the reset, a limited amount of job can be done in the ISR.

Basic Operations of WDOG

1. To initialize WDOG, call WDOG_DRV_Init() with an user configuration structure. In the following code, WDOG is initialized with default settings.

```
#define INST_WDOG1 (0U)

wdog_user_config_t userConfigPtr = {
    .WDOG_LPO_CLOCK,           /* Use the LPO clock as source */
    .opMode = {                /* WDOG not functional in Wait/Debug/Stop mode */
        false,
        false,
        false
    },
    .true,                     /* Enable further updates of the WDOG configuration */
    .false,                    /* Timeout interrupt disabled */
    .false,                    /* Window mode disabled */
    .0U,                       /* Window value */
    .0x400,                    /* Timeout value */
    .false                     /* Prescaler disabled */
};

/* Initialize WDOG module */
WDOG_DRV_Init(INST_WDOG1, &userConfigPtr);
```

2. To get default configuration of WDOG module, just call the function WDOG_DRV_GetDefaultConfig(). Make sure that the operation before WDOG timeout executing.

```
wdog_user_config_t userConfigPtr;

/* Get default configuration of WDOG module */
WDOG_DRV_GetDefaultConfig(&userConfigPtr);
```

3. To refresh WDOG counter of WDOG module, just call the function WDOG_DRV_Trigger(). Make sure that the operation before WDOG timeout executing.

```
/* Refresh counter of WDOG counter */
WDOG_DRV_Trigger(INST_WDOG1);
```

4. To de-initialize WDOG module, just call the function WDOG_DRV_Deinit(). Make sure that the operation before WDOG timeout executing.

```
/* De-initialize WDOG module */
WDOG_DRV_Deinit(INST_WDOG1);
```

Example:

```
#define INST_WDOG1 (0U)

wdog_user_config_t userConfigPtr = {
    .WDOG_LPO_CLOCK,          /* Use the LPO clock as source */
    .opMode = {               /* WDOG not functional in Wait/Debug/Stop mode */
        false,
        false,
        false
    },
    .true,                    /* Enable further updates of the WDOG configuration */
    .false,                   /* Timeout interrupt disabled */
    .false,                   /* Window mode disabled */
    .0U,                      /* Window value */
    .0x400,                   /* Timeout value */
    .false                    /* Prescaler disabled */
};

/* Initialize WDOG module */
WDOG_DRV_Init(INST_WDOG1, &userConfigPtr);

/* Enable the timeout interrupt and set the ISR */
WDOG_DRV_SetInt(INST_WDOG1, true);

while (1) {

    /* Do something that takes between 0x100 and 0x400 clock cycles */

    /* Refresh the counter */
    WDOG_DRV_Trigger(INST_WDOG1);
}

/* De-initialize WDOG module */
WDOG_DRV_Deinit(INST_WDOG1);
```

Data Structures

-   struct wdog_op_mode_t

    WDOG option mode configuration structure Implements : wdog_op_mode_t_Class. More...
-   struct wdog_user_config_t

    WDOG user configuration structure Implements : wdog_user_config_t_Class. More...

Enumerations

-   enum wdog_clk_source_t { WDOG_BUS_CLOCK = 0x00U, WDOG_LPO_CLOCK = 0x01U, WDOG_SOS←C_CLOCK = 0x02U, WDOG_SIRC_CLOCK = 0x03U }

    Clock sources for the WDOG. Implements : wdog_clk_source_t_Class.
-   enum wdog_test_mode_t { WDOG_TST_DISABLED = 0x00U, WDOG_TST_USER = 0x01U, WDOG_TS←T_LOW = 0x02U, WDOG_TST_HIGH = 0x03U }

Test modes for the WDOG. Implements : wdog_test_mode_t_Class.

- enum wdog_set_mode_t { WDOG_DEBUG_MODE = 0x00U, WDOG_WAIT_MODE = 0x01U, WDOG_ST↩
OP_MODE = 0x02U }

    set modes for the WDOG. Implements : wdog_set_mode_t_Class

WDOG Driver API

- status_t WDOG_DRV_Init (uint32_t instance, const wdog_user_config_t ∗userConfigPtr)

    Initializes the WDOG driver.
- void WDOG_DRV_Deinit (uint32_t instance)

    De-initializes the WDOG driver.
- void WDOG_DRV_GetConfig (uint32_t instance, wdog_user_config_t ∗const config)

