JBLopen

Embedded Software Insight

# BASEplatform GR712RC Reference Manual

PM0005

July 10, 2019

# Contents

Chapter

1

# Overview

Welcome to the BASEplatform™ platform reference manual for the Cobham Gaisler GR712RC dual-core fault-tolerant SoC. This reference manual covers the platform-specific API relevant to the GR712RC. This document includes important configuration information as well as the complete API reference for the GR712RC. 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 GR712RC.

For convenience during development, all the information related to each individual API elements 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.

## Missing Module or API?

The BASEplatform's set of modules is developed prioritizing our customer's need. If a needed module or API function is missing do not hesitate to inform us. In addition make sure to consult both the BASEplatform API reference manual as well as the platform-specific reference manual for a complete list of API and modules.

# 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.

# GR712RC Specific Information

## SoC Definition

The generic definitions for the GR712RC SoC peripherals can be found in `C:/ip_dev/base_platform/soc/cobham/gr712rc/bp_gr712rc_soc_def.h`. These can be used to write a custom board definition.

## External Clocks Frequency

Since there is no way for the BASEplatform to derive the input frequency of external clock inputs and external oscillators, they must be specified in the board definition file. This also holds true for the DLLBPN input pin. For example:

```
// System input clock frequency. #define BP_GR712RC_BOARD_DEF_INCLK_FREQ
(80000000U)

// State of the DLLBPN pin. #define BP_GR712RC_BOARD_DEF_DLLBPN (0U)

// External SpaceWire clock source frequency. #define
BP_GR712RC_BOARD_DEF_SPWCLK_FREQ (10000000U)

// External telecommand clock source frequency. #define
BP_GR712RC_BOARD_DEF_TMCLK_FREQ (1000000U)

// External MIL-STD-1553B clock source frequency. #define
BP_GR712RC_BOARD_DEF_1553CK_FREQ (1000000U)
```

## Important Information on GPIO and Pin Multiplexing

The GR712RC uses an I/O switch matrix to allow multiple peripherals to share the same set of pins. The multiplexing scheme is not software programmable and peripherals are assigned control according to a set of priorities for each pin. Care should be taken when using the BASEplatform peripheral reset, enable and disable functions as they may change which peripherals are driving the pins. The application is responsible to ensure that upon disabling a peripheral, the next peripheral in control won't drive the pin in an incorrect state for the external components.

## Timer scaler Configuration.

The GRLIB general purpose timer unit found in the GR712RC is made of four 32-bit timers. All four timers share the same prescaler. Care should be taken not to change the scaler value after it is first configured as this will corrupt the primary timebase as well as changing any running RTOS tick rate.

Chapter

2

# GR712RC Definitions

GR712RC global definitions. This module contains various definitions pertaining to the Cobham Gaisler GR712RC 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 GR712RC but can also be used as input values for the clock, reset and interrupt management modules.

Data Type

## bp_gr712rc_clk_gate_t

<soc/cobham/gr712rc/bp_gr712rc_def.h>

GR712RC clock gate list. These enumeration constants can be used with the clock management API such as `bp_clock_en()` and `bp_clock_dis()`.

*Values*

| | |
|---|---|
| `BP_GR712RC_CLK_GATE_ETH` | Ethernet. |
| `BP_GR712RC_CLK_GATE_SPW0` | SpaceWire 0. |
| `BP_GR712RC_CLK_GATE_SPW1` | SpaceWire 1. |
| `BP_GR712RC_CLK_GATE_SPW2` | SpaceWire 2. |
| `BP_GR712RC_CLK_GATE_SPW3` | SpaceWire 3. |
| `BP_GR712RC_CLK_GATE_SPW4` | SpaceWire 4. |
| `BP_GR712RC_CLK_GATE_SPW5` | SpaceWire 5. |
| `BP_GR712RC_CLK_GATE_CAN` | CAN. |
| `BP_GR712RC_CLK_GATE_RSVD` | Reserved. |
| `BP_GR712RC_CLK_GATE_CCSDS_ENC` | CCSDS telemetry encoder. |
| `BP_GR712RC_CLK_GATE_CCSDS_DEC` | CCSDS telecommand decoder. |

| | |
|---|---|
| `BP_GR712RC_CLK_GATE_1553B` | MIL-STD-1553B. |
| `BP_GR712RC_CLK_GATE_NONE` | No clock gate. |
| `BP_GR712RC_CLK_GATE_NULL` | Special invalid value. |

Data Type

## bp_gr712rc_clk_t

<soc/cobham/gr712rc/bp_gr712rc_def.h>

GR712RC clock list. These enumeration constants can be used with the clock management API such as `bp_clk_freq_get()`.

*Values*

| | |
|---|---|
| `BP_GR712RC_CLK_INCLK` | System clock input. |
| `BP_GR712RC_CLK_SPWCLK` | External spacewire clock. |
| `BP_GR712RC_CLK_TMCLKI` | Telemetry transponder clock input. |
| `BP_GR712RC_CLK_1553CK` | External 1553 input clock. |
| `BP_GR712RC_CLK_SYS` | System clock. |
| `BP_GR712RC_CLK_SPW` | SpaceWire TX clock. |
| `BP_GR712RC_CLK_SDCLK` | SDRAM clock. |
| `BP_GR712RC_CLK_1553` | 1553 clock. |
| `BP_GR712RC_CLK_TM` | Telemetry clock. |
| `BP_GR712RC_CLK_NULL` | Special invalid value. |

Data Type

## bp_gr712rc_int_t

<soc/cobham/gr712rc/bp_gr712rc_def.h>

GR712RC 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_GR712RC_INT_AHBSTAT` | AHB bus error. |
| `BP_GR712RC_INT_APBUART0` | APBUART 0. |
| `BP_GR712RC_INT_GPIO1` | External interrupt. |
| `BP_GR712RC_INT_GPIO2` | External interrupt. |
| `BP_GR712RC_INT_CAN0` | CAN 0. |

| | |
|---|---|
| BP_GR712RC_INT_CAN1 | CAN 1. |
| BP_GR712RC_INT_TIMER | GRTIMER. |
| BP_GR712RC_INT_GPTIMER0 | GPTIMER0. |
| BP_GR712RC_INT_GPTIMER1 | GPTIMER1. |
| BP_GR712RC_INT_GPTIMER2 | GPTIMER2. |
| BP_GR712RC_INT_GPTIMER3 | GPTIMER3. |
| BP_GR712RC_INT_IRQMP | IRQMP extended interrupt. |
| BP_GR712RC_INT_SPI | SPI, shared with SLINK. |
| BP_GR712RC_INT_1553 | 1553, shared with Ethernet and Telecommand interrupts. |
| BP_GR712RC_INT_RSVD1 | Reserved. |
| BP_GR712RC_INT_ASCS | ASCS interrupt. |
| BP_GR712RC_INT_APBUART1 | APBUART 1. |
| BP_GR712RC_INT_APBUART2 | APBUART 2. |
| BP_GR712RC_INT_APBUART3 | APBUART 3. |
| BP_GR712RC_INT_APBUART4 | APBUART 4. |
| BP_GR712RC_INT_APBUART5 | APBUART 5. |
| BP_GR712RC_INT_SPW0 | SpaceWire 0. |
| BP_GR712RC_INT_SPW1 | SpaceWire 1. |
| BP_GR712RC_INT_SPW2 | SpaceWire 2. |
| BP_GR712RC_INT_SPW3 | SpaceWire 3. |
| BP_GR712RC_INT_SPW4 | SpaceWire 4. |
| BP_GR712RC_INT_SPW5 | SpaceWire 5. |
| BP_GR712RC_INT_I2C | I2C. |
| BP_GR712RC_INT_TME | Telemetry encoder interrupt. |
| BP_GR712RC_INT_SLINK | SLINK, shared with SPI. |
| BP_GR712RC_INT_TMS | Telemetry encoder time strobe interrupt. |
| BP_GR712RC_INT_ETH | Ethernet, shared with 1553 and Telecommand interrupts. |
| BP_GR712RC_INT_TM | Telecommand, shared with 1553 and Ethernet interrupts. |
| BP_GR712RC_INT_NULL | Special invalid value. |

