JBLopen

Embedded Software Insight

# BASEplatform Zynq-7000 Reference Manual

PM0001

July 25, 2019

# Contents

Chapter

1

# Overview

Welcome to the BASEplatform™ platform reference manual for the Xilinx Zynq-7000®. This reference manual covers the platform-specific API relevant to the Zynq-7000 as well as the Xilinx FPGA based IPs. This document includes important configuration information as well as the complete API reference for the Zynq-7000. The core API of the BASEplatform can be found in the BASEplatform API reference available on the documentation section of the JBLopen website. Similarly to the core API, the platform-specific API is written in ISO/IEC 9899:1999 (C99) compliant C and designed to be portable across the toolchains supporting the Zynq-7000.

For convenience during development, all the information related to each individual API element is also reproduced within the relevant header source files in human readable format.

## About the BASEplatform

The BASEplatform is a collection of low-level interface modules, drivers and board support packages (BSPs) designed to provide the foundation for an embedded software application. The BASEplatform can support a variety of free or commercial RTOSes as well as bare-metal applications, both in multi-core and single core configurations. BASEplatform packages are created specifically for an application's needs, and usually include support for an RTOS or bare-metal, low level I/Os, such as UART, I2C, GPIO etc. as well as communication and storage stacks, as selected by the application developer, alongside the necessary drivers, integration and IDE files to get everything working out of the box.

# Elements of the API Reference

Each documented API element, be it a function, data type or preprocessor definition is presented using a similar layout which are described below. This section briefly describes the various elements of the API reference.

## Functions

The most numerous and important API elements documented are functions. Below is an example of API reference for a hypothetical function named `bp_example_func()`:

Function

### `bp_example_func()`

<example/bp_example.h>

Example function description.

*Prototype*

```
int  bp_example_func  ( uint32_t  arg1,
                        uint32_t  arg2 );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `arg1` | First argument's description. |
| `arg2` | Second argument's description. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

*Example*

```
uint32_t a = 0u;
uint32_t b = 1u;
int result;

result = bp_example_func(a, b);
if(result != RTNC_SUCCESS) \{
    // Handle error.
\}
```

### Function Name

At the top of each API is the name of the function or object as it appears in the source code. BASEplatform functions are always prefixed with `bp_` followed by the module name and then the function's specific name.

### Header

Following the name is the header file where the declaration of the documented API can be found. It is recommended to use the displayed path relative to the root of the source directory of the BASEplatform when including BASEplatform's headers.

For example, to include the UART module header file `bp_uart.h` the following include directive is recommended.

```
#include <uart/bp_uart.h>
```

The root of the BASEplatform source directory should be added to the include path of the compiler.

### Description

A description of the API element including basic usage information.

### Prototype

For functions, the full signature of the API along with parameter names, types, and function return type.

### Attributes

For functions only, this section lists the relevant function attributes. See the function attributes section of this manual for a detailed description of each attribute.

### Parameters

Function parameters list along with a short description of each parameter.

### Returned Errors or Return Values

For functions that return a BASEplatform standard error code, this section is named Returned Errors and lists the relevant errors that can be returned. See the error handling convention section of this manual for more information on the BASEplatform error handling.

For other functions that do not return a standard error code, this section lists the possible output values of the function. In this case the section is named "Returned Values".

### Example

Some API functions may include a small code example to illustrate usage. Note that these examples are for documentation purpose and may not include error handling and checking to keep the examples concise.

## Data Types

Data types include structure definitions, enumerated types as well as scalar type definitions. They all follow a similar documentation layout, below is an example of API reference for a hypothetical structure definition named `bp_example_struct_t`:

Data Type

### bp_example_struct_t

<example/bp_example.h>

Example structure description.

*Members*

| | | |
|---|---|---|
| member1 | uint32_t | First member's description. |
| member2 | uint32_t | Second member's description. |

## Data Type Name

At the top of each API is the name of the data type as it appears in the source code. In the case of structures and enumerated types, this is always the `typedef`'d data type. BASEplatform types always prefixed with `bp_` followed by the module name and then the type's specific name. Types are also always suffixed with `_t` to differentiate them from other definitions.

## Header

Following the name is the header file where the declaration of the documented API can be found. It is recommended to use the displayed path relative to the root of the source directory of the BASEplatform when including BASEplatform's headers.

## Description

A description of the data type including basic usage information.

## Members/Enumeration Values

If documented, the API reference will list the structure members or the list of enumeration constants along with a short description of each member. The list of members for opaque types with no publicly accessible members are omitted from the list of members in the API documentation.

# Macros

Relevant and preprocessor macros that are part of the public API are documented in the API reference. This includes function-like macros as well as object-like macros. The latter is often referred to as preprocessor definitions or simply defines. Below is an example of function-like macro named `BP_EXAMPLE_MACRO()`:

Macro

### BP_EXAMPLE_MACRO()

<example/bp_example.h>

Example macro description.

*Prototype*

```
BP_EXAMPLE_MACRO ( arg1,
                   arg2 );
```

*Parameters*

| | |
|---|---|
| arg1 | First argument's description. |
| arg2 | First argument's description. |

*Expansion* Macro expansion's description.

### Macro Name

At the top of each API is the name of the macro as it appears in the source code. BASEplatform preprocessor definitions are always in capital letters and prefixed with `BP_` followed by the module name and then the macro's specific name.

### Header

Following the name is the header file where the declaration of the documented API can be found. It is recommended to use the displayed path relative to the root of the source directory of the BASEplatform when including BASEplatform's headers.

### Description

A description of the macro including basic usage information.

### Parameters

Macro parameters list along with a short description of each parameter.

### Expansion

For function-like macros an expansion section describes the macro's expansion including the type if applicable.

# Function Attributes

The API reference documentation for API functions includes a set of attributes that clarifies in which context it is safe to call a specific API function. The attributes are as follows:

- Blocking
- ISR-safe
- Critical-safe
- Thread-safe

## Blocking

The function is potentially blocking, which means it can wait or pend on a kernel object such as a semaphore or mutex, in order to wait for a resource to be available or for an operation to complete. Some functions may be optionally blocking depending on the function's arguments. Those functions are always marked as blocking in the API reference regardless.

In a bare-metal environment, any function marked as blocking can potentially suspend the background task while waiting for a specific interrupt. Many of those functions take a timeout parameter that can be set to 0 to make them non-blocking (polling) if suspension of the background task is undesired.

As a general rule, blocking functions should not be called from an interrupt service routine, also known as interrupt handler or while the CPU interrupts are disabled. In addition, some RTOSes allow suspending or locking the scheduler, when this is the case, blocking functions should not be called while the scheduler is suspended or locked.

### ISR-Safe

An ISR-safe function can be called from within an interrupt service routine. This also includes callback functions that are called from an interrupt context. Note that while an ISR-safe function is usually critical-safe this is not always the case. Also an ISR-safe function may not necessarily be thread-safe.

### Critical Safe

Critical safe functions can be called when the CPU interrupts are disabled, this is also called a critical context or sometimes a critical section. Critical sections are usually entered by calling a spinlock acquire or critical section enter function. Calling a non-critical-safe function from within a critical section can corrupt the state of the CPU's interrupt disable flags and cause runtime faults or data corruption.

### Thread-Safe

A thread safe function guarantees correct operations between multiple threads or tasks when running under a multitasking kernel. In the context of the BASEplatform API, thread-safe also implies thread safety on an SMP system, which means it is safe to use the API function from different threads in parallel. Due to the design of the BASEplatform, thread-safe functions are also re-entrant assuming that the other function attributes, such as ISR safety, are respected.

### Function Attributes in Header Files

Function attributes are documented slightly differently in the source header files in order to be more concise and easier to maintain. The attributes are documented under an "Attributes" section and are named as follows:

- non-blocking
- non-thread-safe
- ISR-safe
- critical-section-safe

Absence of an attribute implies that the opposite attribute applies to the function. For example, in the absence of any explicit function attribute in the header documentation, a function is assumed to be blocking, thread-safe and not safe to call from ISRs and critical sections.

## API Conventions

The BASEplatform API adheres to a few conventions with respect to the naming, error handling and timeouts that are useful for the application developers.

## Naming

The BASEplatform API function names are all written in lower case, except preprocessor macros which are in upper case. Words within an object name are separated by underscores and the whole name is prefixed with bp_ followed by the module name and finally the function specific part of the name.

For example, the time module function to get the current time is written as follows:

```
bp_time_get()
```

And the memory barrier macro from the architecture module, ”arch” for short, is named as follows:

```
BP_ARCH_MB()
```

## Error Handling

Most API functions return a status in the form of a plain int as the function’s return value. As a general exception, some functions that cannot fail are allowed to return nothing (void) or another value.

In general, the BASEplatform attempts to minimize the number of different error codes to simplify the application’s error handling and improve performance. The list of possible error codes is included within every function’s documentation. The meaning of each error code is also documented in a function’s description. See the Error Codes chapter for a list of defined error codes.

As with other preprocessor macros and enumeration constants, the application should never rely on the exact numerical value of any specific error code. However, two guarantees are made with respect to the error code numerical values. The first is that `RTNC_SUCCESS` will always expand to 0. The second is that all other error codes are negative. Positive values are not used for any valid error code. Any undefined or unexpected error code returned by a function should be treated as a fatal error.

Two error codes have the exact same meaning for all the functions, namely `RTNC_SUCCESS` and `RTNC_FATAL`.

`RTNC_SUCCESS` is returned when a function completed successfully without issue.

`RTNC_FATAL` is returned if and only if an unexpected situation that should not happen at runtime is detected. This includes invalid function arguments, internal data corruption and assertion failures within the code. In addition, any unexpected error code returned from a function should be treated as a fatal error. It is up to the application to decide on the proper action to perform upon receiving a fatal error. As a general rule, the application should not perform any other calls to that module instance. Safety critical applications should consider an `RTNC_FATAL` error code as a severe assertion failure and act accordingly.

Some modules, especially IO modules such as UART and I2C, provides a reset API call that can be used to reset the internal state of a module as well as the underlying peripheral. This can be used to attempt to recover from a fatal error in case the error condition is temporary.

## Timeouts

Most of the blocking functions have a timeout argument that takes a timeout value in milliseconds. The timeout period is guaranteed to be at least the requested value rounded up to the next multiple of the kernel’s tick rate if necessary. Internally, the BASEplatform modules and drivers will attempt to respect the timeout value as closely as possible while guaranteeing the minimum timeout value. However, RTOS

scheduling, higher priority tasks and interrupt response time may increase the amount of time taken to return from a timeout condition.

For all functions that take a timeout value, specifying a timeout value of 0 means that the function will return immediately instead of blocking when having to wait on a mutex or an interrupt. A value of `TIMEOUT_INF` or -1 will result in an infinite timeout.

### Numerical Values of Macros and Enumeration Constants

To ease maintainability and ensure compatibility with future versions, the application should never rely on enumeration constants and macros numerical value.

## Driver API

Many of the BASEplatform modules, especially the IO modules, use drivers to perform hardware access. In those situations the top-level module provides lifecycle management as well as thread-safety. However, it may be desirable in some circumstances to access the driver API directly. The various driver function signatures are gathered at the end of this manual but additional details may be available from each platform's reference manual.

### Advanced Driver API

Each driver is allowed to implement additional, driver specific, functionalities not available from the top level module API. These functions are usually meant to control advanced features of the underlying peripherals. Each I/O module provides an API to retrieve the driver's handle which can be used to access those advanced functions directly. There is also an optional locking mechanism that can be used to ensure thread safety while performing direct operations on the drivers.

### Accessing the Drivers Directly

It is also possible to access the drivers standard operation directly at the driver level. This reduces the overhead associated the kernel mutexes and driver dereference at the cost of thread safety. As such, direct driver access should be done with care. As with the case of the advanced driver features, there is an optional exclusive lock mechanism that can be used to ensure thread safety.

## Zynq-7000 Specific Configurations

To accommodate the highly configurable nature of the Xilinx Zynq-7000, the BASEplatform includes additional configurations specific to the Zynq-7000.

### Configuration Files

To improve integration with the Xilinx SDK, the BASEplatform modules for the Zynq-7000 requires the inclusion of the `xparameters.h` header file generated by the SDK. To do so, an additional configuration header named `bp_xilinx_def_cfg.h` must be created and should contain a single include directive for `xparameters.h`.

For example:

```
#include </path/to/project_bsp/ps7_cortexa9_0/include/xparameters.h>;
```

## SoC Definition

The generic definitions for the Zynq-7000 SoC peripherals can be found in `soc/zynq/bp_zynq_soc_def.h`. These can be used to write a custom board definition.

## Generic Xilinx SoC and Board Definition

To help in writing custom board files that may contain various soft IPs, the BASEplatform can automatically include the required definitions from the Xilinx SDK. The SoC level definitions can be found in `soc/xilinx/bp_xilinx_soc_def.h` and the board definitions can be found in `board/xilinx/xilinx_generic/bp_xilinx_generic_board_def.h`.

For example, if the Vivado project contains a single AXI UARTLite IP instance `g_xil_axiuart0` will be automatically defined in `bp_xilinx_generic_board_def.h` and can be used as an argument to 'bp_uart_create()' to create a UART module instance as usual.

## External Clocks Frequency

Since there is no way for the BASEplatform to derive the input frequency, it must be specified in the board definition file. For example:

```
#define BOARD_DEF_ZYNQ_PS_CLK_FREQ (33333333U)
```

Chapter

# 2 Zynq-7000 Definitions

Zynq-7000 global definitions. This module contains various definitions pertaining to the Zynq-7000 including interrupts, resets and clocks lists as well as the base addresses of the various peripherals. These definitions are used in the SoC definition files for the Zynq but can also be used as input values for the clock, reset and interrupt management modules.

Data Type

## bp_zynq_clk_t

<soc/zynq/bp_zynq_def.h>

Zynq-7000 clock list. These enumeration constants can be used with the clock management API such as `bp_clk_freq_get()` and `bp_clock_en()`.

*Values*

| | |
|---|---|
| `BP_ZYNQ_CLK_PS_CLK` | PS input clock |
| `BP_ZYNQ_CLK_ARM_PLL` | ARM PLL |
| `BP_ZYNQ_CLK_DDR_PLL` | DDR PLL |
| `BP_ZYNQ_CLK_IO_PLL` | IO PLL |
| `BP_ZYNQ_CLK_CPU_6X4X` | CPU 6X4X |
| `BP_ZYNQ_CLK_CPU_3X2X` | CPU 3X2X |
| `BP_ZYNQ_CLK_CPU_2X` | CPU 2X |
| `BP_ZYNQ_CLK_CPU_1X` | CPU 1X |
| `BP_ZYNQ_CLK_DDR3X` | DDR 3X |
| `BP_ZYNQ_CLK_DDR2X` | DDR 2X |
| `BP_ZYNQ_CLK_DDR_DCI` | DDR DCI |

| | |
|---|---|
| `BP_ZYNQ_CLK_DMAC` | DMA controller |
| `BP_ZYNQ_CLK_USB0` | USB 0 |
| `BP_ZYNQ_CLK_USB1` | USB 1 |
| `BP_ZYNQ_CLK_GIGE0` | Ethernet 0 |
| `BP_ZYNQ_CLK_GIGE1` | Ethernet 1 |
| `BP_ZYNQ_CLK_SDIO0` | SDIO 0 |
| `BP_ZYNQ_CLK_SDIO1` | SDIO 1 |
| `BP_ZYNQ_CLK_SPI0` | SPI 0 |
| `BP_ZYNQ_CLK_SPI1` | SPI 1 |
| `BP_ZYNQ_CLK_CAN0` | CAN 0 |
| `BP_ZYNQ_CLK_CAN1` | CAN 1 |
| `BP_ZYNQ_CLK_I2C0` | I2C 0 |
| `BP_ZYNQ_CLK_I2C1` | I2C 1 |
| `BP_ZYNQ_CLK_UART0` | UART 0 |
| `BP_ZYNQ_CLK_UART1` | UART 1 |
| `BP_ZYNQ_CLK_GPIO` | GPIO |
| `BP_ZYNQ_CLK_QSPI` | QSPI |
| `BP_ZYNQ_CLK_SMC` | Static memory controller |
| `BP_ZYNQ_CLK_PCAP2X` | PCAP 2X |
| `BP_ZYNQ_CLK_TRACE` | TRACE |
| `BP_ZYNQ_CLK_FCLK0` | PL clock 0 |
| `BP_ZYNQ_CLK_FCLK1` | PL clock 1 |
| `BP_ZYNQ_CLK_FCLK2` | PL clock 2 |
| `BP_ZYNQ_CLK_FCLK3` | PL clock 3 |
| `BP_ZYNQ_CLK_NONE` | Special invalid clock value. |

Data Type

## `bp_zynq_int_t`

<soc/zynq/bp_zynq_def.h>

Zynq-7000 interrupt list. These enumeration constants can be used with the interrupt management API such as `bp_int_reg()` and `bp_int_src_en()` for convenience.

*Values*

| | |
|---|---|
| BP_ZYNQ_INT_TIMER | CPU MPCORE global timer. |
| BP_ZYNQ_INT_NFIQ | PL fast interrupt. |
| BP_ZYNQ_INT_PRIV_TIMER | CPU MPCORE private timer. |
| BP_ZYNQ_INT_PRIV_WDOG | CPU MPCORE private watchdog. |
| BP_ZYNQ_INT_NIRQ | PL interrupt. |
| BP_ZYNQ_INT_CPU0 | CPU0 |
| BP_ZYNQ_INT_CPU1 | CPU1 |
| BP_ZYNQ_INT_L2 | L2 Cache |
| BP_ZYNQ_INT_OCM | OCM |
| BP_ZYNQ_INT_PMU0 | PMU Core 0 |
| BP_ZYNQ_INT_PMU1 | PMU Core 1 |
| BP_ZYNQ_INT_XADC | XADC |
| BP_ZYNQ_INT_DEVC | DevC |
| BP_ZYNQ_INT_SWDT | Watchdog |
| BP_ZYNQ_INT_TTC0_0 | Triple Timer Counter 0 counter 0 |
| BP_ZYNQ_INT_TTC0_1 | Triple Timer Counter 0 counter 1 |
| BP_ZYNQ_INT_TTC0_2 | Triple Timer Counter 0 counter 2 |
| BP_ZYNQ_INT_DMAC_ABT | DMA controller abort |
| BP_ZYNQ_INT_DMAC0 | DMA controller IRQ 0 |
| BP_ZYNQ_INT_DMAC1 | DMA controller IRQ 1 |
| BP_ZYNQ_INT_DMAC2 | DMA controller IRQ 2 |
| BP_ZYNQ_INT_DMAC3 | DMA controller IRQ 3 |
| BP_ZYNQ_INT_SMC | Static memory controller |
| BP_ZYNQ_INT_QSPI | QSPI controller |
| BP_ZYNQ_INT_GPIO | GPIO |
| BP_ZYNQ_INT_USB0 | USB 0 |
| BP_ZYNQ_INT_ENET0 | Ethernet 0 |
| BP_ZYNQ_INT_ENET0_WU | Ethernet 0 wake-up |
| BP_ZYNQ_INT_SDIO0 | SDIO 0 |
| BP_ZYNQ_INT_I2C0 | I2C 0 |
| BP_ZYNQ_INT_SPI0 | SPI 0 |
| BP_ZYNQ_INT_UART0 | UART 0 |

| | |
|---|---|
| BP_ZYNQ_INT_CAN0 | CAN 0 |
| BP_ZYNQ_INT_PL0 | Programmable logic 0 |
| BP_ZYNQ_INT_PL1 | Programmable logic 1 |
| BP_ZYNQ_INT_PL2 | Programmable logic 2 |
| BP_ZYNQ_INT_PL3 | Programmable logic 3 |
| BP_ZYNQ_INT_PL4 | Programmable logic 4 |
| BP_ZYNQ_INT_PL5 | Programmable logic 5 |
| BP_ZYNQ_INT_PL6 | Programmable logic 6 |
| BP_ZYNQ_INT_PL7 | Programmable logic 7 |
| BP_ZYNQ_INT_TTC1_0 | Triple Timer Counter 1 counter 0 |
| BP_ZYNQ_INT_TTC1_1 | Triple Timer Counter 1 counter 1 |
| BP_ZYNQ_INT_TTC1_2 | Triple Timer Counter 1 counter 2 |
| BP_ZYNQ_INT_DMAC4 | DMA controller IRQ 4 |
| BP_ZYNQ_INT_DMAC5 | DMA controller IRQ 5 |
| BP_ZYNQ_INT_DMAC6 | DMA controller IRQ 6 |
| BP_ZYNQ_INT_DMAC7 | DMA controller IRQ 7 |
| BP_ZYNQ_INT_USB1 | USB 1 |
| BP_ZYNQ_INT_ENET1 | Ethernet 1 |
| BP_ZYNQ_INT_ENET1_WU | Ethernet 1 wake-up |
| BP_ZYNQ_INT_SDIO1 | SDIO 1 |
| BP_ZYNQ_INT_I2C1 | I2C 1 |
| BP_ZYNQ_INT_SPI1 | SPI 1 |
| BP_ZYNQ_INT_UART1 | UART 1 |
| BP_ZYNQ_INT_CAN1 | CAN 1 |
| BP_ZYNQ_INT_PL8 | Programmable logic 8 |
| BP_ZYNQ_INT_PL9 | Programmable logic 9 |
| BP_ZYNQ_INT_PL10 | Programmable logic 10 |
| BP_ZYNQ_INT_PL11 | Programmable logic 11 |
| BP_ZYNQ_INT_PL12 | Programmable logic 12 |
| BP_ZYNQ_INT_PL13 | Programmable logic 13 |
| BP_ZYNQ_INT_PL14 | Programmable logic 14 |
| BP_ZYNQ_INT_PL15 | Programmable logic 15 |

| | |
|---|---|
| `BP_ZYNQ_INT_SCU` | SCU |
| `BP_ZYNQ_INT_NONE` | Special invalid interrupt value |

Data Type

# `bp_zynq_reset_t`

<soc/zynq/bp_zynq_def.h>

Zynq-7000 peripheral reset lines list. These enumeration constants can be used with the reset management API such as `bp_periph_reset_assert()` and `bp_periph_reset_deassert()`.

*Values*

| | |
|---|---|
| `BP_ZYNQ_RESET_PSS` | Processing system |
| `BP_ZYNQ_RESET_DDR` | DDR |
| `BP_ZYNQ_RESET_TOPSW` | Central interconnect |
| `BP_ZYNQ_RESET_DMAC` | DMA controller |
| `BP_ZYNQ_RESET_USB0` | USB 0 |
| `BP_ZYNQ_RESET_USB1` | USB 1 |
| `BP_ZYNQ_RESET_GEM0` | Ethernet 0 |
| `BP_ZYNQ_RESET_GEM1` | Ethernet 1 |
| `BP_ZYNQ_RESET_SDIO0` | SDIO 0 |
| `BP_ZYNQ_RESET_SDIO1` | SDIO 1 |
| `BP_ZYNQ_RESET_SPI0` | SPI 0 |
| `BP_ZYNQ_RESET_SPI1` | SPI 1 |
| `BP_ZYNQ_RESET_CAN0` | CAN 0 |
| `BP_ZYNQ_RESET_CAN1` | CAN 1 |
| `BP_ZYNQ_RESET_I2C0` | I2C 0 |
| `BP_ZYNQ_RESET_I2C1` | I2C 1 |
| `BP_ZYNQ_RESET_UART0` | UART 0 |
| `BP_ZYNQ_RESET_UART1` | UART 1 |
| `BP_ZYNQ_RESET_GPIO` | GPIO |
| `BP_ZYNQ_RESET_QSPI` | QSPI |
| `BP_ZYNQ_RESET_SMC` | Static memory controller |
| `BP_ZYNQ_RESET_OCM` | On-chip memory controller |
| `BP_ZYNQ_RESET_FPGA0` | PL reset 0 |

| | |
|---|---|
| BP_ZYNQ_RESET_FPGA1 | PL reset 1 |
| BP_ZYNQ_RESET_FPGA2 | PL reset 2 |
| BP_ZYNQ_RESET_FPGA3 | PL reset 3 |
| BP_ZYNQ_RESET_A9_PERIPH | A9 MPCORE |
| BP_ZYNQ_RESET_A9_0 | A9 core 0 |
| BP_ZYNQ_RESET_A9_1 | A9 core 1 |
| BP_ZYNQ_RESET_NONE | Special invalid reset value |

Macro

## BP_ARM_GIC_SOC_DEF_CPU_IF_BASE

<soc/zynq/bp_zynq_def.h>

ARM GIC CPU interface base address.

Macro

## BP_ARM_GIC_SOC_DEF_DIST_BASE

<soc/zynq/bp_zynq_def.h>

ARM GIC distributor base address.

Macro

## BP_ARM_GIC_SOC_DEF_INT_CNT

<soc/zynq/bp_zynq_def.h>

Zynq-7000 number of valid interrupts.

Macro

## BP_ARM_SOC_DEF_BASE_PRIV_PERIPH

<soc/zynq/bp_zynq_def.h>

ARM MPCORE private peripheral base address.

Macro

## BP_L2C310_SOC_DEF_BASE

<soc/zynq/bp_zynq_def.h>

ARM L2C310 L2 Cache base address.

Macro

# BP_ZYNQ_BASE_*

<soc/zynq/bp_zynq_def.h>

Description Zynq-7000 peripheral base addresses.

| | |
|---|---|
| BP_ZYNQ_BASE_DDR_MEM_START | DDR memory start |
| BP_ZYNQ_BASE_SMC_MEM_START | DDR memory end |
| BP_ZYNQ_BASE_QSPI_MEM_START | QSPI memory start |
| BP_ZYNQ_BASE_PL_GP0_START | GP0 memory start |
| BP_ZYNQ_BASE_PL_GP1_START | GP1 memory start |
| BP_ZYNQ_BASE_SLCR | SLCR |
| BP_ZYNQ_BASE_UART0 | UART 0 |
| BP_ZYNQ_BASE_UART1 | UART 1 |
| BP_ZYNQ_BASE_USB0 | USB 0 |
| BP_ZYNQ_BASE_USB1 | USB 1 |
| BP_ZYNQ_BASE_I2C0 | I2C 0 |
| BP_ZYNQ_BASE_I2C1 | I2C 1 |
| BP_ZYNQ_BASE_SPI0 | SPI 0 |
| BP_ZYNQ_BASE_SPI1 | SPI 1 |
| BP_ZYNQ_BASE_CAN0 | CAN 0 |
| BP_ZYNQ_BASE_CAN1 | CAN 1 |
| BP_ZYNQ_BASE_GPIO0 | GPIO |
| BP_ZYNQ_BASE_ENET0 | Ethernet 0 |
| BP_ZYNQ_BASE_ENET1 | Ethernet 1 |
| BP_ZYNQ_BASE_QSPI0 | QSPI |
| BP_ZYNQ_BASE_SMC0 | Static memory controller registers |
| BP_ZYNQ_BASE_SDIO0 | SDIO 0 |
| BP_ZYNQ_BASE_SDIO1 | SDIO 1 |
| BP_ZYNQ_BASE_TTC0 | Triple timer counter 0 |
| BP_ZYNQ_BASE_TTC1 | Triple timer counter 1 |
| BP_ZYNQ_BASE_OCM | On-chip memory controller |

Chapter

# 3 ARMv7-AR Cache Control

Control and status functions for the ARM v7-AR cache control. The cache maintenance functions provided in this module will only manage the core integrated caches. For general and portable cache maintenance functions, an application should use the cache module API instead.

Function

## bp_arch_v7_bp_dis()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Disables branch prediction.

*Prototype* `void bp_arch_v7_bp_dis ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_bp_en()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Enables branch prediction.

*Prototype* `void bp_arch_v7_bp_en ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_bp_is_en()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Returns the enabled/disabled status of branch prediction.

*Prototype*

```
void bp_arch_v7_bp_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if branch prediction is enabled `false` otherwise.

Function

## bp_arch_v7_dcache_dis()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Disables the data cache.

*Prototype*

```
void bp_arch_v7_dcache_dis ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_dcache_en()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Enables the data cache.

*Prototype*

```
void bp_arch_v7_dcache_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_dcache_inv_all()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Invalidates the entire data cache.

*Prototype* `void bp_arch_v7_dcache_inv_all ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_dcache_is_en()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Returns the enabled/disabled status of the data cache.

*Prototype* `bool bp_arch_v7_dcache_is_en ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values* `true` if the data cache is enabled `false` otherwise.

Function

## bp_arch_v7_dcache_max_line_size_get()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Returns the maximum cache line size.

*Prototype* `uint32_t bp_arch_v7_dcache_max_line_size_get ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_arch_v7_dcache_min_line_size_get()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Returns the minimum cache line size.

*Prototype*

```
uint32_t  bp_arch_v7_dcache_min_line_size_get  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_arch_v7_dcache_range_clean()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Cleans an address range from the data cache.

A start address unaligned to a cache line will be truncated to be aligned with the next lowest cache line.

A length which is not a multiple of the cache line size will be rounded up to the next multiple of the cache line size.

Cleaning the cache means writing the dirty cache lines but keeping them stored in the cache.

This function cannot fail and supports cleaning from address 0. Calling `bp_arch_v7_dcache_range_clean()` with a `len` of 0 will have no effect other than executing a memory barrier.

*Prototype*

```
void  bp_arch_v7_dcache_range_clean  ( void *  p_addr,
                                       size_t  len );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `p_addr` | Start of the address range. |
| `len` | Length of the range to clean in bytes. |

Function

# bp_arch_v7_dcache_range_cleaninv()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Cleans and invalidates an address range from the data cache.

A start address unaligned to a cache line will be truncated to be aligned with the next lowest cache line.

A length which is not a multiple of the cache line size will be rounded up to the next multiple of the cache line size.

Combines operation of both `bp_arch_v7_dcache_range_clean()` and `bp_arch_v7_dcache_range_inv()` in one call. Some platforms may have optimized way of performing the combined operation.

It should not be assumed that the clean and invalidate operation are atomic between each other.

This function cannot fail and supports cleaning from address 0. Calling `bp_arch_v7_dcache_range_cleaninv()` with a `len` of 0 will have no effect other than executing a memory barrier.

*Prototype*

```
void  bp_arch_v7_dcache_range_cleaninv  ( void *  p_addr,
                                          size_t  len );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `p_addr` | Start of the address range. |
| `len` | Length of the range to clean and invalidate in bytes. |

Function

## `bp_arch_v7_dcache_range_inv()`

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Invalidates an address range from the data cache.

A start address unaligned to a cache line will be truncated to be aligned with the next lowest cache line.

A length which is not a multiple of the cache line size will be rounded up to the next multiple of the cache line size.

Invalidating the cache means clearing entries from the cache without writing them to main memory if dirty.

This function cannot fail and supports cleaning from address 0. Calling `bp_arch_v7_dcache_range_inv()` with a `len` of 0 will have no effect other than executing a memory barrier.

*Prototype*

```
void  bp_arch_v7_dcache_range_inv  ( void *  p_addr,
                                     size_t  len );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `p_addr` | Start of the address range. |
| `len` | Length of the range to invalidate in bytes. |

Function

## bp_arch_v7_icache_dis()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Disables the instruction cache.

*Prototype*

```
void  bp_arch_v7_icache_dis  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_icache_en()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Enables the instruction cache.

*Prototype*

```
void  bp_arch_v7_icache_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_icache_inv_all()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Invalidates the entire instruction cache.

*Prototype*

```
void  bp_arch_v7_icache_inv_all  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_arch_v7_icache_is_en()

<arch/port/arm-v7ar/bp_arm-v7ar_cache.h>

Returns the enabled/disabled status of the instruction cache.

*Prototype*

```
bool  bp_arch_v7_icache_is_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if the instruction cache is enabled `false` otherwise.

Chapter

# 4 ARMv7-AR Core Control

Control and status functions for the ARM v7-AR architecture.

Function

## bp_arch_v7_alloc_one_way_dis()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Disables the cache allocation policy in one way only.

*Prototype* `void bp_arch_v7_alloc_one_way_dis ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_alloc_one_way_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Enables the cache allocation policy in one way only.

*Prototype* `void bp_arch_v7_alloc_one_way_en ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_alloc_one_way_is_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns the enabled/disabled status of the cache allocation in one way feature.

*Prototype*

```
bool  bp_arch_v7_alloc_one_way_is_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if cache allocation in one way is enabled `false` otherwise.

Function

## bp_arch_v7_excl_dis()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Disables the exclusive cache policy.

*Prototype*

```
void  bp_arch_v7_excl_dis  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_excl_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Enables the exclusive cache policy.

*Prototype*

```
void  bp_arch_v7_excl_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_arch_v7_excl_is_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns the enabled/disabled status of the exclusive cache feature.

*Prototype*

```
bool bp_arch_v7_excl_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if exclusive cache is enabled `false` otherwise.

Function

# bp_arch_v7_fw_dis()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Disables cache maintenance operation forwarding.

*Prototype*

```
void bp_arch_v7_fw_dis ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_arch_v7_fw_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Enables cache maintenance operation forwarding.

*Prototype*

```
void bp_arch_v7_fw_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_fw_is_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns the enabled/disabled status of cache maintenance operation forwarding.

*Prototype*

```
bool bp_arch_v7_fw_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if cache maintenance operation forwarding is enabled `false` otherwise.

Function

## bp_arch_v7_l1_prefetch_dis()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Disables L1 cache automatic prefetch.

*Prototype*

```
void bp_arch_v7_l1_prefetch_dis ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_l1_prefetch_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Enables L1 cache automatic prefetch.

*Prototype*

```
void bp_arch_v7_l1_prefetch_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_arch_v7_l1_prefetch_is_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns the enabled/disabled status of the L1 cache prefetch.

*Prototype*

```
bool  bp_arch_v7_l1_prefetch_is_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if L1 prefetch is enabled `false` otherwise.

Function

# bp_arch_v7_l2_prefetch_dis()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Disables L2 cache automatic prefetch.

*Prototype*

```
void  bp_arch_v7_l2_prefetch_dis  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_arch_v7_l2_prefetch_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Enables L2 cache automatic prefetch.

*Prototype*

```
void  bp_arch_v7_l2_prefetch_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_l2_prefetch_is_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns the enabled/disabled status of the L2 cache prefetch.

*Prototype*

```
bool  bp_arch_v7_l2_prefetch_is_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if L2 prefetch is enabled `false` otherwise.

Function

## bp_arch_v7_midr_arch_read()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns the architecture portion of the MIDR register value.

*Prototype*

```
uint32_t  bp_arch_v7_midr_arch_read  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

MIDR architecture value.

Function

## bp_arch_v7_midr_impl_read()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns the implementer portion of the MIDR register value.

*Prototype*

```
uint32_t  bp_arch_v7_midr_impl_read  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values* MIDR implementer value.

Function

## bp_arch_v7_midr_pn_read()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns the part number portion of the MIDR register value.

*Prototype*

```
uint32_t  bp_arch_v7_midr_pn_read  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values* MIDR part number value.

Function

## bp_arch_v7_midr_read()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns MIDR register value.

*Prototype*

```
uint32_t  bp_arch_v7_midr_read  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values* MIDR register value.

Function

## bp_arch_v7_midr_rev_read()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns the revision portion of the MIDR register value.

*Prototype*

```
uint32_t  bp_arch_v7_midr_rev_read  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values* MIDR revision value.

Function

## bp_arch_v7_midr_var_read()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns the variant portion of the MIDR register value.

*Prototype*

```
uint32_t  bp_arch_v7_midr_var_read  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values* MIDR variant value.

Function

## bp_arch_v7_parity_dis()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Disables cache parity.

*Prototype*

```
void  bp_arch_v7_parity_dis  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_parity_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Enables cache parity.

*Prototype*

```
void  bp_arch_v7_parity_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_parity_is_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns the enabled/disabled status of the cache parity.

*Prototype*

```
bool  bp_arch_v7_parity_is_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if cache parity is enabled `false` otherwise.

Function

## bp_arch_v7_smp_dis()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Disables SMP operation.

*Prototype*

```
void  bp_arch_v7_smp_dis  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_smp_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Enables SMP operation.

*Prototype*

```
void  bp_arch_v7_smp_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_smp_is_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns the enabled/disabled status of the SMP bit.

*Prototype*

```
bool bp_arch_v7_smp_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if SMP is enabled `false` otherwise.

Function

## bp_arch_v7_write_full_line_of_zero_dis()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Disables the write full line of zeroes feature.

*Prototype*

```
void bp_arch_v7_write_full_line_of_zero_dis ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arch_v7_write_full_line_of_zero_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Enables the write full line of zeroes feature.

*Prototype*

```
void bp_arch_v7_write_full_line_of_zero_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_arch_v7_write_full_line_of_zero_is_en()

<arch/port/arm-v7ar/bp_arm-v7ar_ctrl.h>

Returns the enabled/disabled status of the write full line of zeroes feature.

*Prototype*

```
bool  bp_arch_v7_write_full_line_of_zero_is_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if write full line of zeroes is enabled `false` otherwise.

Chapter

# 5 ARM Generic Interrupt Controller

Interrupt controller implementation for the ARM GIC. The functions contained in this module are specific to the ARM GIC, for general interrupt control the interrupt module portable API should be used.

Function

## bp_gic_int_target_set()

<int/impl/arm_gic/bp_arm_gic_impl.h>

Configures the target CPU(s) of an ISR.

*Prototype*

```
int  bp_gic_int_target_set  ( int       id,
                              uint32_t  target );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `id` | Interrupt id of the interrupt to configure. |
| `target` | Bit mask of the targets to enable. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_gic_int_trig()

<int/impl/arm_gic/bp_arm_gic_impl.h>

Triggers a software interrupt request to the selected CPUs. The `cpus` argument will have no effect if the interrupt `id` is a shared or private peripheral interrupt id.

*Prototype*

```
int  bp_gic_int_trig  ( int       id,
                        uint32_t  cpus );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `id` | Interrupt id of the interrupt to configure. |
| `cpus` | Bit mask of the target CPUs. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_int_handler()

<int/impl/arm_gic/bp_arm_gic_impl.h>

Global interrupt handler. Must be called either from the OS or Bare-Metal port when an IRQ is triggered to handle the interrupt. This function should not be called directly by the application.

*Prototype*

```
void  bp_int_handler  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✓ |

Chapter

# 6 ARM L2C-310 L2 Cache Controller

Interface module to the ARM CoreLink L2C-310 cache controller. The functions contained in this module are specific to the ARM L2 Cache controller. For general cache management operations, the portable cache management API should be used instead.

Function

## bp_arm_l2c310_data_prefetch_dis()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Disables data prefetching.

*Prototype*

```
void  bp_arm_l2c310_data_prefetch_dis  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arm_l2c310_data_prefetch_en()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Enables data prefetching.

*Prototype*

```
void  bp_arm_l2c310_data_prefetch_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arm_l2c310_data_prefetch_is_en()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Returns the enabled/disabled state of the L2 cache controller data prefetching feature.

*Prototype*

```
bool bp_arm_l2c310_data_prefetch_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if data prefetching is enabled, `false` otherwise.

Function

## bp_arm_l2c310_dis()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Disables the L2 cache controller.

This function doesn't perform any cleaning prior to disabling the cache. It is the application responsibility to ensure the cache is clean prior to disabling it in order to prevent memory corruption.

*Prototype*

```
void bp_arm_l2c310_dis ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✗ | ✗ |

Function

## bp_arm_l2c310_early_bresp_dis()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Disables early BRESP.

*Prototype*

```
void bp_arm_l2c310_early_bresp_dis ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arm_l2c310_early_bresp_en()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Enables early BRESP.

*Prototype*

```
void bp_arm_l2c310_early_bresp_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arm_l2c310_early_bresp_is_en()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Returns the enabled/disabled state of the L2 cache controller early BRESP feature.

*Prototype*

```
bool bp_arm_l2c310_early_bresp_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if early BRESP is enabled, `false` otherwise.

Function

## bp_arm_l2c310_en()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Enables the L2 cache controller.

This function is designed to be called at a specific location in an application startup sequence. It is not thread safe and should not be called while the cache is already active.