    Gets the current configuration of the WDOG.
- void WDOG_DRV_GetDefaultConfig (wdog_user_config_t ∗const config)

    Gets default configuration of the WDOG.
- status_t WDOG_DRV_SetInt (uint32_t instance, bool enable)

    Enables/Disables the WDOG timeout interrupt and sets a function to be called when a timeout interrupt is received, before reset.
- void WDOG_DRV_ClearIntFlag (uint32_t instance)

    Clear interrupt flag of the WDOG.
- void WDOG_DRV_Trigger (uint32_t instance)

    Refreshes the WDOG counter.
- uint16_t WDOG_DRV_GetCounter (uint32_t instance)

    Gets the value of the WDOG counter.
- void WDOG_DRV_SetWindow (uint32_t instance, bool enable, uint16_t windowvalue)

    Set window mode and window value of the WDOG.
- void WDOG_DRV_SetMode (uint32_t instance, bool enable, wdog_set_mode_t Setmode)

    Sets the mode operation of the WDOG.
- void WDOG_DRV_SetTimeout (uint32_t instance, uint16_t timeout)

    Sets the value of the WDOG timeout.
- void WDOG_DRV_SetTestMode (uint32_t instance, wdog_test_mode_t testMode)

    Changes the WDOG test mode.
- wdog_test_mode_t WDOG_DRV_GetTestMode (uint32_t instance)

    Gets the WDOG test mode.

16.90.2 Data Structure Documentation

16.90.2.1 struct wdog_op_mode_t

WDOG option mode configuration structure Implements : wdog_op_mode_t_Class.

Definition at line 87 of file wdog_driver.h.

Data Fields

- bool wait
- bool stop
- bool debug

Field Documentation

16.90.2.1.1 bool debug

Debug mode

Definition at line 91 of file wdog_driver.h.

16.90.2.1.2   bool stop

Stop mode

Definition at line 90 of file wdog_driver.h.

16.90.2.1.3   bool wait

Wait mode

Definition at line 89 of file wdog_driver.h.

16.90.2.2   struct wdog_user_config_t

WDOG user configuration structure Implements : wdog_user_config_t_Class.

Definition at line 98 of file wdog_driver.h.

Data Fields

- wdog_clk_source_t clkSource
- wdog_op_mode_t opMode
- bool updateEnable
- bool intEnable
- bool winEnable
- uint16_t windowValue
- uint16_t timeoutValue
- bool prescalerEnable

Field Documentation

16.90.2.2.1   wdog_clk_source_t clkSource

The clock source of the WDOG

Definition at line 100 of file wdog_driver.h.

16.90.2.2.2   bool intEnable

If true, an interrupt request is generated before reset

Definition at line 103 of file wdog_driver.h.

16.90.2.2.3   wdog_op_mode_t opMode

The modes in which the WDOG is functional

Definition at line 101 of file wdog_driver.h.

16.90.2.2.4   bool prescalerEnable

If true, a fixed 256 prescaling of the counter reference clock is enabled

Definition at line 107 of file wdog_driver.h.

16.90.2.2.5   uint16_t timeoutValue

The timeout value

Definition at line 106 of file wdog_driver.h.

16.90.2.2.6   bool updateEnable

If true, further updates of the WDOG are enabled

Definition at line 102 of file wdog_driver.h.

16.90.2.2.7   uint16_t windowValue

The window value

Definition at line 105 of file wdog_driver.h.

16.90.2.2.8   bool winEnable

If true, window mode is enabled

Definition at line 104 of file wdog_driver.h.

16.90.3   Enumeration Type Documentation

16.90.3.1   enum wdog_clk_source_t

Clock sources for the WDOG. Implements : wdog_clk_source_t_Class.

Enumerator

> ***WDOG_BUS_CLOCK***   Bus clock
>
> ***WDOG_LPO_CLOCK***   LPO clock
>
> ***WDOG_SOSC_CLOCK***   SOSC clock
>
> ***WDOG_SIRC_CLOCK***   SIRC clock

Definition at line 52 of file wdog_driver.h.

16.90.3.2   enum wdog_set_mode_t

set modes for the WDOG. Implements : wdog_set_mode_t_Class

Enumerator

> ***WDOG_DEBUG_MODE***   Debug mode
>
> ***WDOG_WAIT_MODE***   Wait mode
>
> ***WDOG_STOP_MODE***   Stop mode

Definition at line 76 of file wdog_driver.h.