Data Type

# bp_gr712rc_reset_t

<soc/cobham/gr712rc/bp_gr712rc_def.h>

GR712RC 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_GR712RC_RESET_ETH` | Ethernet. |
| `BP_GR712RC_RESET_SPW0` | SpaceWire 0. |
| `BP_GR712RC_RESET_SPW1` | SpaceWire 1. |
| `BP_GR712RC_RESET_SPW2` | SpaceWire 2. |
| `BP_GR712RC_RESET_SPW3` | SpaceWire 3. |
| `BP_GR712RC_RESET_SPW4` | SpaceWire 4. |
| `BP_GR712RC_RESET_SPW5` | SpaceWire 5. |
| `BP_GR712RC_RESET_CAN` | CAN. |
| `BP_GR712RC_RESET_RSVD` | Reserved. |
| `BP_GR712RC_RESET_CCSDS_ENC` | CCSDS telemetry encoder. |
| `BP_GR712RC_RESET_CCSDS_DEC` | CCSDS telecommand decoder. |
| `BP_GR712RC_RESET_1553B` | MIL-STD-1553B. |
| `BP_GR712RC_RESET_NONE` | No reset line. |
| `BP_GR712RC_RESET_NULL` | Special invalid value. |

Macro

# BP_GR712RC_CLK_GATE_IS_VALID()

<soc/cobham/gr712rc/bp_gr712rc_def.h>

Checks if a clock gate is valid for the GR712RC.

*Expansion* `true` if the clock gate is valid. `false` otherwise.

Macro

# BP_GR712RC_CLK_IS_VALID()

<soc/cobham/gr712rc/bp_gr712rc_def.h>

Checks if a clock is valid for the GR712RC.

*Expansion* `true` if the reset line is valid. `false` otherwise.

Macro

## BP_GR712RC_CORE_CNT

<soc/cobham/gr712rc/bp_gr712rc_def.h>

Number of core in the GR712RC. Note that this is different from the BP_CFG_CORE_CNT configuration found in bp_cfg.h.

Macro

## BP_GR712RC_INT_IS_EXT()

<soc/cobham/gr712rc/bp_gr712rc_def.h>

Checks if an interrupt id is an extended interrupt on the GR712RC.

*Expansion* `true` if the interrupt is extended. `false` otherwise.

Macro

## BP_GR712RC_INT_IS_SHARED()

<soc/cobham/gr712rc/bp_gr712rc_def.h>

Checks if an interrupt id is a shared interrupt on the GR712RC.

*Expansion* `true` if the interrupt is shared. `false` otherwise.

Macro

## BP_GR712RC_INT_IS_VALID()

<soc/cobham/gr712rc/bp_gr712rc_def.h>

Checks if an interrupt id is valid for the GR712RC.

*Expansion* `true` if the interrupt id is valid. `false` otherwise.

Macro

## BP_GR712RC_RESET_IS_VALID()

<soc/cobham/gr712rc/bp_gr712rc_def.h>

Checks if a peripheral reset line is valid for the GR712RC.

*Expansion* `true` if the reset line is valid. `false` otherwise.

Macro

## BP_GR712RC_BASE_*

<soc/cobham/gr712rc/bp_gr712rc_def.h>

GR712RC base addresses.

| | |
|---|---|
| `BP_GR712RC_BASE_PROM_AREA` | PROM area base address. |
| `BP_GR712RC_BASE_IO_AREA` | External bus I/O area base address. |
| `BP_GR712RC_BASE_SRAM_AREA` | External SRAM/SDRAM area base address. |
| `BP_GR712RC_BASE_AHBRAM0_AREA` | Internal AHB on-chip RAM area base address. |
| `BP_GR712RC_BASE_AHBRAM1_AREA` | Internal AHB on-chip RAM area base address. |
| `BP_GR712RC_BASE_APB1_CTRL` | APB bridge 1 control base address. |
| `BP_GR712RC_BASE_APB2_CTRL` | APB bridge 2 control base address. |
| `BP_GR712RC_BASE_APB1_PP` | APB bridge 1 plug & play base address. |
| `BP_GR712RC_BASE_APB2_PP` | APB bridge 2 plug & play base address. |
| `BP_GR712RC_BASE_APBUART0` | APB UART 0. |
| `BP_GR712RC_BASE_APBUART1` | APB UART 1. |
| `BP_GR712RC_BASE_APBUART2` | APB UART 2. |
| `BP_GR712RC_BASE_APBUART3` | APB UART 3. |
| `BP_GR712RC_BASE_APBUART4` | APB UART 4. |
| `BP_GR712RC_BASE_APBUART5` | APB UART 5. |
| `BP_GR712RC_BASE_IRQMP` | Multiprocessor interrupt controller. |
| `BP_GR712RC_BASE_GPTIMER` | General purpose timer. |
| `BP_GR712RC_BASE_SPI` | SPI. |
| `BP_GR712RC_BASE_CANMUX` | CAN multiplexer. |
| `BP_GR712RC_BASE_GPREG` | General purpose register. |
| `BP_GR712RC_BASE_ASCS` | ASCS controller. |

| | |
|---|---|
| BP_GR712RC_BASE_SLINK | Serial bus based real-time network master. |
| BP_GR712RC_BASE_GPIO1 | GPIO1. |
| BP_GR712RC_BASE_GPIO2 | GPIO2. |
| BP_GR712RC_BASE_TM | CCSDS Telemetry Encoder. |
| BP_GR712RC_BASE_I2C | I2C. |
| BP_GR712RC_BASE_CLKGATE | Clock gate registers. |
| BP_GR712RC_BASE_ETH | Ethernet. |
| BP_GR712RC_BASE_AHBSTAT | AHB status registers. |
| BP_GR712RC_BASE_TIMER1 | Timer 1. |
| BP_GR712RC_BASE_SPW0 | SpaceWire 0. |
| BP_GR712RC_BASE_SPW1 | SpaceWire 1. |
| BP_GR712RC_BASE_SPW2 | SpaceWire 2. |
| BP_GR712RC_BASE_SPW3 | SpaceWire 3. |
| BP_GR712RC_BASE_SPW4 | SpaceWire 4. |
| BP_GR712RC_BASE_SPW5 | SpaceWire 5. |
| BP_GR712RC_BASE_DSU | Debug support unit. |
| BP_GR712RC_BASE_B1553BRM | MIL-STD_1553B. |
| BP_GR712RC_BASE_TC | Telecommand Decoder. |
| BP_GR712RC_BASE_CANOC | CAN interface. |
| BP_GR712RC_BASE_AHB_PP | AHB Plug & Play. |

Chapter

# 3 GR712RC Clock Gating Unit

Interface module to the Cobham Gaisler GR712RC clock gating unit. This module allows control of the clock gate and peripheral reset lines of the GR712RC.

This module also implements the BASEplatform clock and reset interface. See the bp_reset and bp_clock modules documentation in the BASEplatform reference manual for details.

Function

## bp_gr712rc_clock_dis()

<soc_comp/cobham/gr712rc_clock/bp_gr712rc_clock.h>

Disables a specific clock.Passing BP_GR712RC_CLK_GATE_NONE as the clk_gate will do nothing and return successfully.

*Prototype*

```
int  bp_gr712rc_clock_dis  ( bp_gr712rc_clk_gate_t  clk_gate );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| clk_gate | Clock gate of the clock to disable. |

*Returned Errors*

RTNC_SUCCESS
RTNC_FATAL

Function

## bp_gr712rc_clock_en()

<soc_comp/cobham/gr712rc_clock/bp_gr712rc_clock.h>

Enables a specific clock. Passing BP_GR712RC_CLK_GATE_NONE as the clk_gate will do nothing and return successfully.

*Prototype*

```
int  bp_gr712rc_clock_en  ( bp_gr712rc_clk_gate_t  clk_gate );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| clk_gate | Clock gate of the clock to enable. |

*Returned Errors*

RTNC_SUCCESS
RTNC_FATAL

Function

## bp_gr712rc_clock_is_en()

<soc_comp/cobham/gr712rc_clock/bp_gr712rc_clock.h>

Returns the enabled/disabled state of a clock. Passing BP_GR712RC_CLK_GATE_NONE as the clk_gate will return true through p_is_en.