*Prototype*

```
void bp_arm_l2c310_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✗ | ✗ |

Function

## bp_arm_l2c310_flz_dis()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Disables write full line of zeros.

*Prototype* `void bp_arm_l2c310_flz_dis ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arm_l2c310_flz_en()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Enables write full line of zeros.

*Prototype* `void bp_arm_l2c310_flz_en ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arm_l2c310_flz_is_en()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Returns the enabled/disabled state of the L2 cache controller write full line of zeros feature.

*Prototype* `bool bp_arm_l2c310_flz_is_en ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values* `true` if the full line of zero feature is enabled, `false` otherwise.

Function

# bp_arm_l2c310_init()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Initializes the L2 cache controller using platform defined configuration. The cache can subsequently be enabled with `arm_l2c310_en()`.

The various hardware configuration parameters must be available at compile time from the SoC definition.

This function is designed to be called at a specific location in an application startup sequence. It is not thread safe and should not be called while the cache is already active.

The following defines are required in the platform SoC definition include files.

- BP_L2C310_SOC_DEF_ASSOC Associativity, either 8 or 16.
- BP_L2C310_SOC_DEF_WAY_SIZE Way size, calculated as the cache size divided by the associativity.
- BP_L2C310_SOC_DEF_TRAM_WR_LAT Tag ram write latency, this is the value that will be written in the register so a configuration of 0 is equivalent to 1 cycle of latency.
- BP_L2C310_SOC_DEF_TRAM_RD_LAT Tag ram read latency.
- BP_L2C310_SOC_DEF_TRAM_ST_LAT Tag ram setup latency.
- BP_L2C310_SOC_DEF_DRAM_WR_LAT Data ram write latency.
- BP_L2C310_SOC_DEF_DRAM_RD_LAT Data ram read latency.
- BP_L2C310_SOC_DEF_DRAM_ST_LAT Data ram setup latency.

*Prototype*

```
void bp_arm_l2c310_init ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✗ | ✗ |

*Example*

```
// SoC level definitions for the L2C310 cache controller.
\#define BP_L2C310_SOC_DEF_ASSOC 8 // Associativity.

// Way size = cache_size / associativity.
\#define BP_L2C310_SOC_DEF_WAY_SIZE (64U*1024U)

\#define BP_L2C310_SOC_DEF_TRAM_WR_LAT 1 // Tag ram write latency.
\#define BP_L2C310_SOC_DEF_TRAM_RD_LAT 1 // Tag ram read latency.
\#define BP_L2C310_SOC_DEF_TRAM_ST_LAT 1 // Tag ram setup latency.

\#define BP_L2C310_SOC_DEF_DRAM_WR_LAT 1 // Data ram write latency.
\#define BP_L2C310_SOC_DEF_DRAM_RD_LAT 2 // Data ram read latency.
\#define BP_L2C310_SOC_DEF_DRAM_ST_LAT 1 // Data ram setup latency.
```

Function

## bp_arm_l2c310_ins_prefetch_dis()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Disables instruction prefetching.

*Prototype*

```
void bp_arm_l2c310_ins_prefetch_dis ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arm_l2c310_ins_prefetch_en()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Enables instruction prefetching.

*Prototype*

```
void bp_arm_l2c310_ins_prefetch_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_arm_l2c310_ins_prefetch_is_en()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Returns the enabled/disabled state of the L2 cache controller instruction prefetching feature.

*Prototype*

```
bool bp_arm_l2c310_ins_prefetch_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if instruction prefetching is enabled, `false` otherwise.

Function

## bp_arm_l2c310_inv_all()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Invalidates the entire L2 cache.

This function is designed to be called during the L2 cache initialization or when waking up from a sleep mode where the cache state is unknown.

*Prototype*

```
void bp_arm_l2c310_inv_all ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✗ |

Function

## bp_arm_l2c310_is_en()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Returns the enabled/disabled state of the L2 cache controller.

*Prototype*

```
bool bp_arm_l2c310_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if the L2 cache is enabled, `false` otherwise.

Function

## bp_arm_l2c310_range_clean()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Cleans an address range from the L2 cache. This function operates on the L2 cache only, `bp_arch_dcache_range_clean()` should normally be used to perform a general cache clean.

A start address unaligned to a cache line will be truncated to be aligned with the next lowest cache line.

A length which is not a multiple of the cache line size will be rounded up to the next multiple of the cache line size.

Cleaning the cache means writing the dirty cache lines but keeping them stored in the cache.

This function cannot fail and supports cleaning from address 0. Calling `bp_arm_l2c310_range_clean()` with a `len` of 0 will have no effect.

*Prototype*

```
void  bp_arm_l2c310_range_clean  ( void *  p_addr,
                                   size_t  len );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `p_addr` | Start of the address range. |
| `len` | Length of the range to clean in bytes. |

Function

## `bp_arm_l2c310_range_cleaninv()`

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Cleans and invalidates an address range from the L2 cache. This function operates on the L2 cache only, `bp_arch_dcache_range_cleaninv()` should normally be used to perform a general cache clean and invalidate.

A start address unaligned to a cache line will be truncated to be aligned with the next lowest cache line.

A length which is not a multiple of the cache line size will be rounded up to the next multiple of the cache line size.

It should not be assumed that the clean and invalidate operation are atomic between each other.

*Prototype*

```
void  bp_arm_l2c310_range_cleaninv  ( void *  p_addr,
                                     size_t  len );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `p_addr` | Start of the address range. |
| `len` | Length of the range to clean and invalidate in bytes. |

Function

## `bp_arm_l2c310_range_inv()`

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Invalidates an address range from the L2 cache. This function operates on the L2 cache only, `bp_arch_dcache_range_inv()` should normally be used to perform a general cache invalidate.

A start address unaligned to a cache line will be truncated to be aligned with the next lowest cache line.

A length which is not a multiple of the cache line size will be rounded up to the next multiple of the cache line size.

Invalidating the cache means clearing entries from the cache without writing them to main memory if dirty.

This function cannot fail and supports invalidating from address 0. Calling `bp_arm_l2c310_range_inv()` with a `len` of 0 will have no effect.

*Prototype*

```
void  bp_arm_l2c310_range_inv  ( void *  p_addr,
                                 size_t  len );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `p_addr` | Start of the address range. |
| `len` | Length of the range to invalidate in bytes. |

Function

## bp_arm_l2c310_sync()

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

Performs a cache sync.

*Prototype*

```
void  bp_arm_l2c310_sync  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Macro

## BP_ARM_L2C310_CACHE_LINE_SIZE

<soc_comp/arm/arm_l2c310/bp_arm_l2c310.h>

L2 cache line size in bytes.

Chapter

# 7 Zynq-7000 On-Chip Memory (OCM)

Interface module to the Zynq-7000 OCM peripheral. Note that enabling or disabling parity checking while the on-chip RAM is used could have unintended consequences.

Function

## bp_zynq_ocm_arb_dis()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Disables OCM port arbitration.

*Prototype* `void bp_zynq_ocm_arb_dis ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_zynq_ocm_arb_en()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Enables OCM port arbitration.

*Prototype* `void bp_zynq_ocm_arb_en ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_zynq_ocm_arb_is_en()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Returns the enabled/disabled status of the OCM port arbitration.

*Prototype*

```
bool bp_zynq_ocm_arb_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if OCM port arbitration is enabled `false` otherwise.

Function

# bp_zynq_ocm_cb_reg()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Register a callback function to be called when the OCM interrupt fires. The callback function can be NULL in which case no callback will be called when the interrupt fires.

*Prototype*

```
int bp_zynq_ocm_cb_reg ( bp_zynq_ocm_cb_t p_cb );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

`p_cb` Callback function to set.

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_zynq_ocm_init()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Initialized the ZYNQ OCM module. This clear, disable and setup the OCM interrupts.

It is not necessary to call `bp_zynq_ocm_init()` prior to using the on-chip ram itself.

*Prototype*

```
int bp_zynq_ocm_init ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✓ |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_ocm_irq_sts_clear()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Clears any asserted interrupts.

*Prototype*

```
void  bp_zynq_ocm_irq_sts_clear  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

Current value of the interrupt status bits.

Function

## bp_zynq_ocm_irq_sts_get()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Returns the value of the interrupt status bits.

*Prototype*

```
uint32_t  bp_zynq_ocm_irq_sts_get  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

Current value of the interrupt status bits.

Function

## bp_zynq_ocm_lock_fail_irq_dis()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Disables AXI lock request interrupt.

*Prototype*

```
void bp_zynq_ocm_lock_fail_irq_dis ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_zynq_ocm_lock_fail_irq_en()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Enables AXI lock request interrupt.

*Prototype*

```
void bp_zynq_ocm_lock_fail_irq_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_zynq_ocm_lock_fail_irq_is_en()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Returns the enabled/disabled state of the AXI lock request interrupt enable.

*Prototype*

```
bool bp_zynq_ocm_lock_fail_irq_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if the interrupt is enabled `false` otherwise.

Function

# bp_zynq_ocm_multi_par_err_irq_dis()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Disables multiple parity error interrupt.

*Prototype*

```
void bp_zynq_ocm_multi_par_err_irq_dis ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_zynq_ocm_multi_par_err_irq_en()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Enables multiple parity error interrupt.

*Prototype*

```
void bp_zynq_ocm_multi_par_err_irq_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_zynq_ocm_multi_par_err_irq_is_en()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Returns the enabled/disabled state of the multiple parity error interrupt enable.

*Prototype*

```
bool bp_zynq_ocm_multi_par_err_irq_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if the interrupt is enabled `false` otherwise.

Function

## bp_zynq_ocm_odd_parity_get()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Returns the value of the odd parity enable field.

*Prototype*

```
uint32_t bp_zynq_ocm_odd_parity_get ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values* Value of the odd parity enable field.

Function

## bp_zynq_ocm_odd_parity_set()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Sets the value of the odd parity enable field.

*Prototype*

```
void bp_zynq_ocm_odd_parity_set ( uint32_t odd_mask );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| odd_mask | Value of the odd parity enable field to set. |

Function

## bp_zynq_ocm_parity_dis()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Disable OCM parity checking.

*Prototype*

```
void bp_zynq_ocm_parity_dis ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_zynq_ocm_parity_en()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Enables OCM parity checking.

*Prototype*

```
void bp_zynq_ocm_parity_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_zynq_ocm_parity_err_addr()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Returns the parity error address.

*Prototype*

```
uint32_t bp_zynq_ocm_parity_err_addr ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

Current value of the parity error address register.

Function

## bp_zynq_ocm_parity_err_addr_clear()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Clears the parity error address register.

*Prototype*

```
void bp_zynq_ocm_parity_err_addr_clear ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_zynq_ocm_parity_is_en()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Returns the enabled/disabled status of OCM parity checking.

*Prototype*

```
bool bp_zynq_ocm_parity_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if parity checking is enabled 'false otherwise.

Function

## bp_zynq_ocm_scu_wr_prio_low_dis()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Disables OCM SCU write port low priority.

*Prototype*

```
void bp_zynq_ocm_scu_wr_prio_low_dis ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_zynq_ocm_scu_wr_prio_low_en()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Enables OCM SCU write port low priority.

*Prototype*

```
void bp_zynq_ocm_scu_wr_prio_low_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_zynq_ocm_scu_wr_prio_low_is_en()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Returns the enabled/disabled status of OCM SCU write port low priority.

*Prototype*

```
bool bp_zynq_ocm_scu_wr_prio_low_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if OCM SCU write priority is low `false` otherwise.

Function

# bp_zynq_ocm_single_par_err_irq_dis()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Disables single parity error interrupt.

*Prototype*

```
void bp_zynq_ocm_single_par_err_irq_dis ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_zynq_ocm_single_par_err_irq_en()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Enables single parity error interrupt.

*Prototype*

```
void bp_zynq_ocm_single_par_err_irq_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_zynq_ocm_single_par_err_irq_is_en()

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Returns the enabled/disabled state of the single parity error interrupt enable.

*Prototype*

```
bool bp_zynq_ocm_single_par_err_irq_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if the interrupt is enabled `false` otherwise.

Data Type

## bp_zynq_ocm_cb_t

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

OCM interrupt callback function signature.

*Prototype*

```
void bp_zynq_ocm_cb_t ( uint32_t irq_stastus );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `irq_stastus` | Bit mask the asserted interrupts when the handler is called. |

Macro

## BP_ZYNQ_OCM_STS_LOCK_FAIL

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Interrupt status flags.AXI LOCK requested.

Macro

## BP_ZYNQ_OCM_STS_MULTI_PAR_ERR

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Multiple parity error detected.

Macro

# BP_ZYNQ_OCM_STS_SINGLE_PAR_ERR

<soc_comp/zynq/zynq_ocm/bp_zynq_ocm.h>

Single parity error detected.

Chapter

# 8 Zynq-7000 System Level Controller (SLCR)

Interface module to the Zynq-7000 System Level Controller (SLCR). This module also provides the implementations of the clock and reset modules. Note that the generic clock and reset modules can be used to manipulate the clock and reset lines at a higher level using a portable API.

Function

## bp_zynq_slcr_clock_dis()

<soc_comp/zynq/zynq_slcr/bp_zynq_slcr.h>

Disables a specific clock.

*Prototype*

```
int bp_zynq_slcr_clock_dis ( bp_zynq_clk_t clk );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

`clk` Clock to disable.

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_slcr_clock_dump()

<soc_comp/zynq/zynq_slcr/bp_zynq_slcr.h>

Dump the clock frequency of all the clocks to the console.

*Prototype*

```
int  bp_zynq_slcr_clock_dump  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_slcr_clock_en()

<soc_comp/zynq/zynq_slcr/bp_zynq_slcr.h>

Enables a specific clock.

*Prototype*

```
int  bp_zynq_slcr_clock_en  ( bp_zynq_clk_t  clk );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `clk` | Clock to enable. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_slcr_clock_freq_get()

<soc_comp/zynq/zynq_slcr/bp_zynq_slcr.h>

Returns the frequency of a specific clock in Hertz.

*Prototype*

```
int  bp_zynq_slcr_clock_freq_get  ( bp_zynq_clk_t  clk,
                                    uint32_t *     p_clk_freq );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `clk` | Clock to query. |
| `p_clk_freq` | Pointer to the returned clock frequency in Hertz. |

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Function

# bp_zynq_slcr_clock_is_en()

<soc_comp/zynq/zynq_slcr/bp_zynq_slcr.h>

Returned the state of a specific clock. Returns `true` through `p_is_en` if the clock `clk` is enabled, `false` otherwise.

*Prototype*
```
int  bp_zynq_slcr_clock_is_en  ( bp_zynq_clk_t  clk,
                                 bool *         p_is_en );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `clk` | Clock to query. |
| `p_is_en` | Pointer to the returned state. |

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Function

# bp_zynq_slcr_is_locked()

<soc_comp/zynq/zynq_slcr/bp_zynq_slcr.h>

Returns the state of the SLCR lock.

*Prototype*
```
bool  bp_zynq_slcr_is_locked  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*
- true if the SLCR is locked, false otherwise.

Function

# bp_zynq_slcr_lock()

<soc_comp/zynq/zynq_slcr/bp_zynq_slcr.h>

Lock the SLCR.

*Prototype*

```
void bp_zynq_slcr_lock ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_zynq_slcr_mio_set()

<soc_comp/zynq/zynq_slcr/bp_zynq_slcr.h>

Sets the configuration of an MIO pin. The pin mux and I/O attributes of MIO pin `mio` will be set to the requested parameters.

*Prototype*

```
int bp_zynq_slcr_mio_set (uint32_t mio,
                          uint32_t dis_rcvr,
                          uint32_t pullup,
                          uint32_t io_type,
                          uint32_t speed,
                          uint32_t l3_sel,
                          uint32_t l2_sel,
                          uint32_t l1_sel,
                          uint32_t l0_sel );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `mio` | Numerical id of the MIO pin to set. |
| `dis_rcvr` | Set to 1 to disable the HSTL receiver. |
| `pullup` | Set to 1 to enable pullup. |
| `io_type` | IO type of the MIO pin. |
| `speed` | Set to 1 for fast, 0 for slow. |
| `l3_sel` | Configuration value of the L3 mux. |
| `l2_sel` | Configuration value of the L2 mux. |
| `l1_sel` | Configuration value of the L1 mux. |
| `l0_sel` | Configuration value of the L0 mux. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_slcr_reset_assert()

<soc_comp/zynq/zynq_slcr/bp_zynq_slcr.h>

Asserts a peripheral reset line.

*Prototype*

```
int  bp_zynq_slcr_reset_assert  ( bp_zynq_reset_t  reset );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `reset` | Reset line to assert. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_slcr_reset_deassert()

<soc_comp/zynq/zynq_slcr/bp_zynq_slcr.h>

Deasserts a peripheral reset line.

*Prototype*

```
int  bp_zynq_slcr_reset_deassert  ( bp_zynq_reset_t  reset );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `reset` | Reset line to deassert. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_slcr_reset_is_asserted()

<soc_comp/zynq/zynq_slcr/bp_zynq_slcr.h>

Returns the asserted/deasserted state of a reset line.

*Prototype*

```
int  bp_zynq_slcr_reset_is_asserted  ( bp_zynq_reset_t  reset,
                                       bool *           p_is_asserted );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `reset` | Reset line to query. |
| `p_is_asserted` | Pointer to the result, `true` if the reset is asserted, `false` otherwise. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_zynq_slcr_unlock()

<soc_comp/zynq/zynq_slcr/bp_zynq_slcr.h>

Unlock the SLCR.

*Prototype*

```
void  bp_zynq_slcr_unlock  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Chapter

# 9 Zynq-7000 System Watchdog Timer (SWDT)

Interface module to the Zynq-7000 System Watchdog Timer.

Function

## bp_zynq_swdt_dis()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Disables the watchdog counter.

*Prototype* `void bp_zynq_swdt_dis ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_zynq_swdt_en()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Enables the watchdog counter.

*Prototype* `void bp_zynq_swdt_en ( );`

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_zynq_swdt_init()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Initializes the system watchdog module.

*Prototype*

```
int  bp_zynq_swdt_init  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✓ |

*Returned Errors*

RTNC_SUCCESS
RTNC_FATAL

Function

# bp_zynq_swdt_irq_dis()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Disables the watchdog interrupt.

*Prototype*

```
void  bp_zynq_swdt_irq_dis  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_zynq_swdt_irq_en()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Enables the watchdog interrupt.

*Prototype*

```
void  bp_zynq_swdt_irq_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_zynq_swdt_irq_is_en()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Returns the enabled/disabled status of the watchdog interrupt.

*Prototype*

```
bool bp_zynq_swdt_irq_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if the watchdog interrupt is enabled `false` otherwise.

Function

# bp_zynq_swdt_irqln_get()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Returns the length of the IRQ pulse.

*Prototype*

```
bp_zynq_swdt_irqln_t bp_zynq_swdt_irqln_get ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

Length of the configured interrupt pulse.

Function

# bp_zynq_swdt_irqln_set()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Sets the length of the IRQ pulse. `bp_zynq_swdt_irqln_set()` will return `RTNC_FATAL` if an invalid pulse length is specified.

*Prototype*

```
int bp_zynq_swdt_irqln_set ( bp_zynq_swdt_irqln_t irqln );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters* `irqln` IRQ pulse length to set.

*Returned Errors* `RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_swdt_is_en()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Returns the enabled/disabled status of the watchdog counter.

*Prototype*

```
bool  bp_zynq_swdt_is_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values* `true` if the watchdog is started `false` otherwise.

Function

## bp_zynq_swdt_prescale_get()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Returns the clock prescaler value.

*Prototype*

```
uint32_t  bp_zynq_swdt_prescale_get  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values* Configured clock prescaler value.

Function

## bp_zynq_swdt_prescale_set()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Sets the clock prescaler value. `bp_zynq_swdt_prescale_set()` will return `RTNC_FATAL` if an invalid prescaler value is specified.

*Prototype*

```
int bp_zynq_swdt_prescale_set ( uint32_t prescale );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

`prescale` Prescaler value.

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_swdt_restart()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Restarts the watchdog timer.

*Prototype*

```
void bp_zynq_swdt_restart ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

## bp_zynq_swdt_restart_val_get()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Returns the counter restart value

*Prototype*

```
uint32_t bp_zynq_swdt_restart_val_get ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

Upper 4 bits of the configured counter restart value.

Function

# bp_zynq_swdt_restart_val_set()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Sets the restart count value. `bp_zynq_swdt_restart_val_set()` will return `RTNC_FATAL` if an invalid pulse length is specified.

*Prototype*

```
int bp_zynq_swdt_restart_val_set (uint32_t restart_val);
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `restart_val` | Upper 4 bits of the Counter restart value. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_zynq_swdt_rst_dis()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Disables the watchdog reset output.

*Prototype*

```
void bp_zynq_swdt_rst_dis ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_zynq_swdt_rst_en()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Enables the watchdog reset output.

*Prototype*

```
void bp_zynq_swdt_rst_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

Function

# bp_zynq_swdt_rst_is_en()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Returns the enabled/disabled status of the watchdog reset output.