16.90.3.3   enum wdog_test_mode_t

Test modes for the WDOG. Implements : wdog_test_mode_t_Class.

Enumerator

> ***WDOG_TST_DISABLED***   Test mode disabled
>
> ***WDOG_TST_USER***   User mode enabled. (Test mode disabled.)
>
> ***WDOG_TST_LOW***   Test mode enabled, only the low byte is used.
>
> ***WDOG_TST_HIGH***   Test mode enabled, only the high byte is used.

Definition at line 64 of file wdog_driver.h.

16.90.4   Function Documentation

16.90.4.1   void WDOG_DRV_ClearIntFlag (uint32_t *instance*)

Clear interrupt flag of the WDOG.

Parameters

in	*instance*	WDOG peripheral instance number

Definition at line 244 of file wdog_driver.c.

16.90.4.2   void WDOG_DRV_Deinit (uint32_t *instance*)

De-initializes the WDOG driver.

Parameters

in	*instance*	WDOG peripheral instance number

Definition at line 143 of file wdog_driver.c.

16.90.4.3   void WDOG_DRV_GetConfig (uint32_t *instance,* wdog_user_config_t ∗const *config*)

Gets the current configuration of the WDOG.

Parameters

in	*instance*	WDOG peripheral instance number
out	*configures*	the current configuration

Definition at line 167 of file wdog_driver.c.

16.90.4.4   uint16_t WDOG_DRV_GetCounter (uint32_t *instance*)

Gets the value of the WDOG counter.

Parameters

in	*instance*	WDOG peripheral instance number.

Returns

> the value of the WDOG counter.

Definition at line 275 of file wdog_driver.c.

16.90.4.5   void WDOG_DRV_GetDefaultConfig (wdog_user_config_t ∗const *config*)

Gets default configuration of the WDOG.

Parameters

out	*configures*	the default configuration

Definition at line 184 of file wdog_driver.c.

16.90.4.6   wdog_test_mode_t WDOG_DRV_GetTestMode (uint32_t *instance*)

Gets the WDOG test mode.

This function verifies the test mode of the WDOG.

Parameters

in	*instance*	WDOG peripheral instance number

Returns

> Test modes for the WDOG

Definition at line 389 of file wdog_driver.c.

16.90.4.7 status_t WDOG_DRV_Init (uint32_t *instance,* const wdog_user_config_t ∗ *userConfigPtr*)

Initializes the WDOG driver.

Parameters

in	*instance*	WDOG peripheral instance number
in	*userConfigPtr*	pointer to the WDOG user configuration structure

Returns

> operation status
>
> - STATUS_SUCCESS: Operation was successful.
>
> - STATUS_ERROR: Operation failed. Possible causes: previous clock source or the one specified in the configuration structure is disabled; WDOG configuration updates are not allowed.

Definition at line 105 of file wdog_driver.c.

16.90.4.8   status_t WDOG_DRV_SetInt (uint32_t *instance,* bool *enable*)

Enables/Disables the WDOG timeout interrupt and sets a function to be called when a timeout interrupt is received, before reset.

Parameters

in	*instance*	WDOG peripheral instance number
in	*enable*	enable/disable interrupt

Returns

> operation status
>
> - STATUS_SUCCESS: Operation was successful.
>
> - STATUS_ERROR: Operation failed. Possible causes: failed to WDOG configuration updates not allowed.

Definition at line 210 of file wdog_driver.c.

16.90.4.9   void WDOG_DRV_SetMode (uint32_t *instance,* bool *enable,* wdog_set_mode_t *Setmode*)

Sets the mode operation of the WDOG.

This function changes the mode operation of the WDOG.

Parameters

in	*instance*	WDOG peripheral instance number.
in	*enable*	enable/disable mode of the WDOG.
in	*Setmode*	select mode of the WDOG.

Definition at line 315 of file wdog_driver.c.

16.90.4.10   void WDOG_DRV_SetTestMode (uint32_t *instance,* wdog_test_mode_t *testMode*)

Changes the WDOG test mode.

This function changes the test mode of the WDOG. If the WDOG is tested in mode, software should set this field to 0x01U in order to indicate that the WDOG is functioning normally.

Parameters

in	*instance*	WDOG peripheral instance number

in	*testMode*	Test modes for the WDOG.

Definition at line 367 of file wdog_driver.c.

16.90.4.11 void WDOG_DRV_SetTimeout (uint32_t *instance,* uint16_t *timeout*)

Sets the value of the WDOG timeout.