*Prototype*

```
int  bp_gr712rc_clock_is_en  ( bp_gr712rc_clk_gate_t  clk_gate,
                               bool *                 p_is_en );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| clk_gate | Clock gate of the clock to query. |
| p_is_en | Pointer to the returned clock state. |

*Returned Errors*

RTNC_SUCCESS
RTNC_FATAL

Function

## bp_gr712rc_reset_assert()

<soc_comp/cobham/gr712rc_clock/bp_gr712rc_clock.h>

Asserts a specific peripheral reset. Passing BP_GR712RC_RESET_NONE as the reset will do nothing and return successfully.

*Prototype*

```
int bp_gr712rc_reset_assert ( bp_gr712rc_reset_t reset );
```

*Attributes*

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

*Parameters*

`reset` Peripheral reset to assert.

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_gr712rc_reset_deassert()

<soc_comp/cobham/gr712rc_clock/bp_gr712rc_clock.h>

Deasserts a specific peripheral reset. Passing `BP_GR712RC_RESET_NONE` as the `reset` will do nothing and return successfully.

*Prototype*

```
int bp_gr712rc_reset_deassert ( bp_gr712rc_reset_t reset );
```

*Attributes*

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

*Parameters*

`reset` Peripheral reset to deassert.

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_gr712rc_reset_is_asserted()

<soc_comp/cobham/gr712rc_clock/bp_gr712rc_clock.h>

Returns the asserted/deasserted state of a peripheral reset. Passing `BP_GR712RC_RESET_NONE` as the `reset` will return `false` through `p_is_asserted`.

*Prototype*

```
int bp_gr712rc_reset_is_asserted (       reset,
                                   bool* p_is_asserted );
```

*Attributes*

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

| | | |
|---|---|---|
| *Parameters* | `reset` | Peripheral reset to query. |
| | `p_is_asserted` | Pointer to the returned reset state. |

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

Chapter

# 4 GR712RC General Purpose Register

Interface module to the Cobham Gaisler GR712RC general purpose register.

Function

## bp_gr712rc_gpr_1553_clk_cfg_get()

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

Retrieves the 1553 clock configuration.

*Prototype*

```
int bp_gr712rc_gpr_1553_clk_cfg_get ( bp_gr712rc_gpr_1553_clk_cfg_t * p_cfg );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| p_cfg | Pointer to the returned configuration. |

*Returned Errors*

RTNC_SUCCESS
RTNC_FATAL

Function

## bp_gr712rc_gpr_1553_clk_cfg_set()

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

Sets the 1553 clock configuration.

*Prototype*

```
int bp_gr712rc_gpr_1553_clk_cfg_set ( const bp_gr712rc_gpr_1553_clk_cfg_t * p_
```

*Attributes*

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

*Parameters*

`p_cfg` Pointer to the configuration to set.

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_gr712rc_gpr_clk_freq_get()

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

Returns the frequency of a specific clock in Hertz through `p_freq`. If the clock frequency is unknown, the returned clock frequency will be 0.

*Prototype*

```
int  bp_gr712rc_gpr_clk_freq_get  ( bp_gr712rc_clk_t  clk,
                                    uint32_t *        p_freq );
```

*Attributes*

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

*Parameters*

`clk` Clock to query.
`p_freq` Pointer to the returned clock frequency in Hertz.

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_gr712rc_gpr_sdclk_del_get()

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

Returns the currently configured programmable SDCLK delay line.

*Prototype*

```
uint32_t  bp_gr712rc_gpr_sdclk_del_get  (  );
```

*Attributes*

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

*Returned Values* Current configuration of the SDCLK delay.

Function

## bp_gr712rc_gpr_sdclk_del_set()

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

Configures the programmable SDCLK delay line.

*Prototype*

```
int bp_gr712rc_gpr_sdclk_del_set ( uint32_t delay );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `delay` | Delay value to set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_gr712rc_gpr_spw_clk_cfg_get()

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

Retrieves the SpaceWire clock configuration.

*Prototype*

```
int bp_gr712rc_gpr_spw_clk_cfg_get ( bp_gr712rc_spw_clk_cfg_t * p_cfg );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `p_cfg` | Pointer to the returned configuration. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_gr712rc_gpr_spw_clk_cfg_set()

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

Sets the SpaceWire clock configuration.

*Prototype*

```
int bp_gr712rc_gpr_spw_clk_cfg_set ( bp_gr712rc_spw_clk_cfg_t * p_cfg );
```

*Attributes*

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

*Parameters*

`p_cfg` Pointer to the configuration to set.

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_gr712rc_gpr_spw_reset_assert()

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

Asserts the SpaceWire DLL reset.

*Prototype*

```
void bp_gr712rc_gpr_spw_reset_assert ( );
```

*Attributes*

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

Function

## bp_gr712rc_gpr_spw_reset_deassert()

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

Deasserts the SpaceWire DLL reset.

*Prototype*

```
void bp_gr712rc_gpr_spw_reset_deassert ( );
```

*Attributes*

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

Function

## bp_gr712rc_gpr_spw_reset_is_asserted()

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

Returns the state of the SpaceWire DLL reset.

*Prototype*

```
bool bp_gr712rc_gpr_spw_reset_is_asserted ( );
```

*Attributes*

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

*Returned Values*

`true` if the SpaceWrite DLL reset is asserted, `false` otherwise.

Function

## bp_gr712rc_gpr_tm_clk_get()

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

Gets the telecommand module clock source.

*Prototype*

```
bp_gr712rc_gpr_tm_clk_sel_t bp_gr712rc_gpr_tm_clk_get ( );
```

*Attributes*

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

*Returned Values*

Currently configured telecommand module clock source.

Function

## bp_gr712rc_gpr_tm_clk_set()

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

Sets the telecommand module clock source.

*Prototype*

```
int bp_gr712rc_gpr_tm_clk_set ( bp_gr712rc_gpr_tm_clk_sel_t clk_sel );
```

*Attributes*

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

*Parameters*

`clk_sel` Clock selection to set.

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Data Type

# bp_gr712rc_gpr_1553_clk_div_out_t

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

1553 clock divider output selection. Used within the bp_gr712rc_gpr_1553_clk_cfg_t structure to configure the 1553 clock generation.

See bp_gr712rc_gpr_1553_clk_cfg_set() and bp_gr712rc_gpr_1553_clk_cfg_get() for usage details.

*Values*

| | |
|---|---|
| BP_GR712RC_1553_CLK_DIV_OUT_SYS | Use system clock. |
| BP_GR712RC_1553_CLK_DIV_OUT_DIV | Use divided clock. |

Data Type

# bp_gr712rc_gpr_1553_clk_sel_t

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

1553 clock selection. Used within the bp_gr712rc_gpr_1553_clk_cfg_t structure to configure the 1553 clock generation.

See bp_gr712rc_gpr_1553_clk_cfg_set() and bp_gr712rc_gpr_1553_clk_cfg_get() for usage details.

*Values*

| | |
|---|---|
| BP_GR712RC_1553_CLK_SEL_GEN | Use generated clock. |
| BP_GR712RC_1553_CLK_SEL_INT | Use dedicated external clock. |

Data Type

# bp_gr712rc_gpr_tm_clk_sel_t

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

Telecommand module clock selection values.

See bp_gr712rc_gpr_tm_clk_set() and bp_gr712rc_gpr_tm_clk_get() for usage details.

*Values*

| | |
|---|---|
| BP_GR712RC_TM_CLK_SEL_CLKI | Use dedicated external clock. |
| BP_GR712RC_TM_CLK_SEL_SYS | Use system clock. |

Data Type

## bp_gr712rc_spw_clk_sel_t

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

SpaceWire module clock selection values.

See bp_gr712rc_gpr_spw_clk_cfg_set() and bp_gr712rc_gpr_spw_clk_cfg_get() for usage details.