*Prototype*

```
bool bp_zynq_swdt_rst_is_en ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if the watchdog reset is enabled `false` otherwise.

Function

# bp_zynq_swdt_status_get()

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Returns the watchdog status.

*Prototype*

```
bool bp_zynq_swdt_status_get ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Values*

`true` if the watchdog expired `false` otherwise.

Data Type

# bp_zynq_swdt_irqln_t

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

IRQ pulse length. See `bp_zynq_swdt_irqln_set()` and `bp_zynq_swdt_irqln_get()` for usage details.

*Values*

| | |
|---|---|
| `BP_ZYNQ_SWDT_IRQLN_4` | WDOG IRQ held for 4 pclk. |
| `BP_ZYNQ_SWDT_IRQLN_8` | WDOG IRQ held for 8 pclk. |
| `BP_ZYNQ_SWDT_IRQLN_16` | WDOG IRQ held for 16 pclk. |
| `BP_ZYNQ_SWDT_IRQLN_32` | WDOG IRQ held for 32 pclk. |

Data Type

# bp_zynq_swdt_prescale_t

<soc_comp/zynq/zynq_swdt/bp_zynq_swdt.h>

Prescaler values. See `bp_zynq_swdt_prescale_set()` and `bp_zynq_swdt_prescale_get()` for usage details.

*Values*

| | |
|---|---|
| `BP_ZYNQ_SWDT_PRESCALE_8` | Input clock divided by 8. |
| `BP_ZYNQ_SWDT_PRESCALE_64` | Input clock divided by 64. |
| `BP_ZYNQ_SWDT_PRESCALE_512` | Input clock divided by 512. |
| `BP_ZYNQ_SWDT_PRESCALE_4096` | Input clock divided by 4096. |

Chapter

10

# Zynq-7000 Triple Timer Counter (TTC)

Interface module to the Zynq-7000 Triple Timer Counter. For all the API functions, `ttc_id` is the TTC peripheral instance to access (i.e. 0 for TTC0, 1 for TTC1) and `counter_id` is the counter number within the TTC instance (i.e 0 to 2).

Function

## bp_zynq_ttc_cfg_get()

<soc_comp/zynq/zynq_ttc/bp_zynq_ttc.h>

Retrieves the configuration of a counter. Returns the configuration of counter `counter_id` of instance `ttc_id` through `p_cfg`.

*Prototype*

```
int  bp_zynq_ttc_cfg_get  ( uint32_t             ttc_id,
                            uint32_t             counter_id,
                            bp_zynq_ttc_cfg_t *  p_cfg );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `ttc_id` | Triple Timer Counter instance id. |
| `counter_id` | Counter id. |
| `p_cfg` | Pointer to the returned configuration. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_zynq_ttc_cfg_set()

<soc_comp/zynq/zynq_ttc/bp_zynq_ttc.h>

Configures a counter. Configures the counter `counter_id` of instance `ttc_id` with the configuration `p_cfg`.

*Prototype*

```
int  bp_zynq_ttc_cfg_set  ( uint32_t             ttc_id,
                            uint32_t             counter_id,
                            bp_zynq_ttc_cfg_t *  p_cfg );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `ttc_id` | Triple Timer Counter instance id. |
| `counter_id` | Counter id. |
| `p_cfg` | Configuration to apply. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_zynq_ttc_count_get()

<soc_comp/zynq/zynq_ttc/bp_zynq_ttc.h>

Returns the counter value of a counter. Returns the counter value of counter `counter_id` of instance `ttc_id` through `p_count`.

*Prototype*

```
int  bp_zynq_ttc_count_get  ( uint32_t   ttc_id,
                              uint32_t   counter_id,
                              uint32_t *  p_count );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `ttc_id` | Triple Timer Counter instance id. |
| `counter_id` | Counter id. |
| `p_count` | Pointer to the returned count value. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_zynq_ttc_dis()

<soc_comp/zynq/zynq_ttc/bp_zynq_ttc.h>

Disables a counter. Disables counter `counter_id` of instance `ttc_id`.

*Prototype*

```
int  bp_zynq_ttc_dis  ( uint32_t  ttc_id,
                        uint32_t  counter_id );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `ttc_id` | Triple Timer Counter instance id. |
| `counter_id` | Counter id. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_zynq_ttc_en()

<soc_comp/zynq/zynq_ttc/bp_zynq_ttc.h>

Enables a counter. Enables counter `counter_id` of instance `ttc_id`.

*Prototype*

```
int  bp_zynq_ttc_en  ( uint32_t  ttc_id,
                       uint32_t  counter_id );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `ttc_id` | Triple Timer Counter instance id. |
| `counter_id` | Counter id. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_zynq_ttc_init()

<soc_comp/zynq/zynq_ttc/bp_zynq_ttc.h>

Initializes the ZYNQ TTC module.

*Prototype*

```
int bp_zynq_ttc_init ( );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✓ |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_ttc_is_en()

<soc_comp/zynq/zynq_ttc/bp_zynq_ttc.h>

Returns the enabled/disabled state of a counter. Returns the state of counter `counter_id` of instance `ttc_id` through `p_is_en`.

*Prototype*

```
int bp_zynq_ttc_is_en ( uint32_t ttc_id,
                        uint32_t counter_id,
                        bool *   p_is_en );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `ttc_id` | Triple Timer Counter instance id. |
| `counter_id` | Counter id. |
| `p_is_en` | Pointer to the returned state. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_ttc_prescaler_get()

<soc_comp/zynq/zynq_ttc/bp_zynq_ttc.h>

Returns the prescaler value of a counter. Returns the prescaler value of counter `counter_id` of instance `ttc_id` through `p_prescale`.

*Prototype*

```
int bp_zynq_ttc_prescaler_get ( uint32_t   ttc_id,
                                uint32_t   counter_id,
                                uint32_t * p_prescale );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `ttc_id` | Triple Timer Counter instance id. |
| `counter_id` | Counter id. |
| `p_prescale` | Pointer to the returned prescaler value. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_ttc_prescaler_set()

<soc_comp/zynq/zynq_ttc/bp_zynq_ttc.h>

Sets the prescaler value of a counter. Sets the prescaler of counter `counter_id` of instance `ttc_id` with the value `prescale`.

*Prototype*

```
int  bp_zynq_ttc_prescaler_set  ( uint32_t  ttc_id,
                                  uint32_t  counter_id,
                                  uint32_t  prescale );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `ttc_id` | Triple Timer Counter instance id. |
| `counter_id` | Counter id. |
| `prescale` | Prescale value to apply. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_ttc_reset()

<soc_comp/zynq/zynq_ttc/bp_zynq_ttc.h>

Resets a counter. Resets counter `counter_id` of instance `ttc_id`.

*Prototype*

```
int  bp_zynq_ttc_reset  ( uint32_t  ttc_id,
                          uint32_t  counter_id );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `ttc_id` | Triple Timer Counter instance id. |
| `counter_id` | Counter id. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

# bp_zynq_ttc_cfg_t

<soc_comp/zynq/zynq_ttc/bp_zynq_ttc.h>

Zynq Triple Timer Counter configuration structure. See `bp_zynq_ttc_cfg_set()` and `bp_zynq_ttc_cfg_get()` for usage details.

*Members*

| | | |
|---|---|---|
| `wave_polarity` | `bool` | Output wave polarity. `true` for high, `false` for low. |
| `wave_en` | `bool` | Enable output waveform. |
| `match_en` | `bool` | Enable compare match. |
| `decrement` | `bool` | Set to `true` for countdown. |
| `clock_src` | `bool` | Set to `true` to use the external clock source. |
| `clock_edge` | `bool` | Set to `true` to use the negative external clock edge. |

Chapter

11

# Zynq-7000 GPIO Driver

GPIO driver for the Xilinx Zynq-7000. Note that for most applications it is recommended to use the GPIO module API instead of the driver interface. This module contains the GPIO driver interface to be used by the GPIO module as well as additional driver specific functions. The driver interface as well as the driver specific functions can be called by the application using the driver handle which can be retrieved using the `bp_gpio_drv_hndl_get()` function.

GPIO pin numbering follows a similar convention to the one found in the manufacturer's documentation. In all cases the bank number should be 0 and pins numbered 0 to 53 corresponds to MIO output 0 to 53 while pins 64 to 127 corresponds to the EMIO pins 0 to 63.

See the BASEplatform manual for additional information on calling the driver interface directly.

Function

## bp_zynq_gpio_create()

<gpio/drv/zynq/bp_zynq_gpio_drv.h>

Creates a GPIO driver instance.

See `bp_gpio_drv_create_t` for usage details.

*Prototype*

```
int  bp_zynq_gpio_create  ( const bp_gpio_board_def_t *  p_def,
                            bp_gpio_drv_hndl_t *         p_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `p_def` | Board definition of the GPIO peripheral to initialize. |
| `p_hndl` | Handle to the created GPIO driver instance. |

*Returned Errors*
- `RTNC_SUCCESS`
- `RTNC_NO_RESOURCE`
- `RTNC_FATAL`

Function

# bp_zynq_gpio_data_get()

<gpio/drv/zynq/bp_zynq_gpio_drv.h>

Gets the state of a GPIO pin.

See `bp_gpio_drv_data_get_t` for usage details.

*Prototype*

```
int  bp_zynq_gpio_data_get  ( bp_gpio_drv_hndl_t  hndl,
                              uint32_t            bank,
                              uint32_t            pin,
                              uint32_t *          p_data );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to query. |
| `bank` | Bank number of the pin to query. |
| `pin` | Pin number of the pin to query. |
| `p_data` | Pointer to the variable that will receive the data. |

*Returned Errors*
- `RTNC_SUCCESS`
- `RTNC_FATAL`

Function

# bp_zynq_gpio_data_set()

<gpio/drv/zynq/bp_zynq_gpio_drv.h>

Sets the state of a GPIO pin.

See `bp_gpio_drv_data_set_t` for usage details.

*Prototype*

```
int  bp_zynq_gpio_data_set  ( bp_gpio_drv_hndl_t  hndl,
                              uint32_t            bank,
                              uint32_t            pin,
                              uint32_t            data );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to set. |
| `bank` | Bank number of the pin to set. |
| `pin` | Pin number of the pin to set. |
| `data` | State of the pin to set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_gpio_data_tog()

<gpio/drv/zynq/bp_zynq_gpio_drv.h>

Toggle the state of a GPIO pin.

See `bp_gpio_drv_data_tog_t` for usage details.

*Prototype*

```
int  bp_zynq_gpio_data_tog  ( bp_gpio_drv_hndl_t  hndl,
                              uint32_t            bank,
                              uint32_t            pin );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to toggle. |
| `bank` | Bank number of the pin to toggle. |
| `pin` | Pin number of the pin to toggle. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_gpio_destroy()

<gpio/drv/zynq/bp_zynq_gpio_drv.h>

Destroys a GPIO driver instance.

See `bp_gpio_drv_destroy_t` for usage details.

*Prototype*

```
int  bp_zynq_gpio_destroy  ( bp_gpio_drv_hndl_t  hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters* `hndl` Handle of the GPIO driver instance to destroy.

*Returned Errors* `RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_zynq_gpio_dir_get()

<gpio/drv/zynq/bp_zynq_gpio_drv.h>

Gets the direction of a GPIO pin.

See `bp_gpio_drv_dir_get_t` for usage details.

*Prototype*

```
int  bp_zynq_gpio_dir_get  ( bp_gpio_drv_hndl_t  hndl,
                             uint32_t            bank,
                             uint32_t            pin,
                             bp_gpio_dir_t *     p_dir );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to query. |
| `bank` | Bank number of the pin to query. |
| `pin` | Pin number of the pin to query. |
| `p_dir` | Pointer to the variable that will receive the direction. |

*Returned Errors* `RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_zynq_gpio_dir_set()

<gpio/drv/zynq/bp_zynq_gpio_drv.h>

Sets the direction of a GPIO pin.

See `bp_gpio_drv_dir_set_t` for usage details.

*Prototype*

```
int  bp_zynq_gpio_dir_set  ( bp_gpio_drv_hndl_t  hndl,
                             uint32_t            bank,
                             uint32_t            pin,
                             dir );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to set. |
| `bank` | Bank number of the pin to set. |
| `pin` | Pin number of the pin to set. |
| `dir` | Direction of the pin to set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_gpio_dis()

<gpio/drv/zynq/bp_zynq_gpio_drv.h>

Disables a GPIO interface.

See `bp_gpio_drv_dis_t` for usage details.

*Prototype*

```
int  bp_zynq_gpio_dis  ( bp_gpio_drv_hndl_t  hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the GPIO driver to disable. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_gpio_en()

<gpio/drv/zynq/bp_zynq_gpio_drv.h>

Enables a GPIO interface.

See `bp_gpio_drv_en_t` for usage details.

*Prototype*

```
int  bp_zynq_gpio_en  ( bp_gpio_drv_hndl_t  hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters* `hndl` Handle of the GPIO driver to enable.

*Returned Errors* `RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_gpio_is_en()

<gpio/drv/zynq/bp_zynq_gpio_drv.h>

Returns the enabled/disabled state of a GPIO interface.

See `bp_gpio_drv_is_en_t` for usage details.

*Prototype*

```
int  bp_zynq_gpio_is_en  ( bp_gpio_drv_hndl_t  hndl,
                           bool *              p_is_en );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the GPIO driver to query. |
| `p_is_en` | Interface state, true if enabled false otherwise. |

*Returned Errors* `RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

## bp_zynq_gpio_drv_def_t

<gpio/drv/zynq/bp_zynq_gpio_drv.h>

Zynq GPIO driver hardware definition structure. Those parameters are required by the GPIO driver and are configured through a `bp_gpio_soc_def_t` structure.

*Members*

| | | |
|---|---|---|
| `base_addr` | `void *` | Peripheral base address. |
| `int_id` | `uint32_t` | Peripheral interrupt id. |

Chapter

12

# Zynq-7000 UART Driver

UART driver for the Xilinx Zynq-7000. Note that for most applications it is recommended to use the UART module API instead of the driver interface. This module contains the UART driver interface to be used by the UART module as well as additional driver specific functions. The driver interface as well as the driver specific functions can be called by the application using the driver handle which can be retrieved using the `bp_uart_drv_hndl_get()` function.

See the BASEplatform manual for additional information on calling the driver interface directly.

Function

## `bp_zynq_uart_cfg_get()`

<uart/drv/zynq/bp_zynq_uart_drv.h>

Retrieves the current configuration of a UART interface.

See `bp_uart_drv_cfg_get_t` for usage details.

*Prototype*

```
int  bp_zynq_uart_cfg_get  ( bp_uart_drv_hndl_t  hndl,
                             bp_uart_cfg_t *     p_cfg,
                                                 timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART peripheral to query. |
| `p_cfg` | Pointer to the UART configuration. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Function

# bp_zynq_uart_cfg_set()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Configures a UART peripheral.

See bp_uart_drv_cfg_set_t for usage details.

*Prototype*

```
int  bp_zynq_uart_cfg_set  (                        hndl,
                            const bp_uart_cfg_t *   p_cfg,
                            uint32_t                timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the UART peripheral to configure. |
| p_cfg | UART configuration. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_NOT_SUPPORTED
RTNC_FATAL

Function

# bp_zynq_uart_create()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Creates a UART driver instance.

See bp_uart_drv_create_t for usage details.

*Prototype*

```
int  bp_zynq_uart_create  ( const bp_uart_board_def_t *  p_def,
                            bp_uart_drv_hndl_t *         p_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `p_def` | Board definition of the UART peripheral to initialize. |
| `p_hndl` | Pointer to the newly created UART driver instance. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

## bp_zynq_uart_destroy()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Destroys a UART driver instance.

See `bp_uart_drv_destroy_t` for usage details.

*Prototype*

```
int  bp_zynq_uart_destroy  ( bp_uart_drv_hndl_t  hndl,
                             uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART driver instance to destroy. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_uart_dis()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Disables a UART peripheral.

See `bp_uart_drv_dis_t` for usage details.

*Prototype*

```
int  bp_zynq_uart_dis  ( bp_uart_drv_hndl_t  hndl,
                         uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART peripheral to disable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_uart_en()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Enables a UART peripheral.

See `bp_uart_drv_en_t` for usage details.

*Prototype*

```
int  bp_zynq_uart_en  ( bp_uart_drv_hndl_t  hndl,
                        uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART peripheral to enable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_uart_is_en()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Returns the enabled/disabled state of a UART peripheral.

See `bp_uart_drv_is_en_t` for usage details.

*Prototype*

```
int  bp_zynq_uart_is_en  ( bp_uart_drv_hndl_t  hndl,
                           bool *              p_is_en );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART peripheral to check. |
| `p_is_en` | Interface state, true if enabled false otherwise. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_zynq_uart_mode_get()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Driver specific function to get the Zynq UART mode. See `bp_zynq_uart_mode_t` for a list of available modes.

`bp_zynq_uart_mode_get()` is a driver specific API. See `bp_uart_drv_hndl_get()` for information on how to call driver specific functions.

*Prototype*

```
int  bp_zynq_uart_mode_get  ( bp_uart_drv_hndl_t     hndl,
                              bp_zynq_uart_mode_t *  p_mode );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART interface to query. |
| `p_mode` | Pointer to the returned mode. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_zynq_uart_mode_set()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Driver specific function to set the Zynq UART mode. See `bp_zynq_uart_mode_t` for a list of available modes.

`bp_zynq_uart_mode_set()` is a driver specific API. See `bp_uart_drv_hndl_get()` for information on how to call driver specific functions.

*Prototype*

```
int  bp_zynq_uart_mode_set  ( bp_uart_drv_hndl_t   hndl,
                              bp_zynq_uart_mode_t  mode );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART interface to set. |
| `mode` | Selected mode. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_uart_reset()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Resets a UART peripheral.

See `bp_uart_drv_reset_t` for usage details.

*Prototype*

```
int  bp_zynq_uart_reset  ( bp_uart_drv_hndl_t  hndl,
                           uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART peripheral to reset. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_uart_rx()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Receives data.

See `bp_uart_drv_rx_t` for usage details.

*Prototype*

```
int  bp_zynq_uart_rx  (           hndl,
                       void *     p_buf,
                       size_t     len,
                       size_t *   p_rx_len,
                       uint32_t   timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to use for reception. |
| `p_buf` | Pointer to the buffer that will receive the data. |
| `len` | Length of the data to receive in bytes. |
| `p_rx_len` | Return pointer of the actual number of bytes read, can be NULL. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

## bp_zynq_uart_rx_async()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Receive data asynchronously.

See `bp_uart_drv_rx_async_t` for usage details.

*Prototype*

```
int  bp_zynq_uart_rx_async  (                hndl,
                             bp_uart_tf_t *  p_tf,
                             uint32_t        timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to use for reception. |
| `p_tf` | Transfer parameters. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_uart_rx_async_abort()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Aborts an asynchronous transfer.

See `bp_uart_drv_rx_async_abort_t` for usage details.

*Prototype*

```
int  bp_zynq_uart_rx_async_abort  (          hndl,
                                   size_t *  p_rx_len,
                                   uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to abort. |
| `p_rx_len` | Pointer to the number of bytes received, can be NULL. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_uart_rx_flush()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Flush the transmit path.

See `bp_uart_drv_rx_flush_t` for usage details.

*Prototype*

```
int  bp_zynq_uart_rx_flush  (          hndl,
                              uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to flush. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_uart_rx_idle_wait()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Waits for a UART interface to be idle.

See bp_uart_drv_rx_idle_wait_t for usage details.

*Prototype*

```
int  bp_zynq_uart_rx_idle_wait  (          hndl,
                                  uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the interface to wait. |
| timeout_ms | Timeout in milliseconds. |

*Returned Errors*

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_zynq_uart_tx()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Transmits data.

See bp_uart_drv_tx_t for usage details.

*Prototype*

```
int  bp_zynq_uart_tx  (             hndl,
                        const void *  p_buf,
                        size_t        len,
                        uint32_t      timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the interface to use for transmission. |
| p_buf | Pointer to the buffer to transmit. |
| len | Length of the data to transmit in bytes. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_IO_ERR
RTNC_FATAL

Function

# bp_zynq_uart_tx_async()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Transmits data asynchronously.

See bp_uart_drv_tx_async_t for usage details.