This function sets the value of the WDOG timeout.

Parameters

in	*instance*	WDOG peripheral instance number.
in	*timeout*	the value of the WDOG timeout.

Definition at line 349 of file wdog_driver.c.

16.90.4.12 void WDOG_DRV_SetWindow (uint32_t *instance,* bool *enable,* uint16_t *windowvalue*)

Set window mode and window value of the WDOG.

This function set window mode, window value is set when window mode enabled.

Parameters

in	*instance*	WDOG peripheral instance number.
in	*enable*	enable/disable window mode and window value.
in	*windowvalue*	the value of the WDOG window.

Definition at line 290 of file wdog_driver.c.

16.90.4.13 void WDOG_DRV_Trigger (uint32_t *instance*)

Refreshes the WDOG counter.

Parameters

in	*instance*	WDOG peripheral instance number

Definition at line 260 of file wdog_driver.c.

16.91   Watchdog timer (WDOG)

16.91.1   Detailed Description

The S32 SDK provides Peripheral Driver for the Watchdog timer (WDOG) module of S32 SDK devices.
.

Hardware background

The Watchdog Timer (WDOG) module is an independent timer that is available for system use. It provides a safety feature to ensure that software is executing as planned and that the CPU is not stuck in an infinite loop or executing unintended code. If the WDOG module is not serviced (refreshed) within a certain period, it resets the MCU.

Features of the WDOG module include:

- Configurable clock source inputs independent from the bus clock

- Programmable timeout period

  - Programmable 16-bit timeout value
  - Optional fixed 256 clock prescaler when longer timeout periods are needed

- Window mode option for the refresh mechanism

  - Programmable 16-bit window value
  - Provides robust check that program flow is faster than expected
  - Early refresh attempts trigger a reset

- Optional timeout interrupt to allow post-processing diagnostics

  - Interrupt request to CPU with interrupt vector for an interrupt service routine (ISR)
  - Forced reset occurs 128 bus clocks after the interrupt vector fetch

- Configuration bits are write-once-after-reset to ensure watchdog configuration cannot be mistakenly altered

- Robust write sequence for unlocking write-once configuration bits

Modules

- WDOG Driver

    Watchdog Timer Peripheral Driver.

17 Data Structure Documentation

17.1 adc_callback_info_t Struct Reference

`#include <platform/devices/callbacks.h>`

Data Fields

- uint32_t groupIndex
- uint16_t resultBufferTail

17.1.1 Detailed Description

Definition at line 112 of file callbacks.h.

17.1.2 Field Documentation

17.1.2.1 uint32_t groupIndex

Index of the group executing the callback.

Definition at line 114 of file callbacks.h.

17.1.2.2 uint16_t resultBufferTail

Offset of the most recent conversion result in the result buffer.

Definition at line 115 of file callbacks.h.

The documentation for this struct was generated from the following file:

- platform/devices/callbacks.h

17.2 clock_manager_user_config_t Struct Reference

Clock configuration structure. Implements clock_manager_user_config_t_Class.

`#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>`

Data Fields

- scg_config_t scgConfig
- sim_clock_config_t simConfig
- pcc_config_t pccConfig
- pmc_config_t pmcConfig

17.2.1 Detailed Description

Clock configuration structure. Implements clock_manager_user_config_t_Class.

Definition at line 673 of file clock_S32K1xx.h.

17.2.2 Field Documentation

17.2.2.1 pcc_config_t pccConfig

PCC Clock configuration.

Definition at line 677 of file clock_S32K1xx.h.

17.2.2.2 pmc_config_t pmcConfig

PMC Clock configuration.

Definition at line 678 of file clock_S32K1xx.h.

17.2.2.3 scg_config_t scgConfig

SCG Clock configuration.

Definition at line 675 of file clock_S32K1xx.h.

17.2.2.4 sim_clock_config_t simConfig

SIM Clock configuration.

Definition at line 676 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.3 clock_source_config_t Struct Reference

Clock source configuration. Implements clock_source_config_t_Class.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- uint32_t refFreq
- xosc_ref_t refClk
- clk_freq_range_t clkSrcFreq
- pwr_modes_t modes

17.3.1 Detailed Description

Clock source configuration. Implements clock_source_config_t_Class.

Definition at line 755 of file clock_S32K1xx.h.

17.3.2 Field Documentation

17.3.2.1 clk_freq_range_t clkSrcFreq

Output frequency of the clock source (disabled, lowest, medium, highest)

Definition at line 759 of file clock_S32K1xx.h.