*Values*

| | |
|---|---|
| BP_GR712RC_SPW_CLK_SEL_1X | Use 1X SpaceWire clock. |
| BP_GR712RC_SPW_CLK_SEL_SYS | Use system clock. |
| BP_GR712RC_SPW_CLK_SEL_2X | Use 2X SpaceWire clock. |
| BP_GR712RC_SPW_CLK_SEL_4X | Use 4X SpaceWire clock. |

Data Type

## bp_gr712rc_spw_clk_src_t

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

SpaceWire module clock source values.

See bp_gr712rc_gpr_spw_clk_cfg_set() and bp_gr712rc_gpr_spw_clk_cfg_get() for usage details.

*Values*

| | |
|---|---|
| BP_GR712RC_SPW_CLK_SRC_SPW | Use SpaceWire clock. |
| BP_GR712RC_SPW_CLK_SRC_IN | Use dedicated external clock. |

Data Type

## bp_gr712rc_gpr_1553_clk_cfg_t

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

1553 clock configuration structure.

See bp_gr712rc_gpr_1553_clk_cfg_set() and bp_gr712rc_gpr_1553_clk_cfg_get() for usage details.

*Members*

| | | |
|---|---|---|
| div_out | bp_gr712rc_gpr_1553_clk_div_out_t | Clock divider output selection. |
| clk_sel | bp_gr712rc_gpr_1553_clk_sel_t | Clock selection. |
| div_low | uint32_t | Clock low cycle count. |

```
div_high    uint32_t
```

Data Type

# bp_gr712rc_spw_clk_cfg_t

<soc_comp/cobham/gr712rc_gpr/bp_gr712rc_gpr.h>

SpaceWire module clock configuration structure.

See `bp_gr712rc_gpr_spw_clk_cfg_set()` and `bp_gr712rc_gpr_spw_clk_cfg_get()` for usage details.

Chapter

# 5 GR712RC Multi-Processor Interrupt Controller

Interface module to the Cobham Gaisler GR712RC IRQMP interrupt controller. This module contains platform specific API to the GR712RC. See the bp_int interrupt management module of the BASEplatform for general interrupt management API.

Function

## bp_gr712rc_int_core1_en()

<soc_comp/cobham/gr712rc_int/bp_gr712rc_int.h>

Enables core 1 of the GR712RC.

*Prototype*

```
void  bp_gr712rc_int_core1_en  (  );
```

*Attributes*

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

Chapter

# 6 LEON3 Control

Core control function for the LEON3.

Function

## bp_leon3_core_id_get()

<arch/port/sparcv8/leon3/bp_leon3_ctrl.h>

Returns the current CPU core id.

*Prototype*

```
uint32_t  bp_leon3_core_id_get  (  );
```

*Attributes*

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

*Returned Values*

CPU core id.

Function

## bp_leon3_pcr_get()

<arch/port/sparcv8/leon3/bp_leon3_ctrl.h>

Returns the value of the LEON3 processor configuration register.

*Prototype*

```
uint32_t  bp_leon3_pcr_get  (  );
```

*Attributes*

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

*Returned Values* Processor configuration register value read.

Function

## bp_leon3_pcr_set()

<arch/port/sparcv8/leon3/bp_leon3_ctrl.h>

Sets the value of the LEON3 processor configuration register.

*Prototype*

```
void bp_leon3_pcr_set ( uint32_t pcr );
```

*Attributes*

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

*Parameters* `pcr` PCR register value to set.

Function

## bp_leon3_svt_dis()

<arch/port/sparcv8/leon3/bp_leon3_ctrl.h>

Disables single vector trapping

*Prototype*

```
void bp_leon3_svt_dis ( );
```

*Attributes*

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

Function

## bp_leon3_svt_en()

<arch/port/sparcv8/leon3/bp_leon3_ctrl.h>

Enables single vector trapping.

*Prototype*

```
void bp_leon3_svt_en ( );
```

*Attributes*

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

Function

## bp_leon3_svt_is_en()

<arch/port/sparcv8/leon3/bp_leon3_ctrl.h>

Returns the enabled/disabled state of single vector trapping.

*Prototype*

```
bool bp_leon3_svt_is_en ( );
```

*Attributes*

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

Function

## bp_leon3_wt_dis()

<arch/port/sparcv8/leon3/bp_leon3_ctrl.h>

Disables write error trap.

*Prototype*

```
void bp_leon3_wt_dis ( );
```

*Attributes*

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

Function

## bp_leon3_wt_en()

<arch/port/sparcv8/leon3/bp_leon3_ctrl.h>

Enables write error trap.

*Prototype*

```
void bp_leon3_wt_en ( );
```

*Attributes*

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

Function

# bp_leon3_wt_is_en()

<arch/port/sparcv8/leon3/bp_leon3_ctrl.h>

Returns the enabled/disabled state of the write error trap.

*Prototype*

```
bool bp_leon3_wt_is_en ( );
```

*Attributes*

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

Chapter

# 7 LEON3 Cache Control

Control and status functions for the LEON3 cache system.

Function

## bp_leon3_ccr_get()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Returns the current value of the cache control register (CCR).

*Prototype*

```
uint32_t  bp_leon3_ccr_get  (  );
```

*Attributes*

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

Function

## bp_leon3_ccr_set()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Sets the cache control register (CCR).

*Prototype*

```
void  bp_leon3_ccr_set  (  );
```

*Attributes*

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

Function

## bp_leon3_dcache_dis()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Disables the data cache.

*Prototype* `void bp_leon3_dcache_dis ( );`

*Attributes*

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

Function

## bp_leon3_dcache_en()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Enables the data cache.

*Prototype* `void bp_leon3_dcache_en ( );`

*Attributes*

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

Function

## bp_leon3_dcache_flush()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Flushes (invalidates) the entire data cache.

*Prototype* `void bp_leon3_dcache_flush ( );`

*Attributes*

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

Function

## bp_leon3_dcache_is_en()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Returns the enabled/disabled status of the data cache.

*Prototype* `bool bp_leon3_dcache_is_en ( );`

*Attributes*

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

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

Function

## bp_leon3_ds_dis()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Disables data cache snooping.

*Prototype*

```
void bp_leon3_ds_dis ( );
```

*Attributes*

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

Function

## bp_leon3_ds_en()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Enables data cache snooping.

*Prototype*

```
void bp_leon3_ds_en ( );
```

*Attributes*

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

Function

## bp_leon3_ds_is_en()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Returns the enabled/disabled status of data cache snooping

*Prototype*

```
bool bp_leon3_ds_is_en ( );
```

*Attributes*

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

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

Function

## bp_leon3_ib_dis()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Disables instruction cache bursts.

*Prototype* `void bp_leon3_ib_dis ( );`

*Attributes*

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

Function

## bp_leon3_ib_en()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Enables instruction cache bursts.

*Prototype* `void bp_leon3_ib_en ( );`

*Attributes*

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

Function

## bp_leon3_ib_is_en()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Returns the enabled/disabled status of instruction cache bursts.

*Prototype* `bool bp_leon3_ib_is_en ( );`

*Attributes*

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

*Returned Values* `true` if the instruction cache bursts are enabled `false` otherwise.

Function

## bp_leon3_icache_dis()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Disables the instruction cache.

*Prototype* `void bp_leon3_icache_dis ( );`

*Attributes*

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

Function

## bp_leon3_icache_en()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Enables the instruction cache.

*Prototype* `void bp_leon3_icache_en ( );`

*Attributes*

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

Function

## bp_leon3_icache_flush()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Flushes (invalidates) the entire instruction cache.

*Prototype* `void bp_leon3_icache_flush ( );`

*Attributes*

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

Function

## bp_leon3_icache_is_en()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Returns the enabled/disabled status of the instruction cache.

*Prototype* `bool bp_leon3_icache_is_en ( );`

*Attributes*

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

*Returned Values*

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

Function

## bp_leon3_tlb_flush()

<arch/port/sparcv8/leon3/bp_leon3_cache_ctrl.h>

Flushes (invalidates) the TLB.

*Prototype*

```
void  bp_leon3_tlb_flush  (  );
```

*Attributes*

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

Chapter

# 8 LEON3 ASI Mapping

LEON3 Address Space Identifier (ASI) mapping. Note that this mapping differs from the generic ASI mapping documented in the SPARCv8 reference manual.