*Prototype*

```
int  bp_zynq_uart_tx_async  (                hndl,
                             bp_uart_tf_t *  p_tf,
                             uint32_t        timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the interface to use for reception. |
| p_tf | Transfer parameters. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_zynq_uart_tx_async_abort()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Aborts an asynchronous transfer.

See bp_uart_drv_tx_async_abort_t for usage details.

*Prototype*

```
int  bp_zynq_uart_tx_async_abort  (           hndl,
                                   size_t *   p_tx_len,
                                   uint32_t   timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the interface to abort. |
| p_tx_len | Pointer to the number of bytes transmitted, can be NULL. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

## bp_zynq_uart_tx_flush()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Flush the receive path.

See bp_uart_drv_tx_flush_t for usage details.

*Prototype*
```
int  bp_zynq_uart_tx_flush  (          hndl,
                              uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the interface to flush. |
| timeout_ms | Timeout in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

## bp_zynq_uart_tx_idle_wait()

<uart/drv/zynq/bp_zynq_uart_drv.h>

Waits for a UART interface to be idle.

See bp_uart_drv_tx_idle_wait_t for usage details.

*Prototype*
```
int  bp_zynq_uart_tx_idle_wait  (          hndl,
                                  uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the interface to wait. |
| timeout_ms | Timeout in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Data Type

# bp_zynq_uart_mode_t

<uart/drv/zynq/bp_zynq_uart_drv.h>

Zynq UART mode. See bp_zynq_uart_mode_set() and bp_zynq_uart_mode_get() for usage details.

*Values*

| | |
|---|---|
| BP_ZYNQ_UART_MODE_NORMAL | Normal operation. |
| BP_ZYNQ_UART_MODE_ECHO | Automatic echoing, sent characters will be echoed to the receive path. |
| BP_ZYNQ_UART_MODE_LOOPBACK | Local loopback rx and tx are tied together. |
| BP_ZYNQ_UART_MODE_REMOTE_LOOPBACK | Remote loopback, external rx and tx lines are tied together. |

Data Type

# bp_zynq_uart_drv_def_t

<uart/drv/zynq/bp_zynq_uart_drv.h>

Zynq UART driver hardware definition structure. Those parameters are required by the UART driver and are configured through a bp_uart_soc_def_t structure.

*Members*

| | | |
|---|---|---|
| base_addr | void * | Peripheral base address. |
| int_id | int | Peripheral interrupt id. |
| clk_id | int | Peripheral clock id. |
| reset_id | int | Peripheral reset id. |

Chapter

13

# Zynq-7000 I2C Driver

I2C driver for the Xilinx Zynq-7000. Note that for most applications it is recommended to use the I2C module API instead of the driver interface. This module contains the I2C driver interface to be used by the I2C module as well as additional driver specific functions. The driver interface as well as the driver specific functions can be called by the application using the driver handle which can be retrieved using the `bp_i2c_drv_hndl_get()` function.

See the BASEplatform manual for additional information on calling the driver interface directly.

Function

## `bp_zynq_i2c_cfg_get()`

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Retrieves the current configuration of an I2C interface.

See `bp_i2c_drv_cfg_get_t` for usage details.

*Prototype*

```
int  bp_zynq_i2c_cfg_get  ( bp_i2c_drv_hndl_t  hndl,
                            bp_i2c_cfg_t *     p_cfg,
                            uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to query. |
| `p_cfg` | Pointer to the I2C configuration. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_zynq_i2c_cfg_set()

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Configures an I2C interface.

See bp_i2c_drv_cfg_set_t for usage details.

*Prototype*

```
int  bp_zynq_i2c_cfg_set  ( bp_i2c_drv_hndl_t    hndl,
                            const bp_i2c_cfg_t *  p_cfg,
                            uint32_t              timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the I2C driver to configure. |
| p_cfg | I2C configuration. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_NOT_SUPPORTED
RTNC_FATAL

Function

# bp_zynq_i2c_create()

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Creates an I2C driver instance.

See bp_i2c_drv_create_t for usage details.

*Prototype*

```
int  bp_zynq_i2c_create  ( const bp_i2c_board_def_t *  p_def,
                           bp_i2c_drv_hndl_t *         p_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `p_def` | Board definition of the I2C peripheral to create. |
| `p_hndl` | Pointer to the newly created I2C driver interface. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

## bp_zynq_i2c_destroy()

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Destroys an I2C interface.

See `bp_i2c_drv_destroy_t` for usage details.

*Prototype*

```
int  bp_zynq_i2c_destroy  ( bp_i2c_drv_hndl_t  hndl,
                            uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to destroy. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_i2c_dis()

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Disables an I2C interface.

See `bp_i2c_drv_dis_t` for usage details.

*Prototype*

```
int  bp_zynq_i2c_dis  ( bp_i2c_drv_hndl_t  hndl,
                        uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C interface to disable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_i2c_en()

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Enables an I2C interface.

See `bp_i2c_drv_en_t` for usage details.

*Prototype*

```
int  bp_zynq_i2c_en  ( bp_i2c_drv_hndl_t  hndl,
                       uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to enable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_i2c_flush()

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Flush the transmit and receive paths.

See `bp_i2c_drv_flush_t` for usage details.

*Prototype*

```
int  bp_zynq_i2c_flush  ( bp_i2c_drv_hndl_t  hndl,
                          uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to flush. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_i2c_idle_wait()

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Waits for an I2C interface to be idle.

See `bp_i2c_drv_idle_wait_t` for usage details.

*Prototype*

```
int  bp_zynq_i2c_idle_wait  ( bp_i2c_drv_hndl_t  hndl,
                              uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to wait on. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_i2c_is_en()

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Returns the enabled/disabled state of an I2C interface.

See `bp_i2c_drv_is_en_t` for usage details.

*Prototype*

```
int  bp_zynq_i2c_is_en  ( bp_i2c_drv_hndl_t  hndl,
                          bool *             p_is_en );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to query. |
| `p_is_en` | Interface state, true if enabled false otherwise. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_zynq_i2c_reset()

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Resets an I2C interface.

See `bp_i2c_drv_reset_t` for usage details.

*Prototype*

```
int  bp_zynq_i2c_reset  ( bp_i2c_drv_hndl_t  hndl,
                          uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to reset. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_i2c_xfer()

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Performs an I2C operation.

See `bp_i2c_drv_xfer_t` for usage details.

*Prototype*

```
int  bp_zynq_i2c_xfer  ( bp_i2c_drv_hndl_t  hndl,
                         bp_i2c_tf_t *      p_tf,
                                            p_recv_len,
                         uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to use. |
| `p_tf` | Pointer to an `bp_i2c_tf_t` structure describing the transfer to perform. |
| `p_recv_len` | Amount of data actually received. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

# bp_zynq_i2c_xfer_async()

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Transfers data asynchronously.

See `bp_i2c_drv_xfer_async_t` for usage details.

*Prototype*

```
int  bp_zynq_i2c_xfer_async  ( bp_i2c_drv_hndl_t  hndl,
                               bp_i2c_tf_t *      p_tf,
                               uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to use for transferring. |
| `p_tf` | Transfer parameters. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_zynq_i2c_xfer_async_abort()

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Aborts an asynchronous transfer.

See `bp_i2c_drv_xfer_async_abort_t` for usage details.

*Prototype*

```
int  bp_zynq_i2c_xfer_async_abort  ( bp_i2c_drv_hndl_t  hndl,
                                     size_t *           p_tf_len,
                                     uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to abort. |
| `p_tf_len` | Amount of data transferred. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

# bp_zynq_i2c_drv_def_t

<i2c/drv/zynq/bp_zynq_i2c_drv.h>

Zynq I2C driver hardware definition structure. Those parameters are required by the I2C driver and are configured through a `bp_i2c_soc_def_t` structure.

*Members*

| | | |
|---|---|---|
| `base_addr` | `void *` | Peripheral base address. |
| `int_id` | `int` | Peripheral interrupt id. |
| `clk_id` | `int` | Peripheral clock id. |

Chapter

# 14 Zynq-7000 SPI Driver

SPI driver for the Xilinx Zynq-7000. Note that for most applications it is recommended to use the SPI module API instead of the driver interface. This module contains the SPI driver interface to be used by the SPI module as well as additional driver specific functions. The driver interface as well as the driver specific functions can be called by the application using the driver handle which can be retrieved using the `bp_spi_drv_hndl_get()` function.

See the BASEplatform manual for additional information on calling the driver interface directly.

Function

## `bp_zynq_spi_cfg_get()`

<spi/drv/zynq/bp_zynq_spi_drv.h>

Retrieves the current configuration of an SPI peripheral.

See `bp_spi_drv_cfg_get_t` for usage details.

The configuration get procedure for this driver is non-blocking, as such the timeout_ms argument is ignored.

*Prototype*

```
int  bp_zynq_spi_cfg_get  (                hndl,
                           bp_spi_cfg_t *  p_cfg,
                           uint32_t        timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI peripheral to query. |
| `p_cfg` | Pointer to the SPI configuration. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Function

# bp_zynq_spi_cfg_set()

<spi/drv/zynq/bp_zynq_spi_drv.h>

Configures an SPI peripheral.

See bp_spi_drv_cfg_set_t for usage details.

*Prototype*

```
int  bp_zynq_spi_cfg_set  (                      hndl,
                           const bp_spi_cfg_t *  p_cfg,
                           uint32_t              timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the SPI driver to configure. |
| p_cfg | SPI configuration. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_NOT_SUPPORTED
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_zynq_spi_create()

<spi/drv/zynq/bp_zynq_spi_drv.h>

Creates an SPI driver instance.

See bp_spi_drv_driver_t for usage details.

*Prototype*

```
int  bp_zynq_spi_create  ( const bp_spi_board_def_t *  p_def,
                           bp_spi_drv_hndl_t *         p_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `p_def` | Board definition of the SPI driver to create. |
| `p_hndl` | Pointer to the newly created SPI interface. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

# bp_zynq_spi_destroy()

<spi/drv/zynq/bp_zynq_spi_drv.h>

Destroys an SPI driver instance.

See `bp_spi_drv_destroy_t` for usage details.

*Prototype*

```
int  bp_zynq_spi_destroy  (          hndl,
                           uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to destroy. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_zynq_spi_dis()

<spi/drv/zynq/bp_zynq_spi_drv.h>

Disables an SPI peripheral.

See `bp_spi_drv_dis_t` for usage details.

*Prototype*

```
int  bp_zynq_spi_dis  (          hndl,
                       uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to disable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_zynq_spi_en()

<spi/drv/zynq/bp_zynq_spi_drv.h>

Enables an SPI peripheral

See `bp_spi_drv_en_t` for usage details.

*Prototype*

```
int  bp_zynq_spi_en  (          hndl,
                       uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to enable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_zynq_spi_flush()

<spi/drv/zynq/bp_zynq_spi_drv.h>

Flush the transmit and receive paths.

See `bp_spi_drv_flush_` for usage details.

*Prototype*

```
int  bp_zynq_spi_flush  (          hndl,
                          uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to flush. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_spi_idle_wait()

<spi/drv/zynq/bp_zynq_spi_drv.h>

Wait for an SPI peripheral to be idle.

See `bp_spi_drv_idle_wait_t` for usage details.

*Prototype*

```
int  bp_zynq_spi_idle_wait  (          hndl,
                              uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the peripheral driver to wait on. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_zynq_spi_is_en()

<spi/drv/zynq/bp_zynq_spi_drv.h>

Returns the enabled/disabled state of an SPI peripheral.

See `bp_spi_drv_is_en_t` for usage details.

*Prototype*

```
int  bp_zynq_spi_is_en  (        hndl,
                          bool *  p_is_en );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to query. |
| `p_is_en` | Pointer to the returned state. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## `bp_zynq_spi_reset()`

<spi/drv/zynq/bp_zynq_spi_drv.h>

Resets an SPI peripheral.

See `bp_spi_drv_reset_t` for usage details.

The reset procedure for this driver is non-blocking, as such the `timeout_ms` argument is ignored.

*Prototype*

```
int  bp_zynq_spi_reset  (          hndl,
                          uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to reset. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## `bp_zynq_spi_slave_desel()`

<spi/drv/zynq/bp_zynq_spi_drv.h>

Deselect a selected SPI slave.

See `bp_spi_drv_slave_desel_t` for usage details.

*Prototype*

```
int  bp_zynq_spi_slave_desel  (          hndl,
                                uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_zynq_spi_slave_sel()

<spi/drv/zynq/bp_zynq_spi_drv.h>

Select a specific SPI slave.

See `bp_spi_drv_slave_sel_t` for usage details.

*Prototype*

```
int  bp_zynq_spi_slave_sel  (          hndl,
                             uint32_t  ss_id,
                             uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI peripheral. |
| `ss_id` | Numeric id of the slave select line to assert. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_zynq_spi_xfer()

<spi/drv/zynq/bp_zynq_spi_drv.h>

Performs an SPI operation.

See `bp_spi_drv_xfer_t` for usage details.

*Prototype*

```
int  bp_zynq_spi_xfer  (               hndl,
                        bp_spi_tf_t *  p_tf,
                                       p_recv_len,
                        uint32_t       timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to use. |
| `p_tf` | Pointer to a `bp_spi_tf_t` structure describing the transfer to perform. |
| `p_recv_len` | Amount of data actually received. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

# bp_zynq_spi_xfer_async()

<spi/drv/zynq/bp_zynq_spi_drv.h>

Transfer data asynchronously.

See `bp_spi_drv_xfer_async_t` for usage details.

*Prototype*

```
int  bp_zynq_spi_xfer_async  (              hndl,
                               bp_spi_tf_t * p_tf,
                               uint32_t      timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to use for transferring. |
| `p_tf` | Transfer parameters. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_zynq_spi_xfer_async_abort()

<spi/drv/zynq/bp_zynq_spi_drv.h>

Aborts an asynchronous transfer.

See `bp_spi_drv_xfer_async_abort_` for usage details.

*Prototype*

```
int  bp_zynq_spi_xfer_async_abort  (          hndl,
                                    size_t *  p_tx_len,
                                    size_t *  p_rx_len,
                                    uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to abort. |
| `p_tx_len` | Pointer to the amount of data already transferred. |
| `p_rx_len` | Pointer to the amount of data already received. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

# bp_zynq_spi_drv_def_t

<spi/drv/zynq/bp_zynq_spi_drv.h>

Zynq SPI driver hardware definition structure. Those parameters are required by the SPI driver and are configured through a `bp_spi_soc_def_t` structure.

*Members*

| | | |
|---|---|---|
| `base_addr` | `void *` | Peripheral base address. |
| `int_id` | `int` | Peripheral interrupt id. |
| `clk_id` | `int` | Peripheral clock id. |

Chapter

15

# Xilinx AXI Timer

Interface module to the Xilinx AXI Timer soft IP. This module can also optionally provide the BASEplatform time base implementation for a MicroBlaze system.

Function

## bp_xil_axi_timer_cfg_get()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Retrieves the configuration of timer index `timer_ix` of a timer module instance. The configuration is read from the timer hardware registers and returned through `p_cfg`.

*Prototype*

```
int  bp_xil_axi_timer_cfg_get  ( bp_xil_axi_timer_hndl_t   timer_hndl,
                                 uint32_t                  timer_ix,
                                 bp_xil_axi_timer_cfg_t *  p_cfg );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `timer_ix` | Timer index to query. |
| `p_cfg` | Pointer to the configuration to the returned configuration. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_xil_axi_timer_cfg_set()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Configures timer index `timer_ix` of a timer module instance with configuration `p_cfg`. Note that if the timer is started the configuration will be applied while the timer is running which may cause undesired side-effects. Prior to starting the timer, `bp_axi_axi_timer_reload_set()` can be called to set the timer reload value.

*Prototype*

```
int  bp_xil_axi_timer_cfg_set  ( bp_xil_axi_timer_hndl_t   timer_hndl,
                                 uint32_t                  timer_ix,
                                 bp_xil_axi_timer_cfg_t *  p_cfg );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `timer_ix` | Timer index to configure. |
| `p_cfg` | Pointer to the configuration to set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_xil_axi_timer_create()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Creates an AXI Timer module instance. The created instance is associated with the peripheral board definition `p_def`. If successful, a handle to the newly created instance is returned through the `p_hndl` argument.

The definition structure pointed to by `p_def` must be unique and can only be associated with a single timer instance.

A timer cannot be opened more than once. If an attempt is made to open the same interface twice, `bp_xil_axi_timer_create()` returns an `RTNC_ALREADY_EXIST` error without affecting the already opened interface.

The board definition `p_def` passed to `bp_xil_axi_timer_create()` must be kept valid for the lifetime of the timer module instance.

When `bp_xil_axi_timer_create()` returns with either an `RTNC_NO_RESOURCE` or `RTNC_ALREADY_EXIST` error, the destination of `p_hndl` is left in an undefined state.

*Prototype*

```
int  bp_xil_axi_timer_create  ( bp_xil_axi_timer_board_def_t *  p_def,
                                bp_xil_axi_timer_hndl_t *       p_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `p_def` | Definition of the timer peripheral. |
| `p_hndl` | Pointer to the created timer module instance. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_ALREADY_EXIST`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

# bp_xil_axi_timer_read()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Returns the current timer counter value of timer index `timer_ix` of timer instance `timer_hndl`.

*Prototype*

```
int  bp_xil_axi_timer_read  ( bp_xil_axi_timer_hndl_t  timer_hndl,
                              uint32_t                 timer_ix,
                              uint32_t *               p_val );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `timer_ix` | Timer index to read. |
| `p_val` | Pointer to the returned timer value. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_xil_axi_timer_read64()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Returns the current combined timer counter value of timers of timer instance `timer_hndl`. The timers will be read and the value reported as if the timer were chained whether or not this is the case.

*Prototype*

```
int  bp_xil_axi_timer_read64  ( bp_xil_axi_timer_hndl_t  timer_hndl,
                                uint64_t *               p_val );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `p_val` | Pointer to the returned timer value. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_xil_axi_timer_reload_get()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Gets the reload value of timer index `timer_ix` of a timer module instance.

*Prototype*

```
int  bp_xil_axi_timer_reload_get  ( bp_xil_axi_timer_hndl_t  timer_hndl,
                                    uint32_t                 timer_ix,
                                    uint32_t *               p_reload_val );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `timer_ix` | Timer index to query. |
| `p_reload_val` | Pointer to the returned reload value. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_xil_axi_timer_reload_get64()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Gets the reload value of a timer module instance as if the two timers were cascaded. The reload value of both timers will be read whether or not the timers are actually cascaded.

*Prototype*

```
int  bp_xil_axi_timer_reload_get64  ( bp_xil_axi_timer_hndl_t  timer_hndl,
                                      uint32_t *               p_reload_val );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `p_reload_val` | Pointer to the returned reload value. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_xil_axi_timer_reload_set()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Sets the reload value of timer index `timer_ix` of a timer module instance.

*Prototype*

```
int  bp_xil_axi_timer_reload_set  ( bp_xil_axi_timer_hndl_t  timer_hndl,
                                    uint32_t                 timer_ix,
                                    uint32_t                 reload_val );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `timer_ix` | Timer index to set. |
| `reload_val` | Reload value to set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_xil_axi_timer_reload_set64()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Sets the reload value of a timer module instance as if the timer were cascaded. The reload value of both timers will be set whether or not the timers are actually cascaded.

*Prototype*

```
int  bp_xil_axi_timer_reload_set64  ( bp_xil_axi_timer_hndl_t  timer_hndl,
                                      uint32_t                 reload_val );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `reload_val` | Reload value to set. |

*Returned Errors* RTNC_SUCCESS
RTNC_FATAL

Function

## bp_xil_axi_timer_start()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Starts timer index `timer_ix` of a timer module instance. The timer reload value will be loaded and started. If the timer is already started, it will be stopped and restarted with the currently configured reload value.

*Prototype*

```
int  bp_xil_axi_timer_start  ( bp_xil_axi_timer_hndl_t  timer_hndl,
                               uint32_t                 timer_ix );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `timer_ix` | Timer index to start. |

*Returned Errors* RTNC_SUCCESS
RTNC_FATAL

Function

## bp_xil_axi_timer_start64()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Reload and starts both timers of timer instance `timer_hndl` together. Both timer will be started whether or not they are actually cascaded. The reload value can be set by calling `bp_xil_axi_timer_reload_val_set64()`.