17.3.2.2 pwr_modes_t modes

Modes in which clock source is enabled

Definition at line 760 of file clock_S32K1xx.h.

17.3.2.3 xosc_ref_t refClk

Bypass option. It applies to external oscillator clock sources

Definition at line 758 of file clock_S32K1xx.h.

17.3.2.4 uint32_t refFreq

Frequency of the input reference clock. It applies to external oscillator clock sources

Definition at line 757 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.4 drv_config_t Struct Reference

Data Fields

- sbc_wtdog_ctr_t watchdogCtr
- uint32_t lpspiIntace
- bool isInit

17.4.1 Detailed Description

Definition at line 61 of file sbc_uja1169_driver.c.

17.4.2 Field Documentation

17.4.2.1 bool isInit

Definition at line 64 of file sbc_uja1169_driver.c.

17.4.2.2 uint32_t lpspiIntace

Definition at line 63 of file sbc_uja1169_driver.c.

17.4.2.3 sbc_wtdog_ctr_t watchdogCtr

Definition at line 62 of file sbc_uja1169_driver.c.

The documentation for this struct was generated from the following file:

- middleware/sbc/sbc_uja1169/source/sbc_uja1169_driver.c

17.5 lin_product_id_t Struct Reference

Product id structure Implements : lin_product_id_t_Class.

```
#include <middleware/lin/include/lin_types.h>
```

Data Fields

- l_u16 supplier_id
- l_u16 function_id
- l_u8 variant

17.5.1   Detailed Description

Product id structure Implements : lin_product_id_t_Class.

Definition at line 57 of file lin_types.h.

17.5.2   Field Documentation

17.5.2.1   l_u16 function_id

Function ID

Definition at line 60 of file lin_types.h.

17.5.2.2   l_u16 supplier_id

Supplier ID

Definition at line 59 of file lin_types.h.

17.5.2.3   l_u8 variant

Variant value

Definition at line 61 of file lin_types.h.

The documentation for this struct was generated from the following file:

- middleware/lin/include/lin_types.h

17.6   module_clk_config_t Struct Reference

module clock configuration. Implements module_clk_config_t_Class

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- clk_src_type_t clkSrcType
- clk_freq_range_t clkSrcFreq

17.6.1   Detailed Description

module clock configuration. Implements module_clk_config_t_Class

Definition at line 734 of file clock_S32K1xx.h.

17.6.2   Field Documentation

17.6.2.1   clk_freq_range_t clkSrcFreq

Frequency of the input peripheral clock source (disabled, lowest, medium, highest)

Definition at line 737 of file clock_S32K1xx.h.

17.6.2.2   clk_src_type_t clkSrcType

Type of the input peripheral clock source.

Definition at line 736 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.7 pcc_config_t Struct Reference

PCC configuration. Implements pcc_config_t_Class.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- uint32_t count
- peripheral_clock_config_t ∗ peripheralClocks

17.7.1 Detailed Description

PCC configuration. Implements pcc_config_t_Class.

Definition at line 647 of file clock_S32K1xx.h.

17.7.2 Field Documentation

17.7.2.1 uint32_t count

Number of peripherals to be configured.

Definition at line 649 of file clock_S32K1xx.h.

17.7.2.2 peripheral_clock_config_t∗ peripheralClocks

Pointer to the peripheral clock configurations array.

Definition at line 650 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.8 peripheral_clock_config_t Struct Reference

PCC peripheral instance clock configuration. Implements peripheral_clock_config_t_Class.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- clock_names_t clockName
- bool clkGate
- peripheral_clock_source_t clkSrc
- peripheral_clock_frac_t frac
- peripheral_clock_divider_t divider

17.8.1 Detailed Description

PCC peripheral instance clock configuration. Implements peripheral_clock_config_t_Class.

Definition at line 628 of file clock_S32K1xx.h.

17.8.2   Field Documentation

17.8.2.1   bool clkGate

Peripheral clock gate.

Definition at line 638 of file clock_S32K1xx.h.

17.8.2.2   peripheral_clock_source_t clkSrc

Peripheral clock source.

Definition at line 639 of file clock_S32K1xx.h.

17.8.2.3   clock_names_t clockName

Definition at line 637 of file clock_S32K1xx.h.

17.8.2.4   peripheral_clock_divider_t divider

Peripheral clock divider value.