Macro

## BP_LEON3_ASI_*

<arch/port/sparcv8/leon3/bp_leon3_asi.h>

LEON3 ASI mapping definitions.

| | |
|---|---|
| BP_LEON3_ASI_CACHE_BYPASS | Forced cache miss. |
| BP_LEON3_ASI_SCR | System control registers. |
| BP_LEON3_ASI_NORMAL | Normal access. |
| BP_LEON3_ASI_INS_CTAG | Instruction cache tags. |
| BP_LEON3_ASI_INS_CDATA | Instruction cache data. |
| BP_LEON3_ASI_DATA_CTAG | Data cache tags. |
| BP_LEON3_ASI_DATA_CDATA | Data cache data. |
| BP_LEON3_ASI_CACHE_FLUSH | Instruction and data cache flush. |
| BP_LEON3_ASI_DCACHE_FLUSH | Data cache flush. |
| BP_LEON3_ASI_MMU_DIAG_DATA | MMU diagnostic data cache context. |
| BP_LEON3_ASI_MMU_DIAG_INS | MMU diagnostic instruction cache context. |
| BP_LEON3_ASI_CACHE_TLB_FLUSH | Flush MMU, TLB, instruction and data caches. |
| BP_LEON3_ASI_MMU_REG | MMU diagnostic instruction cache context. |
| BP_LEON3_ASI_MMU_BYPASS | MMU bypass. |

| | |
|---|---|
| `BP_LEON3_ASI_MMU_DIAG` | MMU diagnostic access. |
| `BP_LEON3_ASI_MMU_SNOOP` | MMU snoop tag diagnostic access. |

Chapter

# 9 SPARCv8 Control

Generic core control function for the SPARCv8.

Function

## bp_sparcv8_asi_load()

<arch/port/sparcv8/bp_sparcv8_ctrl.h>

Loads a 4 bytes word from an alternate address space identifier. This function only supports ASI 0 to 31. ASI ID outside of the supported range will be ignored and the return value in those cases is undefined.

Specifying an unaligned address will likely cause a processor trap.

*Prototype*

```
uint32_t  bp_sparcv8_asi_load  ( uint32_t  address,
                                 uint32_t  asi );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `address` | Address to load, should be 4 bytes aligned. |
| `asi` | Address space identifier (ASI) |

*Returned Values*

Value read.

Function

# bp_sparcv8_asi_store()

<arch/port/sparcv8/bp_sparcv8_ctrl.h>

Stores a 4 bytes word to an alternate address space identifier. This function only supports ASI 0 to 31. ASI ID outside of the supported range will be ignored and the function will return without performing any store operation.

Specifying an unaligned address will likely cause a processor trap.

*Prototype*

```
void  bp_sparcv8_asi_store  ( uint32_t  address,
                              uint32_t  asi,
                              uint32_t  val );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `address` | Address to load, should be 4 bytes aligned. |
| `asi` | Address space identifier (ASI) |
| `val` | Value to store. |

Chapter

10

# SPARCv8 Reference MMU

Interface module to the SPARC Reference MMU for the SPARCv8 architecture.

Function

## bp_sparcv8_mmu_init()

<soc_comp/sparc/sparcv8_mmu/bp_sparcv8_mmu.h>

Initializes the SPARCv8 MMU.

*Prototype*

```
int  bp_sparcv8_mmu_init  (  );
```

*Attributes*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_sparcv8_mmu_map()

<soc_comp/sparc/sparcv8_mmu/bp_sparcv8_mmu.h>

Maps a memory region using the specified permission and cacheability attribute.

Note that the region start address and size must be a multiple of 16 MiB.

*Prototype*

```
int  bp_sparcv8_mmu_map  ( uint32_t  permissions,
                           bool      cacheable,
                           void *    p_start_addr,
                           uint32_t  size );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `permissions` | Region permission. |
| `cacheable` | `true` if memory region should be cacheable, `false` otherwise. |
| `p_start_addr` | Start address of the MMU region to map, should be aligned to 16 MiB. |
| `size` | Size of the memory region, should be aligned to 16 MiB. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Macro

# BP_SPARCV8_MMU_ACC_*

<soc_comp/sparc/sparcv8_mmu/bp_sparcv8_mmu.h>

Description SPARCv8 MMU access permission definitions.

| | |
|---|---|
| `BP_SPARCV8_MMU_ACC_UR_SR` | User read only and supervisor read only. |
| `BP_SPARCV8_MMU_ACC_URW_SRW` | User read/write and supervisor read/write. |
| `BP_SPARCV8_MMU_ACC_URE_SRE` | User read/execute and supervisor read/execute. |
| `BP_SPARCV8_MMU_ACC_URWE_SRWE` | User read/write/execute and supervisor read/write/execute. |
| `BP_SPARCV8_MMU_ACC_UE_SE` | User execute only and supervisor execute only. |
| `BP_SPARCV8_MMU_ACC_UR_SRW` | User read only and supervisor read/write. |
| `BP_SPARCV8_MMU_ACC_SRW` | User no access and supervisor read/execute. |
| `BP_SPARCV8_MMU_ACC_SRWE` | User no access and supervisor read/write/execute. |

Chapter

# 11 GRLIB General Purpose Timer

Interface module to the Cobham Gaisler GRLIB General Purpose Timer (GPTIMER).

Function

## bp_grlib_gptimer_create()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

Creates a GPTIMER 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_grlib_gptimer_create()` returns an `RTNC_ALREADY_EXIST` error without affecting the already opened interface.

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

When `bp_grlib_gptimer_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_grlib_gptimer_create  ( bp_grlib_gptimer_board_def_t *  p_def,
                                bp_grlib_gptimer_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_grlib_gptimer_freeze_dis()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

Disables the freezing of timer instance `timer_hndl` when the CPU enters debug state. This applies to all the timers of an instance.

*Prototype*

```
int  bp_grlib_gptimer_freeze_dis  ( bp_grlib_gptimer_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_grlib_gptimer_freeze_en()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

Enables the freezing of timer instance `timer_hndl` when the CPU enters debug state. This applies to all the timers of an instance.

*Prototype*

```
int  bp_grlib_gptimer_freeze_en  ( bp_grlib_gptimer_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_grlib_gptimer_freeze_is_en()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

Queries the enabled/disabled state of timer freezing under debug state of timer instance `timer_hndl`.

*Prototype*
```
int  bp_grlib_gptimer_freeze_is_en  ( bp_grlib_gptimer_hndl_t  timer_hndl,
                                      bool *                   p_is_en );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `p_is_en` | Pointer to the returned state, `true` if freezing is enabled, `false` otherwise. |

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Function

## bp_grlib_gptimer_int_dis()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

Disables the interrupt signal of timer index `timer_ix` of timer instance `timer_hndl`.

*Prototype*
```
int  bp_grlib_gptimer_int_dis  ( bp_grlib_gptimer_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 set. |

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Function

# bp_grlib_gptimer_int_en()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

Enables the interrupt signal of timer index `timer_ix` of timer instance `timer_hndl`.

*Prototype*

```
int  bp_grlib_gptimer_int_en  ( bp_grlib_gptimer_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 set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_grlib_gptimer_int_is_en()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

Queries the enabled/disabled state of interrupts for timer index `timer_ix` of timer instance `timer_hndl`.

*Prototype*

```
int  bp_grlib_gptimer_int_is_en  ( bp_grlib_gptimer_hndl_t  timer_hndl,
                                   uint32_t                 timer_ix,
                                   bool *                   p_is_en );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `timer_ix` | Timer index to query. |
| `p_is_en` | Pointer to the returned state, `true` if interrupts are enabled, `false` otherwise. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_grlib_gptimer_read()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

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

*Prototype*

```
int  bp_grlib_gptimer_read  ( bp_grlib_gptimer_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_grlib_gptimer_read64()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

Returns the current combined timer counter value of timers `timer_ix` and `timer_ix`

- 1 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_grlib_gptimer_read64  ( bp_grlib_gptimer_hndl_t  timer_hndl,
                                uint32_t                 timer_ix,
                                uint64_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_grlib_gptimer_start()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

Starts timer index `timer_ix` of a timer module instance. The timer reload value will be set to `reload_val` and started. If the timer is already started, it will be stopped and restarted with the new settings. If `restart` is true the timer will be set to automatically reload the reload value upon expiration.