*Prototype*

```
int  bp_xil_axi_timer_start64  ( bp_xil_axi_timer_hndl_t  timer_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |

*Returned Errors* RTNC_SUCCESS
RTNC_FATAL

Function

# bp_xil_axi_timer_stop()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Stops timer index `timer_ix` of a timer module instance. If the timer wasn't started, nothing is done and `RTNC_SUCCESS` is returned.

*Prototype*

```
int  bp_xil_axi_timer_stop  ( bp_xil_axi_timer_hndl_t  timer_hndl,
                              uint32_t                 timer_ix );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `timer_ix` | Timer index to stop. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_xil_axi_timer_stop64()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Stops the timers of a timer module instance. If the timers weren't started, nothing is done and `RTNC_SUCCESS` is returned. If the timers weren't cascaded, they will be stopped nonetheless.

*Prototype*

```
int  bp_xil_axi_timer_stop64  ( bp_xil_axi_timer_hndl_t  timer_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

# bp_xil_axi_timer_board_def_t

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

AXI Timer module board level hardware definition structure. Used to set the name of a timer instance.

See bp_xil_axi_timer_create() for details.

*Members*

| | | |
|---|---|---|
| p_soc_def | const bp_xil_axi_timer_soc_def_t * | SoC level hardware definition. |
| p_name | const char * | Peripheral name. |

Data Type

## bp_xil_axi_timer_cfg_t

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

AXI Timer configuration structure. Used with bp_xil_axi_timer_cfg_set() and bp_axi_axi_timer_cfg_get().

*Members*

| | | |
|---|---|---|
| cascade | bool | Enable cascade mode. |
| pwm_en | bool | Enable pulse width modulation. |
| int_en | bool | Enable interrupt. |
| reload | bool | Enable auto-reload. |
| capt | bool | Enable capture. |
| gen_sig | bool | Enable external signal. |
| down | bool | Enable down counter mode. |

Data Type

## bp_xil_axi_timer_hndl_t

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

AXI timer handle. Returned by bp_xil_axi_timer_create(). The pointer contained in the handle is private and should not be accessed by calling code.

*Members*

| | | |
|---|---|---|
| p_hndl | bp_xil_axi_timer_inst_t * | Pointer to internal module data. |

Data Type

## bp_xil_axi_timer_inst_t

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Data Type

## bp_xil_axi_timer_soc_def_t

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

AXI Timer module SoC level hardware definition structure.

The hardware definition structure is used to describe the peripheral at the SoC level. This structure is used with the `bp_xil_axi_timer_board_def_t` board definition structure to describe a complete GPTIMER instance.

See bp_xil_axi_timer_create() for details.

*Members*

| | | |
|---|---|---|
| `base_addr` | `void *` | Peripheral base address. |
| `int_id` | `uint32_t` | Peripheral base interrupt id. |
| `timer_cnt` | `uint32_t` | Number of implemented timers. |
| `casc_support` | `bool` | 64-bit mode(cascaded) support. |

Macro

## BP_UART_HNDL_IS_NULL()

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

Evaluates if an AXI Timer module handle is NULL.

*Prototype* `BP_UART_HNDL_IS_NULL ( hndl );`

*Parameters* `hndl` Handle to be checked.

*Expansion* `true` if the handle is NULL, `false` otherwise.

Macro

## BP_XIL_AXI_TIMER_NULL_HNDL

<soc_comp/xilinx/axi_timer/bp_xil_axi_timer.h>

NULL AXI Timer handle.

Chapter

16

# Xilinx AXI GPIO Driver

GPIO driver for the Xilinx AXI GPIO Soft IP. Note that for most applications it is recommended to use the GPIO module API instead of the driver interface. This module contains the GPIO driver interface to be used by the GPIO module as well as additional driver specific functions. The driver interface as well as the driver specific functions can be called by the application using the driver handle which can be retrieved using the `bp_gpio_drv_hndl_get()` function.

See the BASEplatform manual for additional information on calling the driver interface directly.

Function

## bp_xil_axi_gpio_create()

<soc_comp/xilinx/axi_gpio/bp_xil_axi_gpio_drv.h>

Creates a GPIO driver instance.

See `bp_gpio_drv_create_t` for usage details.

*Prototype*

```
int  bp_xil_axi_gpio_create  ( const bp_gpio_board_def_t *  p_def,
                               bp_gpio_drv_hndl_t *         p_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `p_def` | Board definition of the GPIO peripheral to create. |
| `p_hndl` | Handle to the created GPIO driver instance. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

# bp_xil_axi_gpio_data_get()

<soc_comp/xilinx/axi_gpio/bp_xil_axi_gpio_drv.h>

Gets the state of a GPIO pin.

See bp_gpio_drv_data_get_t for usage details.

*Prototype*

```
int  bp_xil_axi_gpio_data_get  ( bp_gpio_drv_hndl_t  hndl,
                                 uint32_t            bank,
                                 uint32_t            pin,
                                 uint32_t *          p_data );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the driver to query. |
| bank | Bank number of the pin to query. |
| pin | Pin number of the pin to query. |
| p_data | Pointer to the variable that will receive the data. |

*Returned Errors*

RTNC_SUCCESS
RTNC_FATAL

Function

# bp_xil_axi_gpio_data_set()

<soc_comp/xilinx/axi_gpio/bp_xil_axi_gpio_drv.h>

Sets the state of a GPIO pin.

See bp_gpio_drv_data_set_t for usage details.

*Prototype*

```
int  bp_xil_axi_gpio_data_set  ( bp_gpio_drv_hndl_t  hndl,
                                 uint32_t            bank,
                                 uint32_t            pin,
                                 data );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the driver to set. |
| bank | Bank number of the pin to set. |
| pin | Pin number of the pin to set. |
| data | State of the pin to set. |

*Returned Errors* `RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_xil_axi_gpio_data_tog()

<soc_comp/xilinx/axi_gpio/bp_xil_axi_gpio_drv.h>

Toggle the state of a GPIO pin.

See `bp_gpio_drv_data_tog_t` for usage details.

*Prototype*

```
int  bp_xil_axi_gpio_data_tog  ( bp_gpio_drv_hndl_t  hndl,
                                 uint32_t            bank,
                                 uint32_t            pin );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to toggle. |
| `bank` | Bank number of the pin to toggle. |
| `pin` | Pin number of the pin to toggle. |

*Returned Errors* `RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_xil_axi_gpio_destroy()

<soc_comp/xilinx/axi_gpio/bp_xil_axi_gpio_drv.h>

Destroys a GPIO driver instance.

See `bp_gpio_drv_destroy_t` for usage details.

*Prototype*

```
int  bp_xil_axi_gpio_destroy  ( bp_gpio_drv_hndl_t  hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the GPIO driver instance to destroy. |

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Function

# bp_xil_axi_gpio_dir_get()

<soc_comp/xilinx/axi_gpio/bp_xil_axi_gpio_drv.h>

Gets the direction of a GPIO pin.

See bp_gpio_drv_dir_get_t for usage details.

*Prototype*

```
int  bp_xil_axi_gpio_dir_get  ( bp_gpio_drv_hndl_t  hndl,
                                uint32_t            bank,
                                uint32_t            pin,
                                bp_gpio_dir_t *     p_dir );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the driver to query. |
| bank | Bank number of the pin to query. |
| pin | Pin number of the pin to query. |
| p_dir | Pointer to the variable that will receive the direction. |

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Function

# bp_xil_axi_gpio_dir_set()

<soc_comp/xilinx/axi_gpio/bp_xil_axi_gpio_drv.h>

Sets the direction of a GPIO pin.

See bp_gpio_drv_dir_set_t for usage details.

*Prototype*

```
int  bp_xil_axi_gpio_dir_set  ( bp_gpio_drv_hndl_t  hndl,
                                uint32_t            bank,
                                uint32_t            pin,
                                                    dir );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to set. |
| `bank` | Bank number of the pin to set. |
| `pin` | Pin number of the pin to set. |
| `dir` | Direction of the pin to set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_xil_axi_gpio_dis()

<soc_comp/xilinx/axi_gpio/bp_xil_axi_gpio_drv.h>

Disables a GPIO interface.

See `bp_gpio_drv_dis_t` for usage details.

*Prototype*

```
int  bp_xil_axi_gpio_dis  ( bp_gpio_drv_hndl_t  hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the GPIO driver to disable. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_xil_axi_gpio_en()

<soc_comp/xilinx/axi_gpio/bp_xil_axi_gpio_drv.h>

Disables a GPIO interface.

See `bp_gpio_drv_en_t` for usage details.

*Prototype*

```
int  bp_xil_axi_gpio_en  ( bp_gpio_drv_hndl_t  hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the GPIO driver to enable. |

*Returned Errors* RTNC_SUCCESS
RTNC_FATAL

Function

# bp_xil_axi_gpio_is_en()

<soc_comp/xilinx/axi_gpio/bp_xil_axi_gpio_drv.h>

Returns the enabled/disabled state of a GPIO interface.

See bp_gpio_drv_is_en_t for usage details.

*Prototype*

```
int  bp_xil_axi_gpio_is_en  ( bp_gpio_drv_hndl_t  hndl,
                              bool *              p_is_en );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the GPIO driver to query. |
| p_is_en | Interface state, true if enabled false otherwise. |

*Returned Errors* RTNC_SUCCESS
RTNC_FATAL

Data Type

# bp_xil_axi_gpio_drv_def_t

<soc_comp/xilinx/axi_gpio/bp_xil_axi_gpio_drv.h>

Xilinx AXI GPIO driver hardware definition structure. Those parameters are required by the GPIO driver and are configured through a bp_gpio_soc_def_t structure.

*Members*

| | | |
|---|---|---|
| base_addr | void * | Peripheral base address. |
| int_id | uint32_t | Peripheral interrupt id. |

Chapter

17

# Xilinx AXI UARTLite Driver

UART driver for the Xilinx AXI UARTLite soft ip. Note that for most applications it is recommended to use the UART module API instead of the driver interface. This module contains the UART driver interface to be used by the UART module as well as additional driver specific functions. The driver interface as well as the driver specific functions can be called by the application using the driver handle which can be retrieved using the `bp_uart_drv_hndl_get()` function.

See the BASEplatform manual for additional information on calling the driver interface directly.

Function

## `bp_xil_axi_uartlite_cfg_get()`

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Retrieves the current configuration of a UART interface.

See `bp_uart_drv_cfg_get_t` for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_cfg_get  ( bp_uart_drv_hndl_t  hndl,
                                    bp_uart_cfg_t *     p_cfg,
                                                        timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART peripheral to query. |
| `p_cfg` | Pointer to the UART configuration. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Function

# bp_xil_axi_uartlite_cfg_set()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Configures a UART peripheral.

See bp_uart_drv_cfg_set_t for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_cfg_set  ( bp_uart_drv_hndl_t    hndl,
                                    const bp_uart_cfg_t *  p_cfg,
                                    uint32_t               timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the UART peripheral to configure. |
| p_cfg | UART configuration. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_NOT_SUPPORTED
RTNC_FATAL

Function

# bp_xil_axi_uartlite_create()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Creates a UART driver instance.

See bp_uart_drv_create_t for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_create  (                      p_def,
                                   bp_uart_drv_hndl_t *  p_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `p_def` | Board definition of the UART peripheral to initialize. |
| `p_hndl` | Pointer to the newly created UART driver instance. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

## bp_xil_axi_uartlite_destroy()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Destroys a UART driver instance.

See `bp_uart_drv_destroy_t` for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_destroy  ( bp_uart_drv_hndl_t  hndl,
                                    uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART driver instance to destroy. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_xil_axi_uartlite_dis()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Disables a UART peripheral.

See `bp_uart_drv_dis_t` for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_dis  ( bp_uart_drv_hndl_t  hndl,
                                uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART peripheral to disable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_xil_axi_uartlite_en()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Enables a UART peripheral.

See `bp_uart_drv_en_t` for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_en  ( bp_uart_drv_hndl_t  hndl,
                               uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART peripheral to enable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_xil_axi_uartlite_is_en()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Returns the enabled/disabled state of a UART peripheral.

See `bp_uart_drv_is_en_t` for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_is_en  ( bp_uart_drv_hndl_t  hndl,
                                  bool *              p_is_en );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART peripheral to check. |
| `p_is_en` | Interface state, true if enabled false otherwise. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_xil_axi_uartlite_reset()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Resets a UART peripheral.

See `bp_uart_drv_reset_t` for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_reset  ( bp_uart_drv_hndl_t  hndl,
                                  uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART peripheral to reset. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_xil_axi_uartlite_rx()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Receives data.

See `bp_uart_drv_rx_t` for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_rx  ( bp_uart_drv_hndl_t  hndl,
                               void *              p_buf,
                               size_t              len,
                                                   p_rx_len,
                               uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to use for reception. |
| `p_buf` | Pointer to the buffer that will receive the data. |
| `len` | Length of the data to receive in bytes. |
| `p_rx_len` | Return pointer of the actual number of bytes read, can be NULL. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

# bp_xil_axi_uartlite_rx_async()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Receive data asynchronously.

See `bp_uart_drv_rx_async_t` for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_rx_async  ( bp_uart_drv_hndl_t  hndl,
                                     bp_uart_tf_t *      p_tf,
                                     uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to use for reception. |
| `p_tf` | Transfer parameters. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_xil_axi_uartlite_rx_async_abort()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Aborts an asynchronous transfer.

See `bp_uart_drv_rx_async_abort_t` for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_rx_async_abort  ( bp_uart_drv_hndl_t  hndl,
                                           size_t *            p_rx_len,
                                           uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to abort. |
| `p_rx_len` | Pointer to the number of bytes received, can be NULL. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_xil_axi_uartlite_rx_flush()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Flush the transmit path.

See `bp_uart_drv_rx_flush_t` for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_rx_flush  ( bp_uart_drv_hndl_t  hndl,
                                     uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to flush. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_xil_axi_uartlite_rx_idle_wait()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Waits for a UART interface to be idle.

See bp_uart_drv_rx_idle_wait_t for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_rx_idle_wait  ( bp_uart_drv_hndl_t  hndl,
                                         uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the interface to wait. |
| timeout_ms | Timeout in milliseconds. |

*Returned Errors*

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_xil_axi_uartlite_tx()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Transmits data.

See bp_uart_drv_tx_t for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_tx  ( bp_uart_drv_hndl_t  hndl,
                               const void *        p_buf,
                               size_t              len,
                               uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the interface to use for transmission. |
| p_buf | Pointer to the buffer to transmit. |
| len | Length of the data to transmit in bytes. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_IO_ERR
RTNC_FATAL

Function

# bp_xil_axi_uartlite_tx_async()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Transmits data asynchronously.

See bp_uart_drv_tx_async_t for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_tx_async  ( bp_uart_drv_hndl_t  hndl,
                                     bp_uart_tf_t *      p_tf,
                                     uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the interface to use for reception. |
| p_tf | Transfer parameters. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_xil_axi_uartlite_tx_async_abort()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Aborts an asynchronous transfer.

See bp_uart_drv_tx_async_abort_t for usage details.

*Prototype*

```
int  bp_xil_axi_uartlite_tx_async_abort  ( bp_uart_drv_hndl_t  hndl,
                                           size_t *            p_tx_len,
                                           uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the interface to abort. |
| p_tx_len | Pointer to the number of bytes transmitted, can be NULL. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_xil_axi_uartlite_tx_flush()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Flush the receive path.

See bp_uart_drv_tx_flush_t for usage details.

*Prototype*
```
int  bp_xil_axi_uartlite_tx_flush  ( bp_uart_drv_hndl_t  hndl,
                                     uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the interface to flush. |
| timeout_ms | Timeout in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_xil_axi_uartlite_tx_idle_wait()

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Waits for a UART interface to be idle.

See bp_uart_drv_tx_idle_wait_t for usage details.

*Prototype*
```
int  bp_xil_axi_uartlite_tx_idle_wait  ( bp_uart_drv_hndl_t  hndl,
                                         uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the interface to wait. |
| timeout_ms | Timeout in milliseconds. |

*Returned Errors*
`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

# bp_xil_axi_uartlite_drv_def_t

<soc_comp/xilinx/axi_uartlite/bp_xil_axi_uartlite_drv.h>

Xilinx AXI UARTLite driver hardware definition structure. Those parameters are required by the UART driver and are configured through a `bp_uart_soc_def_t` structure.

Since the AXI UARTLite has a static configuration set a time of synthesis, it is necessary to pass that configuration to the driver.

*Members*

| | | |
|---|---|---|
| `base_addr` | `void *` | Peripheral base address. |
| `int_id` | `int` | Peripheral interrupt id. |
| `baud_rate` | `uint32_t` | Fixed baud rate. |
| `parity` | `uint8_t` | Non zero if parity is enabled. |
| `odd_parity` | `uint8_t` | Non zero if parity is odd. |

Chapter

18

# Xilinx AXI I2C Driver

I2C driver for the Xilinx AXI I2C soft ip. Note that for most applications it is recommended to use the I2C module API instead of the driver interface. This module contains the I2C driver interface to be used by the I2C module as well as additional driver specific functions. The driver interface as well as the driver specific functions can be called by the application using the driver handle which can be retrieved using the `bp_i2c_drv_hndl_get()` function.

See the BASEplatform manual for additional information on calling the driver interface directly.

Function

## `bp_xil_axi_i2c_cfg_get()`

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Retrieves the current configuration of an I2C interface.

See `bp_i2c_drv_cfg_get_t` for usage details.

*Prototype*

```
int  bp_xil_axi_i2c_cfg_get  ( bp_i2c_drv_hndl_t  hndl,
                               bp_i2c_cfg_t *     p_cfg,
                               uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to query. |
| `p_cfg` | Pointer to the I2C configuration. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_xil_axi_i2c_cfg_set()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Configures an I2C interface.

See bp_i2c_drv_cfg_set_t for usage details.

*Prototype*

```
int  bp_xil_axi_i2c_cfg_set  ( bp_i2c_drv_hndl_t    hndl,
                               const bp_i2c_cfg_t *  p_cfg,
                               uint32_t              timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the I2C driver to configure. |
| p_cfg | I2C configuration. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_NOT_SUPPORTED
RTNC_FATAL

Function

# bp_xil_axi_i2c_create()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Creates an I2C driver instance.

See bp_i2c_drv_create_t for usage details.

*Prototype*

```
int  bp_xil_axi_i2c_create  ( const bp_i2c_board_def_t *  p_def,
                              bp_i2c_drv_hndl_t *         p_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `p_def` | Board definition of the I2C peripheral to create. |
| `p_hndl` | Pointer to the newly created I2C driver interface. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

# bp_xil_axi_i2c_destroy()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Destroys an I2C interface.

See `bp_i2c_drv_destroy_t` for usage details.

*Prototype*

```
int  bp_xil_axi_i2c_destroy  ( bp_i2c_drv_hndl_t  hndl,
                               uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to destroy. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_xil_axi_i2c_dis()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Disables an I2C interface.

See `bp_i2c_drv_dis_t` for usage details.

*Prototype*

```
int  bp_xil_axi_i2c_dis  ( bp_i2c_drv_hndl_t  hndl,
                           uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C interface to disable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_xil_axi_i2c_en()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Enables an I2C interface.

See `bp_i2c_drv_en_t` for usage details.

*Prototype*

```
int  bp_xil_axi_i2c_en  ( bp_i2c_drv_hndl_t  hndl,
                          uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to enable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_xil_axi_i2c_flush()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Flush the transmit and receive paths.

See `bp_i2c_drv_flush_t` for usage details.

*Prototype*

```
int  bp_xil_axi_i2c_flush  ( bp_i2c_drv_hndl_t  hndl,
                             uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to flush. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_xil_axi_i2c_gpo_get()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Returns the output value of the general purpose outputs.

The exact width, as configured during synthesis, is unknown to the driver. The value of unimplemented bits returned through p_gpo are undefined.

This function is a driver specific API. See `bp_i2c_drv_hndl_get()` for information on how to call driver specific functions.

*Prototype*

```
int  bp_xil_axi_i2c_gpo_get  ( bp_i2c_drv_hndl_t  hndl,
                               uint32_t *         p_gpo );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to access. |
| `p_gpo` | Pointer to the returned pin value. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_xil_axi_i2c_gpo_set()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Sets the output value of the general purpose outputs.

The exact width, as configured during synthesis, is unknown to the driver. It is the caller's responsibility to ensure that only implemented bits are set.

This function is a driver specific API. See `bp_i2c_drv_hndl_get()` for information on how to call driver specific functions.