Definition at line 641 of file clock_S32K1xx.h.

17.8.2.5   peripheral_clock_frac_t frac

Peripheral clock fractional value.

Definition at line 640 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.9   pmc_config_t Struct Reference

PMC configure structure.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- pmc_lpo_clock_config_t lpoClockConfig

17.9.1   Detailed Description

PMC configure structure.

Definition at line 664 of file clock_S32K1xx.h.

17.9.2   Field Documentation

17.9.2.1   **pmc_lpo_clock_config_t** lpoClockConfig

Low Power Clock configuration.

Definition at line 666 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.10 pmc_lpo_clock_config_t Struct Reference

PMC LPO configuration.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- bool initialize
- bool enable
- int8_t trimValue

17.10.1 Detailed Description

PMC LPO configuration.

Definition at line 654 of file clock_S32K1xx.h.

17.10.2 Field Documentation

17.10.2.1 bool enable

Enable/disable LPO

Definition at line 657 of file clock_S32K1xx.h.

17.10.2.2 bool initialize

Initialize or not the PMC LPO settings.

Definition at line 656 of file clock_S32K1xx.h.

17.10.2.3 int8_t trimValue

LPO trimming value

Definition at line 658 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.11 scg_clock_mode_config_t Struct Reference

SCG Clock Mode Configuration structure. Implements scg_clock_mode_config_t_Class.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- scg_system_clock_config_t rccrConfig
- scg_system_clock_config_t vccrConfig
- scg_system_clock_config_t hccrConfig
- scg_system_clock_src_t alternateClock
- bool initialize

17.11.1   Detailed Description

SCG Clock Mode Configuration structure. Implements scg_clock_mode_config_t_Class.

Definition at line 548 of file clock_S32K1xx.h.

17.11.2   Field Documentation

17.11.2.1   scg_system_clock_src_t alternateClock

Alternate clock used during initialization

Definition at line 553 of file clock_S32K1xx.h.

17.11.2.2   scg_system_clock_config_t hccrConfig

HSRUN Clock Control configuration.

Definition at line 552 of file clock_S32K1xx.h.

17.11.2.3   bool initialize

Initialize or not the Clock Mode Configuration.

Definition at line 554 of file clock_S32K1xx.h.

17.11.2.4   scg_system_clock_config_t rccrConfig

Run Clock Control configuration.

Definition at line 550 of file clock_S32K1xx.h.

17.11.2.5   scg_system_clock_config_t vccrConfig

VLPR Clock Control configuration.

Definition at line 551 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.12   scg_clockout_config_t Struct Reference

SCG ClockOut Configuration structure. Implements scg_clockout_config_t_Class.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- scg_clockout_src_t source
- bool initialize

17.12.1   Detailed Description

SCG ClockOut Configuration structure. Implements scg_clockout_config_t_Class.

Definition at line 561 of file clock_S32K1xx.h.

17.12.2   Field Documentation

17.12.2.1   bool initialize

Initialize or not the ClockOut.

Definition at line 564 of file clock_S32K1xx.h.

17.12.2.2   scg_clockout_src_t source

ClockOut source select.

Definition at line 563 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.13   scg_config_t Struct Reference

SCG configure structure. Implements scg_config_t_Class.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- scg_sirc_config_t sircConfig
- scg_firc_config_t fircConfig
- scg_sosc_config_t soscConfig
- scg_spll_config_t spllConfig
- scg_rtc_config_t rtcConfig
- scg_clockout_config_t clockOutConfig
- scg_clock_mode_config_t clockModeConfig

17.13.1   Detailed Description

SCG configure structure. Implements scg_config_t_Class.

Definition at line 571 of file clock_S32K1xx.h.

17.13.2   Field Documentation

17.13.2.1   scg_clock_mode_config_t clockModeConfig

SCG Clock Mode Configuration.

Definition at line 579 of file clock_S32K1xx.h.

17.13.2.2   scg_clockout_config_t clockOutConfig

SCG ClockOut Configuration.

Definition at line 578 of file clock_S32K1xx.h.

17.13.2.3   scg_firc_config_t fircConfig

Fast internal reference clock configuration.

Definition at line 574 of file clock_S32K1xx.h.

17.13.2.4   scg_rtc_config_t rtcConfig

Real Time Clock configuration.

Definition at line 577 of file clock_S32K1xx.h.

17.13.2.5   scg_sirc_config_t sircConfig

Slow internal reference clock configuration.