*Prototype*

```
int  bp_grlib_gptimer_start  ( bp_grlib_gptimer_hndl_t  timer_hndl,
                               uint32_t                 timer_ix,
                               uint32_t                 reload_val,
                               bool                     restart );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `timer_ix` | Timer index to start. |
| `reload_val` | Reload value to set. |
| `restart` | Set to `true` to enable auto-reloading the timer upon expiration, or `false` for a one-shot timer. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_grlib_gptimer_start64()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

Chains and starts timer index `timer_ix` and `timer_ix` + 1 . The timers reload values will be set to `reload_val` and started. If one or both timers are already started, they will be stopped and restarted with the new settings. If `restart` is true the timers will be set to automatically reload the reload value upon expiration.

*Prototype*

```
int  bp_grlib_gptimer_start64  ( bp_grlib_gptimer_hndl_t  timer_hndl,
                                 uint32_t                 timer_ix,
                                 uint64_t                 reload_val,
                                 bool                     restart );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `timer_hndl` | Handle of the timer instance. |
| `timer_ix` | Timer index to start. |
| `reload_val` | Reload value to set. |
| `restart` | Set to `true` to enable auto-reloading the timer upon expiration, or `false` for a one-shot timer. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_grlib_gptimer_stop()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.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_grlib_gptimer_stop  ( bp_grlib_gptimer_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_grlib_gptimer_stop64()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

Stops chained timer index `timer_ix` and `timer_ix` +1 of a timer module instance. If the timer wasn't started, nothing is done and `RTNC_SUCCESS` is returned. If the timers weren't chained, they will be stopped nonetheless.

*Prototype*

```
int  bp_grlib_gptimer_stop64  ( bp_grlib_gptimer_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`

Data Type

## bp_grlib_gptimer_board_def_t

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

GRLIB GPTIMER module board level hardware definition structure. Used to set the name of a GPTIMER instance.

See bp_grlib_gptimer_create() for details.

*Members*

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

Data Type

## bp_grlib_gptimer_hndl_t

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

GRLIB GPTIMER handle. Returned by `bp_grlib_gptimer_create()`. The pointer contained in the handle is private and should not be accessed by calling code.

Data Type

## bp_grlib_gptimer_inst_t

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

Data Type

## bp_grlib_gptimer_soc_def_t

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

GRLIB GPTIMER 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_grlib_gptimer_t` board definition structure to describe a complete GPTIMER instance.

See bp_grlib_gptimer_create() for details.

*Members*

| | | |
|---|---|---|
| `base_addr` | `void *` | Peripheral base address. |
| `base_int_id` | `uint32_t` | Peripheral base interrupt id. |
| `timer_cnt` | `uint32_t` | Number of implemented timers. |
| `separate_int` | `bool` | Set to `true` if each timer use a separate interrupt line. |
| `time_latch` | `bool` | Set to `true` if time latching is supported. |

Macro

## BP_GRLIB_GPTIMER_NULL_HNDL

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

NULL GRLIB GPTIMER handle.

Macro

## BP_UART_HNDL_IS_NULL()

<soc_comp/cobham/grlib/grlib_gptimer/bp_grlib_gptimer.h>

Evaluates if a GRLIB GPTIMER 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.

Chapter

12

# GRLIB APBUART Driver

UART Driver for the Cobham Gaisler GRLIB APBUART peripheral 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_grlib_apbuart_cfg_get()`

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Retrieves the current configuration of a UART interface.

See `bp_uart_drv_cfg_get_t` for usage details.

*Prototype*

```
int  bp_grlib_apbuart_cfg_get  ( bp_uart_drv_hndl_t  drv_hndl,
                                 bp_uart_cfg_t *     p_cfg,
                                                     timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_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_grlib_apbuart_cfg_set()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Configures a UART peripheral.

See bp_uart_drv_cfg_set_t for usage details.

*Prototype*

```
int  bp_grlib_apbuart_cfg_set  ( bp_uart_drv_hndl_t       drv_hndl,
                                 const bp_uart_cfg_t *    p_cfg,
                                 uint32_t                 timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| drv_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_grlib_apbuart_create()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Creates a UART driver instance.

See bp_uart_drv_create_t for usage details.

*Prototype*

```
int  bp_grlib_apbuart_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_grlib_apbuart_destroy()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Destroys a UART driver instance.

See `bp_uart_drv_destroy_t` for usage details.

*Prototype*

```
int  bp_grlib_apbuart_destroy  ( bp_uart_drv_hndl_t  drv_hndl,
                                 uint32_t            timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_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_grlib_apbuart_dis()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Disables a UART peripheral.

See `bp_uart_drv_dis_t` for usage details.

*Prototype*

```
int  bp_grlib_apbuart_dis  ( bp_uart_drv_hndl_t  drv_hndl,
                             uint32_t            timeout_ms );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_grlib_apbuart_en()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Enables a UART peripheral.

See `bp_uart_drv_en_t` for usage details.

*Prototype*

```
int  bp_grlib_apbuart_en  ( bp_uart_drv_hndl_t  drv_hndl,
                            uint32_t            timeout_ms );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_grlib_apbuart_is_en()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Returns the enabled/disabled state of a UART peripheral.

See `bp_uart_drv_is_en_t` for usage details.

*Prototype*

```
int  bp_grlib_apbuart_is_en  ( bp_uart_drv_hndl_t  drv_hndl,
                               bool *              p_is_en );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_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_grlib_apbuart_lb_dis()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Disables the GRLIB APBUART UART loopback mode.

Note that while this function is thread-safe it doesn't wait for the interface to be idle and will disable the loopback mode immediately.

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

*Prototype*

```
int  bp_grlib_apbuart_lb_dis  ( bp_uart_drv_hndl_t  drv_hndl );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_hndl` | Handle of the UART driver interface to set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_grlib_apbuart_lb_en()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Enables the GRLIB APBUART UART loopback mode.

Note that while this function is thread-safe it doesn't wait for the interface to be idle and will enable the loopback mode immediately.

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

*Prototype*

```
int  bp_grlib_apbuart_lb_en  ( bp_uart_drv_hndl_t  drv_hndl );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_hndl` | Handle of the UART driver interface to set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_grlib_apbuart_lb_is_en()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Returns the enabled/disabled state of the GRLIB APBUART loopback mode feature.

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

*Prototype*

```
int  bp_grlib_apbuart_lb_is_en  ( bp_uart_drv_hndl_t  drv_hndl,
                                  bool *              p_is_en );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_hndl` | Handle of the UART driver interface to query. |
| `p_is_en` | Pointer to the returned state. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_grlib_apbuart_reset()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Resets a UART peripheral.

See `bp_uart_drv_reset_t` for usage details.

*Prototype*

```
int  bp_grlib_apbuart_reset  ( bp_uart_drv_hndl_t  drv_hndl,
                               uint32_t            timeout_ms );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_grlib_apbuart_rx()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Receives data.

See `bp_uart_drv_rx_t` for usage details.

*Prototype*

```
int  bp_grlib_apbuart_rx  ( bp_uart_drv_hndl_t  drv_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*

| | |
|---|---|
| `drv_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_grlib_apbuart_rx_async()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Receive data asynchronously.

See `bp_uart_drv_rx_async_t` for usage details.

*Prototype*

```
int  bp_grlib_apbuart_rx_async  ( bp_uart_drv_hndl_t  drv_hndl,
                                  bp_uart_tf_t *      p_tf,
                                  uint32_t            timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_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_grlib_apbuart_rx_async_abort()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Aborts an asynchronous transfer.

See `bp_uart_drv_rx_async_abort_t` for usage details.

*Prototype*

```
int  bp_grlib_apbuart_rx_async_abort  ( bp_uart_drv_hndl_t  drv_hndl,
                                        size_t *            p_rx_len,
                                        uint32_t            timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_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_grlib_apbuart_rx_flush()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Flush the transmit path.

See bp_uart_drv_rx_flush_t for usage details.

*Prototype*

```
int  bp_grlib_apbuart_rx_flush  ( bp_uart_drv_hndl_t  drv_hndl,
                                  uint32_t            timeout_ms );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_grlib_apbuart_rx_idle_wait()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Waits for a UART interface to be idle.