*Prototype*

```
int  bp_xil_axi_i2c_gpo_set  ( bp_i2c_drv_hndl_t  hndl,
                               uint32_t           gpo );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to access. |
| `gpo` | Values of the GPO pins to set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_xil_axi_i2c_idle_wait()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Waits for an I2C interface to be idle.

See `bp_i2c_drv_idle_wait_t` for usage details.

*Prototype*

```
int  bp_xil_axi_i2c_idle_wait  ( bp_i2c_drv_hndl_t  hndl,
                                 uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to wait on. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_xil_axi_i2c_is_en()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Returns the enabled/disabled state of an I2C interface.

See `bp_i2c_drv_is_en_t` for usage details.

*Prototype*

```
int  bp_xil_axi_i2c_is_en  ( bp_i2c_drv_hndl_t  hndl,
                             bool *             p_is_en );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to query. |
| `p_is_en` | Interface state, true if enabled false otherwise. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_xil_axi_i2c_param_get()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Returns the I2C timing parameters.

This function is a driver specific API. See `bp_i2c_drv_hndl_get()` for information on how to call driver specific functions.

*Prototype*

```
int  bp_xil_axi_i2c_param_get  ( bp_i2c_drv_hndl_t          hndl,
                                 bp_xil_axi_i2c_param_t *  p_param );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to access. |
| `p_param` | Pointer to the returned timing parameters. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_xil_axi_i2c_param_set()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Sets the I2C timing parameters.

This function is a driver specific API. See `bp_i2c_drv_hndl_get()` for information on how to call driver specific functions.

*Prototype*

```
int  bp_xil_axi_i2c_param_set  ( bp_i2c_drv_hndl_t         hndl,
                                 bp_xil_axi_i2c_param_t *  p_param );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to access. |
| `p_param` | Timing parameters. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_xil_axi_i2c_reset()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Resets an I2C interface.

See `bp_i2c_drv_reset_t` for usage details.

*Prototype*

```
int  bp_xil_axi_i2c_reset  ( bp_i2c_drv_hndl_t  hndl,
                             uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to reset. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_xil_axi_i2c_xfer()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Performs an I2C operation.

See `bp_i2c_drv_xfer_t` for usage details.

*Prototype*

```
int  bp_xil_axi_i2c_xfer  ( bp_i2c_drv_hndl_t  hndl,
                            bp_i2c_tf_t *      p_tf,
                                               p_recv_len,
                            uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to use. |
| `p_tf` | Pointer to an `bp_i2c_tf_t` structure describing the transfer to perform. |
| `p_recv_len` | Amount of data actually received. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

# bp_xil_axi_i2c_xfer_async()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Transfers data asynchronously.

See `bp_i2c_drv_xfer_async_t` for usage details.

*Prototype*

```
int  bp_xil_axi_i2c_xfer_async  ( bp_i2c_drv_hndl_t  hndl,
                                  bp_i2c_tf_t *      p_tf,
                                  uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to use for transferring. |
| `p_tf` | Transfer parameters. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_xil_axi_i2c_xfer_async_abort()

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Aborts an asynchronous transfer.

See bp_i2c_drv_xfer_async_abort_t for usage details.

*Prototype*

```
int  bp_xil_axi_i2c_xfer_async_abort  ( bp_i2c_drv_hndl_t  hndl,
                                        size_t *           p_tf_len,
                                        uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the driver to abort. |
| p_tf_len | Amount of data transferred. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Data Type

# bp_xil_axi_i2c_drv_def_t

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Xilinx AXI I2C driver hardware definition structure. Those parameters are required by the I2C driver and are configured through a bp_i2c_soc_def_t structure. Since the SCL frequency as well as 10-bit address support are set during synthesis those informations should be passed to the I2C driver.

*Members*

| | | |
|---|---|---|
| base_addr | void * | Peripheral base address. |
| int_id | int | Peripheral interrupt id. |
| scl_freq | uint32_t | Peripheral SCL clock frequency. |
| ten_bit_addr | uint32_t | True if configured for 10-bit addresses. |

Data Type

# bp_xil_axi_i2c_param_t

<soc_comp/xilinx/axi_iic/bp_xil_axi_i2c_drv.h>

Xilinx AXI I2C timing parameters structure. Used to get and set the I2C timing parameters. See `bp_xil_axi_i2c_param_set()` and `bp_xil_axi_i2c_param_get()` for usage details.

*Members*

| | | |
|---|---|---|
| `tsusta` | `uint32_t` | Repeated start setup time. |
| `tsusto` | `uint32_t` | Repeated stop setup time. |
| `thdsta` | `uint32_t` | Repeated start hold time. |
| `tsudat` | `uint32_t` | Data setup time. |
| `tbuf` | `uint32_t` | Bus free time. |
| `thigh` | `uint32_t` | SCL high period. |
| `tlow` | `uint32_t` | SCL low period. |
| `thddat` | `uint32_t` | Data hold time. |

Chapter

19

# Xilinx AXI SPI Driver

SPI driver for the Xilinx AXI SPI soft IP. Note that for most applications it is recommended to use the SPI module API instead of the driver interface. This module contains the SPI driver interface to be used by the SPI module as well as additional driver specific functions. The driver interface as well as the driver specific functions can be called by the application using the driver handle which can be retrieved using the `bp_spi_drv_hndl_get()` function.

See the BASEplatform manual for additional information on calling the driver interface directly.

Function

## `bp_xil_axi_spi_cfg_get()`

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Retrieves the current configuration of an SPI peripheral.

See `bp_spi_drv_cfg_get_t` for usage details.

The configuration get procedure for this driver is non-blocking, as such the timeout_ms argument is ignored.

*Prototype*

```
int  bp_xil_axi_spi_cfg_get  (                hndl,
                               bp_spi_cfg_t *  p_cfg,
                               uint32_t        timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to query. |
| `p_cfg` | Pointer to the SPI configuration. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Function

# bp_xil_axi_spi_cfg_set()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Configures an SPI peripheral.

See bp_spi_drv_cfg_set_t for usage details.

*Prototype*

```
int  bp_xil_axi_spi_cfg_set  (                      hndl,
                               const bp_spi_cfg_t *  p_cfg,
                               uint32_t              timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the SPI driver to configure. |
| p_cfg | SPI configuration. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_NOT_SUPPORTED
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_xil_axi_spi_create()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Creates an SPI driver instance.

See bp_spi_drv_create_t for usage details.

*Prototype*

```
int  bp_xil_axi_spi_create  ( const bp_spi_board_def_t *  p_def,
                              bp_spi_drv_hndl_t *         p_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `p_def` | Board definition of the SPI driver to create. |
| `p_hndl` | Pointer to the newly created SPI interface. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

# bp_xil_axi_spi_destroy()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Destroys an SPI driver instance.

See `bp_spi_drv_destroy_t` for usage details.

*Prototype*

```
int  bp_xil_axi_spi_destroy  (          hndl,
                               uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to destroy. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_xil_axi_spi_dis()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Disables an SPI peripheral.

See `bp_spi_drv_dis_t` for usage details.

*Prototype*

```
int  bp_xil_axi_spi_dis  (          hndl,
                           uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to disable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_xil_axi_spi_en()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Enables an SPI peripheral

See `bp_spi_drv_en_t` for usage details.

*Prototype*

```
int  bp_xil_axi_spi_en  (          hndl,
                          uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to enable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_xil_axi_spi_flush()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Flush the transmit and receive paths.

See `bp_spi_drv_flush_t` for usage details.

*Prototype*

```
int  bp_xil_axi_spi_flush  (          hndl,
                             uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the peripheral to flush. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_xil_axi_spi_idle_wait()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Wait for an SPI peripheral to be idle.

See `bp_spi_drv_idle_wait_t` for usage details.

*Prototype*

```
int  bp_xil_axi_spi_idle_wait  (          hndl,
                                 uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to wait on. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_xil_axi_spi_is_en()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Returns the enabled/disabled state of an SPI peripheral.

See `bp_spi_drv_is_en_t` for usage details.

*Prototype*

```
int  bp_xil_axi_spi_is_en  (  );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Returned Errors* `RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_xil_axi_spi_reset()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Resets an SPI peripheral.

See `bp_spi_drv_reset_t` for usage details.

The reset procedure for this driver is non-blocking, as such the `timeout_md` argument is ignored.

*Prototype*

```
int  bp_xil_axi_spi_reset  (          hndl,
                            uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✗ | ✓ | ✓ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to reset. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors* `RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_xil_axi_spi_slave_desel()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Deselect a selected SPI slave.

See `bp_spi_drv_slave_desel_t` for usage details.

*Prototype*

```
int  bp_xil_axi_spi_slave_desel  (          hndl,
                                  uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_xil_axi_spi_slave_sel()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Select a specific SPI slave.

See bp_spi_drv_slave_sel_t for usage details.

*Prototype*

```
int  bp_xil_axi_spi_slave_sel  (          hndl,
                                uint32_t  ss_id,
                                uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the SPI peripheral. |
| ss_id | Numeric id of the slave select line to assert. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_xil_axi_spi_xfer()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Performs an SPI operation.

See bp_spi_drv_xfer_t for usage details.

*Prototype*

```
int  bp_xil_axi_spi_xfer  (              hndl,
                           bp_spi_tf_t*  p_tf,
                                         p_recv_len,
                           uint32_t      timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to use. |
| `p_tf` | Pointer to a `bp_spi_tf_t` structure describing the transfer to perform. |
| `p_recv_len` | Amount of data actually received. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO_ERR`
`RTNC_FATAL`

Function

# bp_xil_axi_spi_xfer_async()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Transfer data asynchronously.

See `bp_spi_drv_xfer_async_t` for usage details.

*Prototype*

```
int  bp_xil_axi_spi_xfer_async  (               hndl,
                                 bp_spi_tf_t *  p_tf,
                                 uint32_t       timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to use for transferring. |
| `p_tf` | Pointer to a `bp_spi_tf_t` structure describing the transfer to perform. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_xil_axi_spi_xfer_async_abort()

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Aborts an asynchronous transfer.

See `bp_spi_drv_xfer_async_abort_t` for usage details.

*Prototype*

```
int  bp_xil_axi_spi_xfer_async_abort  (          hndl,
                                        size_t *  p_tx_len,
                                        size_t *  p_rx_len,
                                        uint32_t  timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to abort. |
| `p_tx_len` | Pointer to the amount of data already transferred. |
| `p_rx_len` | Pointer to the amount of data already received. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

# bp_xil_axi_spi_drv_def_t

<soc_comp/xilinx/axi_qspi/bp_xil_axi_spi_drv.h>

Xilinx AXI SPI driver hardware definition structure. Those parameters are required by the SPI driver and are configured through a `bp_spi_soc_def_t` structure. Since the clock frequency as well as the number of configured slave id are set during synthesis those informations should be passed to the SPI driver.

*Members*

| | | |
|---|---|---|
| `base_addr` | `void *` | Peripheral base address. |
| `int_id` | `int` | Peripheral interrupt id. |
| `clk_freq` | `uint32_t` | Peripheral clock frequency. |
| `slave_cnt` | `uint32_t` | Number of configured slave id. |

Chapter

20

# Error Codes

Generic return code definitions. The descriptions below are a general guideline to the meaning of each return code. Consult the API documentation for a detailed list and description of errors that can be returned by each API.

Unexpected error codes returned by any functions, including error codes outside of the range of defined error codes should be treated as a fatal error.

Macro

## RTNC_*

<util/rtnc.h>

Description Return codes.

| | |
|---|---|
| RTNC_SUCCESS | Function completed successfully. |
| RTNC_FATAL | Fatal error occurred. |
| RTNC_NO_RESOURCE | Resource allocation failure. |
| RTNC_IO_ERR | Transfer or peripheral operation failed. |
| RTNC_TIMEOUT | Function timed out. |
| RTNC_NOT_SUPPORTED | API, feature or configuration is not supported. |
| RTNC_NOT_FOUND | Requested object not found. |
| RTNC_ALREADY_EXIST | Object already created or allocated. |
| RTNC_ABORT | Operation aborted by software. |
| RTNC_INVALID_OP | Invalid operation. |
| RTNC_WANT_READ | Read operation requested. |
| RTNC_WANT_WRITE | Write operation requested. |

Chapter

21

# GPIO Driver Reference

The GPIO driver declarations found in this module serves as the basis of GPIO drivers usually used in combination with the GPIO module to access GPIO peripherals. All GPIO drivers are composed of a standard set of API expected by the GPIO module in addition to any number of implementation-specific functions. The driver specific functions can be used by the application to access advanced features of a GPIO peripheral not exposed through the standard API. Note that usage of those extended functionalities is non-portable contrary to the standard API. The GPIO module API function `bp_gpio_drv_hndl_get()` function can be used to retrieve the driver handle associated with a GPIO module instance, and can subsequently be used to call the driver directly. See the individual driver's documentation for details of the extended functions.

In addition to accessing extended functionalities, an application can access the driver standard API directly bypassing the GPIO module. This reduces the call overhead. Contrary to most types of drivers, the GPIO drivers are usually thread-safe by design while other drivers usually require the top-level modules mutexes to be thread-safe.

Finally, as yet another feature of the GPIO driver API, it can be invoked in a standalone fashion without a GPIO module instance. This reduces the RAM overhead of using a GPIO peripheral. In this case the driver create function is called directly by the application in a matter similar to `bp_gpio_create()` to instantiate the driver.

Data Type

## bp_gpio_drv_create_t

<gpio/bp_gpio_drv.h>

GPIO driver's create function.

*Prototype*

```
int  bp_gpio_drv_create_t  ( const bp_gpio_board_def_t *  p_def,
                             bp_gpio_drv_hndl_t *        p_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `p_def` | Board definition of the GPIO peripheral to create. |
| `p_hndl` | Handle to the created GPIO driver instance. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_ALREADY_EXIST`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Data Type

## bp_gpio_drv_data_get_t

<gpio/bp_gpio_drv.h>

GPIO driver's data_get function. Returns the data state of pin number `pin` of bank `bank`.

*Prototype*

```
int  bp_gpio_drv_data_get_t  ( bp_gpio_drv_hndl_t  hndl,
                               uint32_t            bank,
                               uint32_t            pin,
                               uint32_t *          data );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to query. |
| `bank` | Bank number of the pin to query. |
| `pin` | Pin number of the pin to query. |
| `data` | Pointer to the variable that will receive the data. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

## bp_gpio_drv_data_set_t

<gpio/bp_gpio_drv.h>

GPIO driver's data_set function. Set the state of pin number `pin` of bank `bank` to the data specified by `data`.

*Prototype*

```
int  bp_gpio_drv_data_set_t  ( bp_gpio_drv_hndl_t  hndl,
                               uint32_t            bank,
                               uint32_t            pin,
                               uint32_t            data );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to set. |
| `bank` | Bank number of the pin to set. |
| `pin` | Pin number of the pin to set. |
| `data` | State of the pin to set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

## bp_gpio_drv_data_tog_t

<gpio/bp_gpio_drv.h>

Toggle the state of a GPIO pin. Toggle the data of pin number `pin` of bank `bank`.

*Prototype*

```
int  bp_gpio_drv_data_tog_t  ( bp_gpio_drv_hndl_t  hndl,
                               uint32_t            bank,
                               uint32_t            pin );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to toggle. |
| `bank` | Bank number of the pin to toggle. |
| `pin` | Pin number of the pin to toggle. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

## bp_gpio_drv_destroy_t

<gpio/bp_gpio_drv.h>

GPIO driver's destroy function.

*Prototype*

```
int  bp_gpio_drv_destroy_t  ( bp_gpio_drv_hndl_t  hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the GPIO driver instance to destroy. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

## bp_gpio_drv_dir_get_t

<gpio/bp_gpio_drv.h>

GPIO driver'd dir_get function. Returns the direction of pin number `pin` of bank `bank`.

*Prototype*

```
int  bp_gpio_drv_dir_get_t  ( bp_gpio_drv_hndl_t  hndl,
                              uint32_t            bank,
                              uint32_t            pin,
                              bp_gpio_            dir );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to query. |
| `bank` | Bank number of the pin to query. |
| `pin` | Pin number of the pin to query. |
| `dir` | Pointer to the variable that will receive the direction. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

## bp_gpio_drv_dir_set_t

<gpio/bp_gpio_drv.h>

GPIO driver's dir_set function. Sets the direction of pin number `pin` of bank `bank` to the direction specified by `dir`.

*Prototype*

```
int  bp_gpio_drv_dir_set_t  ( bp_gpio_drv_hndl_t  hndl,
                              uint32_t            bank,
                              uint32_t            pin,
                              bp_gpio_dir_t       dir );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to set. |
| `bank` | Bank number of the pin to set. |
| `pin` | Pin number of the pin to set. |
| `dir` | Direction of the pin to set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

## bp_gpio_drv_dis_t

<gpio/bp_gpio_drv.h>

GPIO driver's disable function.

*Prototype*

```
int  bp_gpio_drv_dis_t  ( bp_gpio_drv_hndl_t  hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the GPIO driver instance to disable. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

## bp_gpio_drv_en_t

<gpio/bp_gpio_drv.h>

GPIO driver's enable function.

*Prototype*

```
int  bp_gpio_drv_en_t  ( bp_gpio_drv_hndl_t  hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

`hndl` Handle of the GPIO driver instance to enable.

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

## bp_gpio_drv_is_en_t

<gpio/bp_gpio_drv.h>

GPIO driver's is_en function.

*Prototype*

```
int  bp_gpio_drv_is_en_t  ( bp_gpio_drv_hndl_t  hndl,
                            bool *              p_is_en );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

`hndl` Handle of the GPIO driver instance to query.
`p_is_en` Driver state, true if enabled false otherwise.

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

## bp_gpio_drv_reset_t

<gpio/bp_gpio_drv.h>

GPIO driver's reset function.

*Prototype*

```
int  bp_gpio_drv_reset_t  ( bp_gpio_drv_hndl_t  hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

`hndl` Handle of the GPIO driver to reset.

*Returned Errors* `RTNC_SUCCESS`
`RTNC_FATAL`

Macro

## BP_GPIO_DRV_HNDL_IS_NULL()

<gpio/bp_gpio_drv.h>

Evaluates if a GPIO driver handle is NULL.

*Prototype* `BP_GPIO_DRV_HNDL_IS_NULL ( hndl );`

*Parameters* `hndl` Handle to be checked.

*Expansion* `true` if the handle is NULL, `false` otherwise.

Macro

## BP_GPIO_DRV_NULL_HNDL

<gpio/bp_gpio_drv.h>

NULL GPIO driver handle.

Chapter

22

# I2C Driver Reference

The I2C driver declarations found in this module serves as the basis of I2C drivers usually used in combination with the I2C module to access I2C peripherals. All I2C drivers are composed of a standard set of API expected by the I2C module in addition to any number of implementation specific functions. The driver specific functions can be used by the application to access advanced features of a I2C peripheral not exposed through the standard API. Note that usage of those extended functionalities is non-portable contrary to the standard API. The I2C module API function `bp_i2c_drv_hndl_get()` function can be used to retrieve the driver handle associated with a I2C module instance, and can subsequently be used to call the driver directly. See the individual driver's documentation for details of the extended functions.

In addition to accessing extended functionalities, an application can access the driver standard API directly bypassing the I2C module. This reduces the call overhead at the cost of thread-safety as bare driver functions are usually not thread-safe when called directly. If thread-safety is required while calling driver functions directly, it is possible to use `bp_i2c_acquire()` and `bp_i2c_release()` to lock the I2C module preventing it from being accessed by other threads.

Finally, as yet another feature of the I2C driver API, it can be invoked in a standalone fashion without a UART module instance. This reduces the RAM overhead of using an I2C peripheral by dropping the I2C module mutexes and internal data structures. In this case the driver create function is called directly by the application in a matter similar to `bp_i2c_create()` to instantiate the driver. In this case thread safety has to be managed by the application, either using external mutexes or by ensuring that only one thread accesses the I2C peripheral.

Data Type

## bp_i2c_drv_cfg_get_t

<i2c/bp_i2c_drv.h>

I2C driver's configuration get function.

*Prototype*

```
int  bp_i2c_drv_cfg_get_t  ( bp_i2c_drv_hndl_t  hndl,
                             bp_i2c_cfg_t *     p_cfg,
                             uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to query. |
| `p_cfg` | Pointer to the I2C configuration. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_i2c_drv_cfg_set_t

<i2c/bp_i2c_drv.h>

*Prototype*

```
int  bp_i2c_drv_cfg_set_t  ( bp_i2c_drv_hndl_t    hndl,
                             const bp_i2c_cfg_t *  p_cfg,
                             uint32_t              timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to configure. |
| `p_cfg` | I2C configuration. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_NOT_SUPPORTED`
`RTNC_FATAL`

Data Type

## bp_i2c_drv_create_t

<i2c/bp_i2c_drv.h>

I2C driver's open function.

*Prototype*

```
int  bp_i2c_drv_create_t  ( const bp_i2c_board_def_t *  p_def,
                            bp_i2c_drv_hndl_t *        p_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `p_def` | Board definition of the I2C driver to initialize. |
| `p_hndl` | Pointer to the newly created I2C interface. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_ALREADY_EXIST`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Data Type

## bp_i2c_drv_destroy_t

<i2c/bp_i2c_drv.h>

I2C driver's destroy function.