Definition at line 573 of file clock_S32K1xx.h.

17.13.2.6   scg_sosc_config_t soscConfig

System oscillator configuration.

Definition at line 575 of file clock_S32K1xx.h.

17.13.2.7   scg_spll_config_t spllConfig

System Phase locked loop configuration.

Definition at line 576 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.14   scg_firc_config_t Struct Reference

SCG fast IRC clock configuration. Implements scg_firc_config_t_Class.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- scg_firc_range_t range
- scg_async_clock_div_t div1
- scg_async_clock_div_t div2
- bool enableInStop
- bool enableInLowPower
- bool regulator
- bool locked
- bool initialize

17.14.1   Detailed Description

SCG fast IRC clock configuration. Implements scg_firc_config_t_Class.

Definition at line 429 of file clock_S32K1xx.h.

17.14.2   Field Documentation

17.14.2.1   scg_async_clock_div_t div1

Divider for platform asynchronous clock.

Definition at line 433 of file clock_S32K1xx.h.

17.14.2.2   scg_async_clock_div_t div2

Divider for bus asynchronous clock.

Definition at line 434 of file clock_S32K1xx.h.

17.14.2.3   bool enableInLowPower

FIRC is enable or not in lowpower mode.

Definition at line 437 of file clock_S32K1xx.h.

17.14.2.4   bool enableInStop

FIRC is enable or not in stop mode.

Definition at line 436 of file clock_S32K1xx.h.

17.14.2.5   bool initialize

Initialize or not the FIRC module.

Definition at line 441 of file clock_S32K1xx.h.

17.14.2.6   bool locked

FIRC Control Register can be written.

Definition at line 439 of file clock_S32K1xx.h.

17.14.2.7   scg_firc_range_t range

Fast IRC frequency range.

Definition at line 431 of file clock_S32K1xx.h.

17.14.2.8   bool regulator

FIRC regulator is enable or not.

Definition at line 438 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.15   scg_rtc_config_t Struct Reference

SCG RTC configuration. Implements scg_rtc_config_t_Class.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- uint32_t rtcClkInFreq
- bool initialize

17.15.1   Detailed Description

SCG RTC configuration. Implements scg_rtc_config_t_Class.

Definition at line 538 of file clock_S32K1xx.h.

17.15.2   Field Documentation

17.15.2.1   bool initialize

Initialize or not the RTC.

Definition at line 541 of file clock_S32K1xx.h.

17.15.2.2   uint32_t rtcClkInFreq

RTC_CLKIN frequency.

Definition at line 540 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.16   scg_sirc_config_t Struct Reference

SCG slow IRC clock configuration. Implements scg_sirc_config_t_Class.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- scg_sirc_range_t range
- scg_async_clock_div_t div1
- scg_async_clock_div_t div2
- bool initialize
- bool enableInStop
- bool enableInLowPower
- bool locked

17.16.1   Detailed Description

SCG slow IRC clock configuration. Implements scg_sirc_config_t_Class.

Definition at line 402 of file clock_S32K1xx.h.

17.16.2   Field Documentation

17.16.2.1   scg_async_clock_div_t div1

Divider for platform asynchronous clock.

Definition at line 406 of file clock_S32K1xx.h.

17.16.2.2   scg_async_clock_div_t div2

Divider for bus asynchronous clock.

Definition at line 407 of file clock_S32K1xx.h.

17.16.2.3   bool enableInLowPower

SIRC is enable or not in low power mode.

Definition at line 411 of file clock_S32K1xx.h.

17.16.2.4   bool enableInStop

SIRC is enable or not in stop mode.

Definition at line 410 of file clock_S32K1xx.h.

17.16.2.5   bool initialize

Initialize or not the SIRC module.

Definition at line 409 of file clock_S32K1xx.h.

17.16.2.6   bool locked

SIRC Control Register can be written.

Definition at line 413 of file clock_S32K1xx.h.

17.16.2.7   scg_sirc_range_t range

Slow IRC frequency range.

Definition at line 404 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.17   scg_sosc_config_t Struct Reference

SCG system OSC configuration. Implements scg_sosc_config_t_Class.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- uint32_t freq
- scg_sosc_monitor_mode_t monitorMode
- scg_sosc_ext_ref_t extRef
- scg_sosc_gain_t gain
- scg_sosc_range_t range
- scg_async_clock_div_t div1
- scg_async_clock_div_t div2
- bool enableInStop
- bool enableInLowPower
- bool locked
- bool initialize

17.17.1   Detailed Description

SCG system OSC configuration. Implements scg_sosc_config_t_Class.