See bp_uart_drv_rx_idle_wait_t for usage details.

*Prototype*

```
int  bp_grlib_apbuart_rx_idle_wait  ( bp_uart_drv_hndl_t  drv_hndl,
                                      uint32_t            timeout_ms );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

# bp_grlib_apbuart_tx()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Transmits data.

See bp_uart_drv_tx_t for usage details.

*Prototype*

```
int  bp_grlib_apbuart_tx  ( bp_uart_drv_hndl_t  drv_hndl,
                            const void *        p_buf,
                            size_t              len,
                            uint32_t            timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| drv_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_grlib_apbuart_tx_async()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Transmits data asynchronously.

See bp_uart_drv_tx_async_t for usage details.

*Prototype*

```
int  bp_grlib_apbuart_tx_async  ( bp_uart_drv_hndl_t  drv_hndl,
                                  bp_uart_tf_t *      p_tf,
                                  uint32_t            timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| drv_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_grlib_apbuart_tx_async_abort()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Aborts an asynchronous transfer.

See bp_uart_drv_tx_async_abort_t for usage details.

*Prototype*
```
int  bp_grlib_apbuart_tx_async_abort  ( bp_uart_drv_hndl_t  drv_hndl,
                                        size_t *            p_tx_len,
                                        uint32_t            timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| drv_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_grlib_apbuart_tx_flush()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Flush the receive path.

See bp_uart_drv_tx_flush_t for usage details.

*Prototype*
```
int  bp_grlib_apbuart_tx_flush  ( bp_uart_drv_hndl_t  drv_hndl,
                                  uint32_t            timeout_ms );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_grlib_apbuart_tx_idle_wait()

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

Waits for a UART interface to be idle.

See `bp_uart_drv_tx_idle_wait_t` for usage details.

*Prototype*

```
int  bp_grlib_apbuart_tx_idle_wait  ( bp_uart_drv_hndl_t  drv_hndl,
                                      uint32_t            timeout_ms );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

## bp_grlib_apbuart_drv_def_t

<soc_comp/cobham/grlib/grlib_apbuart/bp_grlib_apbuart_drv.h>

GRLIB APBUART 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. |
| `clk_gate_id` | `int` | Peripheral clock gate id. |

| | | |
|---|---|---|
| `reset_id` | `int` | Peripheral reset id. |

Chapter

13

# GRLIB I2C Driver

I2C driver for the Cobham Gaisler GRLIB I2C peripheral 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.

Note that the GRLIB I2C IP support I2C master operation only. Attempting to configure the driver in slave mode will return an `RTNC_NOT_SUPPORTED` error.

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

Function

## bp_grlib_i2c_cfg_get()

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_i2c_drv.h>

Retrieves the current configuration of an I2C interface.

See `bp_i2c_drv_cfg_get_t` for usage details.

*Prototype*

```
int  bp_grlib_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_grlib_i2c_cfg_set()

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_i2c_drv.h>

Configures an I2C interface.

See bp_i2c_drv_cfg_set_t for usage details.

*Prototype*
```
int  bp_grlib_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_grlib_i2c_create()

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_i2c_drv.h>

Creates an I2C driver instance.

See bp_i2c_drv_create_t for usage details.

*Prototype*
```
int  bp_grlib_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_grlib_i2c_destroy()

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_i2c_drv.h>

Destroys an I2C interface.

See `bp_i2c_drv_destroy_t` for usage details.

*Prototype*

```
int  bp_grlib_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_grlib_i2c_dis()

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_i2c_drv.h>

Disables an I2C interface.

See `bp_i2c_drv_dis_t` for usage details.

*Prototype*

```
int  bp_grlib_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_grlib_i2c_en()

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_i2c_drv.h>

Enables an I2C interface.

See `bp_i2c_drv_en_t` for usage details.

*Prototype*

```
int  bp_grlib_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_grlib_i2c_flush()

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_i2c_drv.h>

Flush the transmit and receive paths.

See `bp_i2c_drv_flush_t` for usage details.

*Prototype*

```
int  bp_grlib_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_grlib_i2c_idle_wait()

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_i2c_drv.h>

Waits for an I2C interface to be idle.

See `bp_i2c_drv_idle_wait_t` for usage details.

*Prototype*

```
int  bp_grlib_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_grlib_i2c_is_en()

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_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_grlib_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_grlib_i2c_reset()

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_i2c_drv.h>

Resets an I2C interface.

See `bp_i2c_drv_reset_t` for usage details.

*Prototype*

```
int  bp_grlib_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_grlib_i2c_xfer()

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_i2c_drv.h>

Performs an I2C operation.

See `bp_i2c_drv_xfer_t` for usage details.

*Prototype*

```
int  bp_grlib_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 a `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_grlib_i2c_xfer_async()

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_i2c_drv.h>

Transfers data asynchronously.

See `bp_i2c_drv_xfer_async_t` for usage details.

*Prototype*

```
int  bp_grlib_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_grlib_i2c_xfer_async_abort()

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_i2c_drv.h>

Aborts an asynchronous transfer.

See `bp_i2c_drv_xfer_async_abort_t` for usage details.

*Prototype*

```
int  bp_grlib_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_grlib_i2c_drv_def_t

<soc_comp/cobham/grlib/grlib_i2c/bp_grlib_i2c_drv.h>

GRLIB 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 GRLIB SPI Driver

SPI driver for the Cobham Gaisler GRLIB SPI 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_grlib_spi_cfg_get()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_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_grlib_spi_cfg_get  ( bp_spi_drv_hndl_t  drv_hndl,
                             bp_spi_cfg_t *     p_cfg,
                             uint32_t           timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_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_grlib_spi_cfg_set()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Configures an SPI peripheral.

See bp_spi_drv_cfg_set_t for usage details.

*Prototype*

```
int  bp_grlib_spi_cfg_set  ( bp_spi_drv_hndl_t    drv_hndl,
                             const bp_spi_cfg_t *  p_cfg,
                             uint32_t              timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| drv_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_grlib_spi_create()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Creates an SPI driver instance.

See bp_spi_drv_create_t for usage details.

*Prototype*

```
int  bp_grlib_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_grlib_spi_destroy()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Destroys an SPI driver instance.

See `bp_spi_drv_destroy_t` for usage details.

*Prototype*

```
int  bp_grlib_spi_destroy  ( bp_spi_drv_hndl_t  drv_hndl,
                             uint32_t           timeout_ms );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_grlib_spi_dis()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Disables an SPI peripheral.

See `bp_spi_drv_dis_t` for usage details.

*Prototype*

```
int  bp_grlib_spi_dis  ( bp_spi_drv_hndl_t  drv_hndl,
                         uint32_t           timeout_ms );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_grlib_spi_en()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Enables an SPI peripheral

See `bp_spi_drv_en_t` for usage details.

*Prototype*

```
int  bp_grlib_spi_en  ( bp_spi_drv_hndl_t  drv_hndl,
                        uint32_t           timeout_ms );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

# bp_grlib_spi_flush()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Flush the transmit and receive paths.

See `bp_spi_drv_flush_t` for usage details.

*Prototype*

```
int  bp_grlib_spi_flush  ( bp_spi_drv_hndl_t  drv_hndl,
                           uint32_t           timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_hndl` | Handle of the peripheral to flush. |
| `timeout_ms` | Timeout in milliseconds. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_grlib_spi_idle_wait()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Wait for an SPI peripheral to be idle.

See `bp_spi_drv_idle_wait_t` for usage details.