*Prototype*

```
int  bp_i2c_drv_destroy_t  ( bp_i2c_drv_hndl_t  hndl,
                             uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to enable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_i2c_drv_dis_t

<i2c/bp_i2c_drv.h>

I2C driver's disable function.

*Prototype*

```
int  bp_i2c_drv_dis_t  ( bp_i2c_drv_hndl_t  hndl,
                         uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to disable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_i2c_drv_en_t

<i2c/bp_i2c_drv.h>

I2C driver's enable function.

*Prototype*

```
int  bp_i2c_drv_en_t  ( bp_i2c_drv_hndl_t  hndl,
                        uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to enable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_i2c_drv_flush_t

<i2c/bp_i2c_drv.h>

I2C driver's flush function.

*Prototype*

```
int  bp_i2c_drv_flush_t  ( bp_i2c_drv_hndl_t  hndl,
                           uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to flush. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_i2c_drv_idle_wait_t

<i2c/bp_i2c_drv.h>

I2C driver's idle wait function.

*Prototype*

```
int  bp_i2c_drv_idle_wait_t  ( bp_i2c_drv_hndl_t  hndl,
                               uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to wait. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_i2c_drv_is_en_t

<i2c/bp_i2c_drv.h>

I2C driver is_en function.

*Prototype*

```
int  bp_i2c_drv_is_en_t  ( bp_i2c_drv_hndl_t  hndl,
                           bool *             p_is_en );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the I2C driver to query. |
| `p_is_en` | Interface state, true if enabled false otherwise. |

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Data Type

## bp_i2c_drv_reset_t

<i2c/bp_i2c_drv.h>

I2C drivers's reset function.

*Prototype*

```
int  bp_i2c_drv_reset_t  ( bp_i2c_drv_hndl_t  hndl,
                           uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the I2C driver to reset. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Data Type

## bp_i2c_drv_xfer_async_abort_t

<i2c/bp_i2c_drv.h>

I2C driver's asynchronous transfer abort function.

*Prototype*

```
int  bp_i2c_drv_xfer_async_abort_t  ( bp_i2c_drv_hndl_t  hndl,
                                      size_t *           p_tf_len,
                                      uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the interface to abort. |
| p_tf_len | Amount of data transferred. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Data Type

## bp_i2c_drv_xfer_async_t

<i2c/bp_i2c_drv.h>

I2C driver asynchronous transfer function.

*Prototype*

```
int  bp_i2c_drv_xfer_async_t  ( bp_i2c_drv_hndl_t  hndl,
                                bp_i2c_tf_t *      p_tf,
                                uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the driver to use for the transfer. |
| p_tf | Transfer parameters. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Data Type

## bp_i2c_drv_xfer_t

<i2c/bp_i2c_drv.h>

I2C driver's transfer function.

*Prototype*

```
int  bp_i2c_drv_xfer_t  ( bp_i2c_drv_hndl_t  hndl,
                          bp_i2c_tf_t *      p_tf,
                          size_t *           p_tf_len,
                          uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

| | | |
|---|---|---|
| *Parameters* | `hndl` | Handle of the interface to use. |
| | `p_tf` | Pointer to an `bp_i2c_tf_t` structure describing the transfer to perform. |
| | `p_tf_len` | |
| | `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*
`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO_ERR`
`RTNC_FATAL`

Macro

## BP_I2C_DRV_HNDL_IS_NULL()

<i2c/bp_i2c_drv.h>

Evaluates if an I2C driver handle is NULL.

*Prototype* `BP_I2C_DRV_HNDL_IS_NULL ( hndl );`

*Parameters* `hndl` Handle to be checked.

*Expansion* `true` if the handle is NULL, `false` otherwise.

Macro

## BP_I2C_DRV_NULL_HNDL

<i2c/bp_i2c_drv.h>

NULL I2C driver handle.

Chapter

23

# SPI Driver Reference

The SPI driver declarations found in this module serves as the basis of SPI drivers usually used in combination with the SPI module to access SPI peripherals. All SPI drivers are composed of a standard set of API expected by the SPI module in addition to any number of implementation-specific functions. The driver specific functions can be used by the application to access advanced features of a SPI peripheral not exposed through the standard API. Note that usage of those extended functionalities is non-portable contrary to the standard API. The SPI module API function `bp_spi_drv_hndl_get()` function can be used to retrieve the driver handle associated with a SPI module instance, and can subsequently be used to call the driver directly. See the individual driver's documentation for details of the extended functions.

In addition to accessing extended functionalities, an application can access the driver standard API directly bypassing the SPI module. This reduces the call overhead at the cost of thread-safety as bare driver functions are usually not thread-safe when called directly. If thread-safety is required while calling driver functions directly, it is possible to use `bp_spi_slave_sel()` and `bp_spi_slave_desel()` to lock the SPI module preventing it from being accessed by other threads.

Finally, as yet another feature of the SPI driver API, it can be invoked in a standalone fashion without a SPI module instance. This reduces the RAM overhead of using an SPI peripheral by dropping the SPI module mutexes and internal data structures. In this case the driver create function is called directly by the application in a matter similar to `bp_spi_create()` to instantiate the driver. In this case thread safety has to be managed by the application, either using external mutexes or by ensuring that only one thread accesses the SPI peripheral.

Data Type

## bp_spi_drv_cfg_get_t

<spi/bp_spi_drv.h>

SPI driver's cfg_get function.

*Prototype*

```
int  bp_spi_drv_cfg_get_t  ( bp_spi_drv_hndl_t  hndl,
                             bp_spi_cfg_t *     p_cfg,
                             uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to query. |
| `p_cfg` | Pointer to the SPI configuration. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

# bp_spi_drv_cfg_set_t

<spi/bp_spi_drv.h>

SPI driver's cfg_set function.

*Prototype*

```
int  bp_spi_drv_cfg_set_t  ( bp_spi_drv_hndl_t    hndl,
                             const bp_spi_cfg_t *  p_cfg,
                             uint32_t              timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to configure. |
| `p_cfg` | SPI configuration. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_NOT_SUPPORTED`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

# bp_spi_drv_create_t

<spi/bp_spi_drv.h>

SPI driver's create function.

*Prototype*

```
int  bp_spi_drv_create_t  ( const bp_spi_board_def_t *  p_def,
                            bp_spi_drv_hndl_t *        p_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| p_def | Board definition of the SPI peripheral to initialize. |
| p_hndl | Pointer to the newly created SPI driver instance. |

*Returned Errors*

RTNC_SUCCESS
RTNC_ALREADY_EXIST
RTNC_NO_RESOURCE
RTNC_FATAL

Data Type

# bp_spi_drv_destroy_t

<spi/bp_spi_drv.h>

SPI driver's destroy function.

*Prototype*

```
int  bp_spi_drv_destroy_t  ( bp_spi_drv_hndl_t  hndl,
                             uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the SPI driver to destroy. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Data Type

# bp_spi_drv_dis_t

<spi/bp_spi_drv.h>

SPI driver's disable function.

*Prototype*

```
int  bp_spi_drv_dis_t  ( bp_spi_drv_hndl_t  hndl,
                         uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to disable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

# bp_spi_drv_en_t

<spi/bp_spi_drv.h>

SPI driver's enable function.

*Prototype*

```
int  bp_spi_drv_en_t  ( bp_spi_drv_hndl_t  hndl,
                        uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to enable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

# bp_spi_drv_flush_t

<spi/bp_spi_drv.h>

SPI driver's flush function.

*Prototype*

```
int  bp_spi_drv_flush_t  ( bp_spi_drv_hndl_t  hndl,
                           uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to flush. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

# bp_spi_drv_idle_wait_t

<spi/bp_spi_drv.h>

SPI driver's idle wait function.

*Prototype*

```
int  bp_spi_drv_idle_wait_t  ( bp_spi_drv_hndl_t  hndl,
                               uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to wait. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

# bp_spi_drv_is_en_t

<spi/bp_spi_drv.h>

SPI driver's is_en function.

*Prototype*

```
int  bp_spi_drv_is_en_t  ( bp_spi_drv_hndl_t  hndl,
                           bool *             p_is_en );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI interface to check. |
| `p_is_en` | Interface state, true if enabled false otherwise. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

# bp_spi_drv_reset_t

<spi/bp_spi_drv.h>

SPI driver's reset function.

*Prototype*

```
int  bp_spi_drv_reset_t  ( bp_spi_drv_hndl_t  hndl,
                           uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI interface to reset. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

# bp_spi_drv_slave_desel_t

<spi/bp_spi_drv.h>

SPI driver's slave deselect function.

*Prototype*

```
int  bp_spi_drv_slave_desel_t  ( bp_spi_drv_hndl_t  hndl,
                                 uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to wait on. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_spi_drv_slave_sel_t

<spi/bp_spi_drv.h>

SPI driver'd slave select function.

*Prototype*

```
int  bp_spi_drv_slave_sel_t  ( bp_spi_drv_hndl_t  hndl,
                               uint32_t           ss_id,
                               uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the SPI driver to wait on. |
| `ss_id` | Numeric id of the slave select line to assert. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_spi_drv_xfer_async_abort_t

<spi/bp_spi_drv.h>

SPI driver's asynchronous transfer abort function.

*Prototype*

```
int  bp_spi_drv_xfer_async_abort_t  ( bp_spi_drv_hndl_t  hndl,
                                      size_t *          p_tx_len,
                                      size_t *          p_rx_len,
                                      uint32_t          timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to abort. |
| `p_tx_len` | Pointer to the amount of data already transferred. |
| `p_rx_len` | Pointer to the amount of data already received. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_spi_drv_xfer_async_t

<spi/bp_spi_drv.h>

SPI driver's asynchronous transfer function.

*Prototype*

```
int  bp_spi_drv_xfer_async_t  ( bp_spi_drv_hndl_t  hndl,
                                bp_spi_tf_t *      p_tf,
                                uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to use for the transfer. |
| `p_tf` | Transfer parameters. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

# bp_spi_drv_xfer_t

<spi/bp_spi_drv.h>

SPI driver's xfer function.

*Prototype*

```
int  bp_spi_drv_xfer_t  ( bp_spi_drv_hndl_t  hndl,
                          bp_spi_tf_t *      p_tf,
                          size_t *           p_tf_len,
                          uint32_t           timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the interface to use for the transfer. |
| `p_tf` | Pointer to an bp_spi_tf_t structure describing the transfer to perform. |
| `p_tf_len` | Amount of data actually transferred. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO_ERR`
`RTNC_FATAL`

Macro

# BP_SPI_DRV_HNDL_IS_NULL()

<spi/bp_spi_drv.h>

Evaluates if an SPI driver handle is NULL.

*Prototype*

```
BP_SPI_DRV_HNDL_IS_NULL  (  hndl );
```

*Parameters*

`hndl` Handle to be checked.

*Expansion*

`true` if the handle is NULL, `false` otherwise.

Macro

# BP_SPI_DRV_NULL_HNDL

<spi/bp_spi_drv.h>

NULL SPI driver handle.

Chapter

24

# UART Driver Reference

The UART driver declarations found in this module serves as the basis of UART drivers usually used in combination with the UART module to access UART peripherals. All UART drivers are composed of a standard set of API expected by the UART module in addition to any number of implementation-specific functions. The driver specific functions can be used by the application to access advanced features of a UART peripheral not exposed through the standard API. Note that usage of those extended functionalities is non-portable contrary to the standard API. The UART module API function `bp_uart_drv_hndl_get()` function can be used to retrieve the driver handle associated with a UART module instance, and can subsequently be used to call the driver directly. See the individual driver's documentation for details of the extended functions.

In addition to accessing extended functionalities, an application can access the driver standard API directly bypassing the UART module. This reduces the call overhead at the cost of thread-safety as bare driver functions are usually not thread-safe when called directly. If thread-safety is required while calling driver functions directly, it is possible to use `bp_uart_acquire()` and `bp_uart_release()` to lock the UART module preventing its access by other threads.

Finally, as yet another feature of the UART driver API, it can be invoked in a standalone fashion without a UART module instance. This reduces the RAM overhead of using a UART peripheral by dropping the UART module mutexes and internal data structures. In this case the driver create function is called directly by the application in a matter similar to `bp_uart_create()` to instantiate the driver. In this case thread safety has to be managed by the application, either using external mutexes or by ensuring that only one thread accesses the UART peripheral.

Data Type

## bp_uart_cfg_get_t

<uart/bp_uart_drv.h>

UART driver's cfg_get function.

*Prototype*

```
int  bp_uart_cfg_get_t  ( bp_uart_drv_hndl_t  hndl,
                          bp_uart_cfg_t *     p_cfg );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART driver to query. |
| `p_cfg` | Pointer to the UART configuration. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_uart_drv_cfg_set_t

<uart/bp_uart_drv.h>

UART driver's cfg_set function.

*Prototype*

```
int  bp_uart_drv_cfg_set_t  ( bp_uart_drv_hndl_t     hndl,
                              const bp_uart_cfg_t *  p_cfg,
                              uint32_t               timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART drover to configure. |
| `p_cfg` | UART configuration. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_NOT_SUPPORTED`
`RTNC_FATAL`

Data Type

## bp_uart_drv_create_t

<uart/bp_uart_drv.h>

UART driver's create function.

*Prototype*

```
int  bp_uart_drv_create_t  ( const bp_uart_board_def_t *  p_def,
                             bp_uart_drv_hndl_t *         p_hndl );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `p_def` | Board definition of the UART peripheral to initialize. |
| `p_hndl` | Pointer to the newly created UART driver instance. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_ALREADY_EXIST`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Data Type

## bp_uart_drv_destroy_t

<uart/bp_uart_drv.h>

UART driver's destroy function.

*Prototype*

```
int  bp_uart_drv_destroy_t  ( bp_uart_drv_hndl_t  hndl,
                              uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART driver instance to enable. |
| `timeout_ms` | |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_NOT_SUPPORTED`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_uart_drv_dis_t

<uart/bp_uart_drv.h>

UART driver'd disable function.

*Prototype*

```
int  bp_uart_drv_dis_t  ( bp_uart_drv_hndl_t  hndl,
                          uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART driver to disable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_uart_drv_en_t

<uart/bp_uart_drv.h>

UART driver's enable function.

*Prototype*

```
int  bp_uart_drv_en_t  ( bp_uart_drv_hndl_t  hndl,
                         uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART driver to enable. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_uart_drv_is_en_t

<uart/bp_uart_drv.h>

UART driver's is_en function.

*Prototype*

```
int  bp_uart_drv_is_en_t  ( bp_uart_drv_hndl_t  hndl,
                            bool *              p_is_en );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART driver to query. |
| `p_is_en` | Interface state, true if enabled false otherwise. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

## bp_uart_drv_reset_t

<uart/bp_uart_drv.h>

UART driver's reset function.

*Prototype*

```
int  bp_uart_drv_reset_t  ( bp_uart_drv_hndl_t  hndl,
                            uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the UART driver to reset. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_uart_drv_rx_async_abort_t

<uart/bp_uart_drv.h>

UART driver's asynchronous receive abort function.

*Prototype*

```
int  bp_uart_drv_rx_async_abort_t  ( bp_uart_drv_hndl_t  hndl,
                                     size_t *            p_rx_len,
                                     uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to abort. |
| `p_rx_len` | Pointer to the number of bytes received, can be NULL. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Data Type

## bp_uart_drv_rx_async_t

<uart/bp_uart_drv.h>

UART driver's asynchronous receive function.

*Prototype*

```
int  bp_uart_drv_rx_async_t  ( bp_uart_drv_hndl_t  hndl,
                               bp_uart_tf_t *      p_tf,
                               uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the driver to use for reception. |
| p_tf | Transfer parameters. |
| timeout_ms | Timeout value in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Data Type

## bp_uart_drv_rx_flush_t

<uart/bp_uart_drv.h>

UART driver's receive flush function.

*Prototype*

```
int  bp_uart_drv_rx_flush_t  ( bp_uart_drv_hndl_t  hndl,
                               uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the driver to flush. |
| timeout_ms | Timeout in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Data Type

## bp_uart_drv_rx_idle_wait_t

<uart/bp_uart_drv.h>

UART driver's receive idle wait function.

*Prototype*

```
int  bp_uart_drv_rx_idle_wait_t  ( bp_uart_drv_hndl_t  hndl,
                                   uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| hndl | Handle of the driver to wait. |
| timeout_ms | Timeout in milliseconds. |

*Returned Errors*
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Data Type

## bp_uart_drv_rx_t

<uart/bp_uart_drv.h>

UART driver's receive function.

*Prototype*

```
int  bp_uart_drv_rx_t  ( bp_uart_drv_hndl_t  hndl,
                         void *              p_buf,
                         size_t              len,
                         size_t *            p_rx_len,
                         uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to use for reception. |
| `p_buf` | Pointer to the buffer that will receive the data. |
| `len` | Length of the data to receive in bytes. |
| `p_rx_len` | |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO_ERR`
`RTNC_FATAL`

Data Type

## bp_uart_drv_tx_async_abort_t

<uart/bp_uart_drv.h>

UART driver's asynchronous transmit abort function.

*Prototype*

```
int  bp_uart_drv_tx_async_abort_t  ( bp_uart_drv_hndl_t  hndl,
                                     size_t *            p_tx_len,
                                     uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to abort. |
| `p_tx_len` | Pointer to the number of bytes transmitted, can be NULL. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_uart_drv_tx_async_t

<uart/bp_uart_drv.h>

UART driver's asynchronous transmit function.

*Prototype*

```
int  bp_uart_drv_tx_async_t  ( bp_uart_drv_hndl_t  hndl,
                               bp_uart_tf_t *      p_tf,
                               uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to use for reception. |
| `p_tf` | Transfer parameters. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_uart_drv_tx_flush_t

<uart/bp_uart_drv.h>

UART driver's transmit flush function.

*Prototype*

```
int  bp_uart_drv_tx_flush_t  ( bp_uart_drv_hndl_t  hndl,
                               uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to flush. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_uart_drv_tx_idle_wait_t

<uart/bp_uart_drv.h>

UART driver's transmit idle wait function.

*Prototype*

```
int  bp_uart_drv_tx_idle_wait_t  ( bp_uart_drv_hndl_t  hndl,
                                   uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to wait. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_uart_drv_tx_t

<uart/bp_uart_drv.h>

UART driver's transmit function.

*Prototype*

```
int  bp_uart_drv_tx_t  ( bp_uart_drv_hndl_t  hndl,
                         const void *        p_buf,
                         size_t              len,
                         uint32_t            timeout_ms );
```

*Attributes*

| Blocking | ISR-safe | Critical safe | Thread-safe |
|---|---|---|---|
| ✓ | ✗ | ✗ | ✓ |

*Parameters*

| | |
|---|---|
| `hndl` | Handle of the driver to use for transmission. |
| `p_buf` | Pointer to the buffer to transmit. |
| `len` | Length of the data to transmit in bytes. |
| `timeout_ms` | Timeout value in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO_ERR`
`RTNC_FATAL`

Macro

## BP_UART_DRV_HNDL_IS_NULL()

<uart/bp_uart_drv.h>

Evaluates if a UART driver handle is NULL.

*Prototype*

```
BP_UART_DRV_HNDL_IS_NULL  (  hndl );
```

*Parameters*

| | |
|---|---|
| `hndl` | Handle to be checked. |

*Expansion* `true` if the handle is NULL, `false` otherwise.

Macro

## BP_UART_DRV_NULL_HNDL

<uart/bp_uart_drv.h>

NULL UART driver handle.

Chapter

25

# Document Revision History

The revision history of the BASEplatform user manual and reference manuals can be found within the BASEplatform source package.