Definition at line 367 of file clock_S32K1xx.h.

17.17.2   Field Documentation

17.17.2.1   scg_async_clock_div_t div1

Divider for platform asynchronous clock.

Definition at line 378 of file clock_S32K1xx.h.

17.17.2.2   scg_async_clock_div_t div2

Divider for bus asynchronous clock.

Definition at line 379 of file clock_S32K1xx.h.

17.17.2.3   bool enableInLowPower

System OSC is enable or not in low power mode.

Definition at line 382 of file clock_S32K1xx.h.

17.17.2.4   bool enableInStop

System OSC is enable or not in stop mode.

Definition at line 381 of file clock_S32K1xx.h.

17.17.2.5   scg_sosc_ext_ref_t extRef

System OSC External Reference Select.

Definition at line 373 of file clock_S32K1xx.h.

17.17.2.6   uint32_t freq

System OSC frequency.

Definition at line 369 of file clock_S32K1xx.h.

17.17.2.7   scg_sosc_gain_t gain

System OSC high-gain operation.

Definition at line 374 of file clock_S32K1xx.h.

17.17.2.8   bool initialize

Initialize or not the System OSC module.

Definition at line 386 of file clock_S32K1xx.h.

17.17.2.9   bool locked

System OSC Control Register can be written.

Definition at line 384 of file clock_S32K1xx.h.

17.17.2.10   scg_sosc_monitor_mode_t monitorMode

System OSC Clock monitor mode.

Definition at line 371 of file clock_S32K1xx.h.

17.17.2.11   scg_sosc_range_t range

System OSC frequency range.

Definition at line 376 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.18 scg_spll_config_t Struct Reference

SCG system PLL configuration. Implements scg_spll_config_t_Class.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- scg_spll_monitor_mode_t monitorMode
- uint8_t prediv
- uint8_t mult
- uint8_t src
- scg_async_clock_div_t div1
- scg_async_clock_div_t div2
- bool enableInStop
- bool locked
- bool initialize

17.18.1 Detailed Description

SCG system PLL configuration. Implements scg_spll_config_t_Class.

Definition at line 517 of file clock_S32K1xx.h.

17.18.2 Field Documentation

17.18.2.1 scg_async_clock_div_t div1

Divider for platform asynchronous clock.

Definition at line 525 of file clock_S32K1xx.h.

17.18.2.2 scg_async_clock_div_t div2

Divider for bus asynchronous clock.

Definition at line 526 of file clock_S32K1xx.h.

17.18.2.3 bool enableInStop

System PLL clock is enable or not in stop mode.

Definition at line 528 of file clock_S32K1xx.h.

17.18.2.4 bool initialize

Initialize or not the System PLL module.

Definition at line 531 of file clock_S32K1xx.h.

17.18.2.5 bool locked

System PLL Control Register can be written.

Definition at line 530 of file clock_S32K1xx.h.

17.18.2.6 scg_spll_monitor_mode_t monitorMode

Clock monitor mode selected.

Definition at line 519 of file clock_S32K1xx.h.

17.18.2.7   uint8_t mult

System PLL multiplier.

Definition at line 522 of file clock_S32K1xx.h.

17.18.2.8   uint8_t prediv

PLL reference clock divider.

Definition at line 521 of file clock_S32K1xx.h.

17.18.2.9   uint8_t src

System PLL source.

Definition at line 523 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

17.19   sys_clk_config_t Struct Reference

System clock configuration. Implements sys_clk_config_t_Class.

```
#include <platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h>
```

Data Fields

- clock_names_t src
- uint16_t dividers [MAX_NO_SYS_CLOCK_DIVS]

17.19.1   Detailed Description

System clock configuration. Implements sys_clk_config_t_Class.

Definition at line 745 of file clock_S32K1xx.h.

17.19.2   Field Documentation

17.19.2.1   uint16_t dividers[MAX_NO_SYS_CLOCK_DIVS]

System clock dividers. Value by which system clock is divided. 0 means that system clock is not divided.

Definition at line 748 of file clock_S32K1xx.h.

17.19.2.2   clock_names_t src

System clock source.

Definition at line 747 of file clock_S32K1xx.h.

The documentation for this struct was generated from the following file:

- platform/drivers/src/clock/S32K1xx/clock_S32K1xx.h

Index