*Prototype*

```
int  bp_grlib_spi_idle_wait  ( bp_spi_drv_hndl_t  drv_hndl,
                               uint32_t           timeout_ms );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_grlib_spi_is_en()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_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_grlib_spi_is_en  ( bp_spi_drv_hndl_t  drv_hndl,
                           bool *             p_is_en );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_hndl` | Handle of the SPI driver to check. |
| `p_is_en` | Interface state, `true` if enabled `false` otherwise. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_grlib_spi_lb_dis()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Disables the GRLIB SPI loopback mode.

Note that while this function is thread-safe it doesn't wait for the interface to be idle and will disable the loopback mode immediately.

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

*Prototype*

```
int  bp_grlib_spi_lb_dis  ( bp_spi_drv_hndl_t  drv_hndl );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_hndl` | Handle of the SPI driver interface to set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

# bp_grlib_spi_lb_en()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Enables the GRLIB SPI loopback mode.

Note that while this function is thread-safe it doesn't wait for the interface to be idle and will enable the loopback mode immediately.

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

*Prototype*

```
int  bp_grlib_spi_lb_en  ( bp_spi_drv_hndl_t  drv_hndl );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_hndl` | Handle of the SPI driver interface to set. |

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_grlib_spi_lb_is_en()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Returns the enabled/disabled state of the GRLIB SPI loopback mode.

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

*Prototype*

```
int  bp_grlib_spi_lb_is_en  ( bp_spi_drv_hndl_t  drv_hndl,
                              bool *             p_is_en );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_grlib_spi_reset()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_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_grlib_spi_reset  ( bp_spi_drv_hndl_t  drv_hndl,
                           uint32_t           timeout_ms );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_grlib_spi_slave_desel()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Deselect a selected SPI slave.

See `bp_spi_drv_slave_desel_t` for usage details.

*Prototype*

```
int  bp_grlib_spi_slave_desel  ( bp_spi_drv_hndl_t  drv_hndl,
                                 uint32_t           timeout_ms );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

## bp_grlib_spi_slave_sel()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Select a specific SPI slave.

See `bp_spi_drv_slave_sel_t` for usage details.

*Prototype*

```
int  bp_grlib_spi_slave_sel  ( bp_spi_drv_hndl_t  drv_hndl,
                               uint32_t           ss_id,
                               uint32_t           timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_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_grlib_spi_xfer()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Performs an SPI operation.

See `bp_spi_drv_xfer_t` for usage details.

*Prototype*

```
int  bp_grlib_spi_xfer  ( bp_spi_drv_hndl_t  drv_hndl,
                          bp_spi_tf_t *      p_tf,
                                             p_recv_len,
                          uint32_t           timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_hndl` | Handle of the driver to use. |
| `p_tf` | Pointer to an `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_grlib_spi_xfer_async()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Transfer data asynchronously.

See bp_spi_drv_xfer_async_t for usage details.

*Prototype*

```
int  bp_grlib_spi_xfer_async  ( bp_spi_drv_hndl_t  drv_hndl,
                                bp_spi_tf_t *      p_tf,
                                uint32_t           timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| drv_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_grlib_spi_xfer_async_abort()

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

Aborts an asynchronous transfer.

See bp_spi_drv_xfer_async_abort_t for usage details.

*Prototype*

```
int  bp_grlib_spi_xfer_async_abort  ( bp_spi_drv_hndl_t  drv_hndl,
                                      size_t *           p_tx_len,
                                      size_t *           p_rx_len,
                                      uint32_t           timeout_ms );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| drv_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_grlib_spi_drv_def_t

<soc_comp/cobham/grlib/grlib_spi/bp_grlib_spi_drv.h>

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. |
| `clk_gate_id` | `int` | Peripheral clock gate id. |
| `reset_id` | `int` | Peripheral reset id. |

Chapter

# 15 GRLIB GPIO Driver

GPIO driver for the Cobham Gaisler GRLIB APBUART peripheral 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_grlib_gpio_create()

<soc_comp/cobham/grlib/grlib_gpio/bp_grlib_gpio_drv.h>

Creates a GPIO driver instance.

See `bp_gpio_drv_create_t` for usage details.

*Prototype*

```
int  bp_grlib_gpio_create  ( const bp_gpio_board_def_t *  p_def,
                             bp_gpio_drv_hndl_t *        p_drv_hndl );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

# bp_grlib_gpio_data_get()

<soc_comp/cobham/grlib/grlib_gpio/bp_grlib_gpio_drv.h>

Gets the state of a GPIO pin.

See bp_gpio_drv_data_get_t for usage details.

*Prototype*

```
int  bp_grlib_gpio_data_get  ( bp_gpio_drv_hndl_t  drv_hndl,
                               uint32_t            bank,
                               uint32_t            pin,
                               uint32_t *          p_data );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| drv_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_grlib_gpio_data_set()

<soc_comp/cobham/grlib/grlib_gpio/bp_grlib_gpio_drv.h>

Sets the state of a GPIO pin.

See bp_gpio_drv_data_set_t for usage details.

*Prototype*

```
int  bp_grlib_gpio_data_set  ( bp_gpio_drv_hndl_t  drv_hndl,
                               uint32_t            bank,
                               uint32_t            pin,
                               uint32_t            data );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| drv_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_grlib_gpio_data_tog()

<soc_comp/cobham/grlib/grlib_gpio/bp_grlib_gpio_drv.h>

Toggle the state of a GPIO pin.

See `bp_gpio_drv_data_tog_t` for usage details.

*Prototype*

```
int  bp_grlib_gpio_data_tog  ( bp_gpio_drv_hndl_t  drv_hndl,
                               uint32_t            bank,
                               uint32_t            pin );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_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_grlib_gpio_destroy()

<soc_comp/cobham/grlib/grlib_gpio/bp_grlib_gpio_drv.h>

Destroys a GPIO driver instance.

See `bp_gpio_drv_destroy_t` for usage details.

*Prototype*

```
int  bp_grlib_gpio_destroy  ( bp_gpio_drv_hndl_t  drv_hndl );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Function

# bp_grlib_gpio_dir_get()

<soc_comp/cobham/grlib/grlib_gpio/bp_grlib_gpio_drv.h>

Gets the direction of a GPIO pin.

See bp_gpio_drv_dir_get_t for usage details.

*Prototype*

```
int  bp_grlib_gpio_dir_get  ( bp_gpio_drv_hndl_t  drv_hndl,
                              uint32_t            bank,
                              uint32_t            pin,
                              bp_gpio_dir_t *     p_dir );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| drv_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_grlib_gpio_dir_set()

<soc_comp/cobham/grlib/grlib_gpio/bp_grlib_gpio_drv.h>

Sets the direction of a GPIO pin.

See bp_gpio_drv_dir_set_t for usage details.

*Prototype*

```
int  bp_grlib_gpio_dir_set  ( bp_gpio_drv_hndl_t  drv_hndl,
                              uint32_t            bank,
                              uint32_t            pin,
                              dir );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| `drv_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_grlib_gpio_dis()

<soc_comp/cobham/grlib/grlib_gpio/bp_grlib_gpio_drv.h>

Disables a GPIO interface.

See `bp_gpio_drv_dis_t` for usage details.

*Prototype*

```
int  bp_grlib_gpio_dis  ( bp_gpio_drv_hndl_t  drv_hndl );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

## bp_grlib_gpio_en()

<soc_comp/cobham/grlib/grlib_gpio/bp_grlib_gpio_drv.h>

Enables a GPIO interface.

See `bp_gpio_drv_en_t` for usage details.

*Prototype*

```
int  bp_grlib_gpio_en  ( bp_gpio_drv_hndl_t  drv_hndl );
```

*Attributes*

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

*Parameters*

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

*Returned Errors*
RTNC_SUCCESS
RTNC_FATAL

Function

# bp_grlib_gpio_is_en()

<soc_comp/cobham/grlib/grlib_gpio/bp_grlib_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_grlib_gpio_is_en  ( bp_gpio_drv_hndl_t  drv_hndl,
                            bool *              p_is_en );
```

*Attributes*

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

*Parameters*

| | |
|---|---|
| drv_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_grlib_gpio_drv_def_t

<soc_comp/cobham/grlib/grlib_gpio/bp_grlib_gpio_drv.h>

GRLIB 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

16

# 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

17

# 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

18

# 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

19

# 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

20

# 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

21

# Document Revision History

The revision history of the BASEplatform user manual and reference manuals can be found within the BASEplatform source package.