JBLopen

Embedded Software Insight

BASEplatform API Reference Manual

RM0001

July 9, 2019

Contents

Chapter

1 Overview

Welcome to the BASEplatform™ API reference manual. This reference manual covers the BASEplatform core API functions, data types and preprocessor definitions along with description and usage information for each API element. The core API is written in ISO/IEC 9899:1999 (C99) compliant C and designed to be portable between platforms and toolchains. Additional platform specific modules and APIs are documented in separate manuals dedicated to each supported platforms.

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.

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.

Chapter

2 Architecture

The architecture module, or ARCH module provides low-level CPU control functionalities as well as important compiler abstractions. These include CPU interrupt flag manipulation, memory barriers, endianness and compiler detection, alignment requirements, and more. The ARCH module is divided in various ports specific to a CPU and compiler combination. When necessary, additional files and API specific to certain CPU cores are also included in the ARCH module.

The current architecture and toolchain need to be selected at compile time by including the relevant port's header file in a master configuration file named `bp_arch_def_cfg.h`.

Data Type

bp_irq_flag_t

<arch/bp_arch.h>

Type used to store the CPU interrupt status flag returned by `bp_slock_acquire_irq_save()` and `bp_critical_section_enter()`.

The value returned by those functions should not be manipulated by the application.

Macro

BP_ARCH_ALIGN_MAX

<arch/bp_arch.h>

Defined by the architecture port to the largest required alignment across all the fundamental data types.

Macro

BP_ARCH_COMPILER

<arch/bp_arch.h>

Defined by the architecture port to the current compiler. The list of defined compilers can be found in `bp_arch_def.h`.

Macro

BP_ARCH_CORE_ID_GET()

<arch/bp_arch.h>

Returns the CPU id of the current core. On single core platforms, `BP_ARCH_CORE_ID_GET()` always returns 0.

Macro

BP_ARCH_CPU

<arch/bp_arch.h>

Defined by the architecture port to the current CPU architecture. The list of defined architectures can be found in `bp_arch_def.h`.

Macro

BP_ARCH_DEBUG_BREAK()

<arch/bp_arch.h>

Inserts a software breakpoint. The current CPU core will break to the debugger if supported. The result of hitting a software breakpoint with no debugger connected is platform specific but will usually trigger a form of CPU fault or exception.

Macro

BP_ARCH_ENDIAN

<arch/bp_arch.h>

Defined by the architecture port to the endianness of the current platform. The list of endianness definitions can be found in `bp_arch_def.h`.

Macro

BP_ARCH_INT_DIS()

<arch/bp_arch.h>

core's interrupts are disabled. The result can be assigned to a variable of type `bp_irq_flag_t` to save the current state of the interrupt flags.

Critical sections such as `bp_critical_section_enter()` and `bp_critical_section_exit()` or spinlocks, `bp_slock_acquire_irq_save()` and `bp_slock_release_irq_restore()` are usually preferable to unconditionally disabling and enabling interrupts.

Macro

BP_ARCH_INT_EN()

<arch/bp_arch.h>

Unconditionally enables CPU interrupts. On multi-core platforms only the current core's interrupts are enabled.

Critical sections such as `bp_critical_section_enter()` and `bp_critical_section_exit()` or spinlocks, `bp_slock_acquire_irq_save()` and `bp_slock_release_irq_restore()` are usually preferable to unconditionally disabling and enabling interrupts.

Macro

BP_ARCH_IS_CRIT()

<arch/bp_arch.h>

Returns a non-zero value if interrupts are disabled, i.e. inside a critical context.

Macro

BP_ARCH_IS_INT()

<arch/bp_arch.h>

Returns a non-zero value if called from within an interrupt service routine.

Macro

BP_ARCH_IS_INT_OR_CRIT()

<arch/bp_arch.h>

Returns a non-zero value if currently called from an interrupt service routine or if interrupts are disabled.

Macro

BP_ARCH_MB()

<arch/bp_arch.h>

Memory barrier.

Macro

BP_ARCH_PANIC()

<arch/bp_arch.h>

Panic, usually disables interrupts and breaks into an infinite loop or the debugger.

Macro

BP_ARCH_RMB()

<arch/bp_arch.h>

Read memory barrier, defaults to `BP_ARCH_MB()` for architectures without a specific read memory barrier.

Macro

BP_ARCH_SEV()

<arch/bp_arch.h>

Send event. The `BP_ARCH_SEV()` macro expands to the current architecture's send event instruction used for SMP signalling between cores. On architectures without any send event instruction this macro expands to a no-op instruction.

Macro

BP_ARCH_WFE()

<arch/bp_arch.h>

Wait for events. The `BP_ARCH_WFE()` macro expands to the current architecture's wait for event instruction used for SMP signalling between cores. On architectures without any wait for event instruction this macro expands to a no-op instruction.

The difference between a wait for event and a wait for interrupt is architecture dependent. In case there is no dedicated wait for event instruction this macro expands to `BP_ARCH_WFI()`.

Macro

BP_ARCH_WFI()

<arch/bp_arch.h>

Wait for interrupts. The `BP_ARCH_WFI()` macro expands to the current architecture's wait for interrupt instruction. On architectures without any wait for interrupt instruction this macro expands to a no-op instruction.

Macro

BP_ARCH_WMB()

<arch/bp_arch.h>

Write memory barrier, defaults to `BP_ARCH_MB()` for architectures without a specific write memory barrier.

Chapter

3

Cache Management

The cache management module enables drivers and applications to perform cache maintenance operations in a platform-independent manner. The various cache maintenance functions can be used to ensure cache coherency when handling hardware buffers, shared memory and similar operations with non-coherent masters in a SoC.

All the maintenance functions, regardless of the implementation, includes a suitable memory barrier at the start and at the end of all the cache maintenance operations. This applies even if a length of zero is passed to functions operating on a range as well as on platforms with no caches or with cache disabled.

The cache operations are not atomic and won't disable interrupts unless required by the platform. If a cache operation must not be interrupted, a critical section or spinlock should be used around the call. The cache operations are, however, thread-safe and re-entrant which means they can be used in parallel without issues.

Cache operations can take a considerable amount of time depending on the range, state of the cache and CPU/RAM performance. While they are marked as non-blocking, care should be taken not to perform excessively long operations from within an interrupt or a critical context.

Function

bp_cache_dcache_inv_all()

<cache/bp_cache.h>

Invalidates the entire data cache. The entire data cache hierarchy and unified caches will be invalidated.

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

Prototype

```
void bp_cache_dcache_inv_all ( );
```

Attributes

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

Function

bp_cache_dcache_max_line_get()

<cache/bp_cache.h>

Returns the largest effective data cache line size. Usually this would be the largest cache line size in the data cache hierarchy.

The special value 0 is returned when no cache is present or if the data cache line size is unknown.

Caches are usually assumed to be fully enabled. The return value of this function reflects the largest data cache line size as if the entire data cache hierarchy was enabled.

Prototype

```
uint32_t bp_cache_dcache_max_line_get ( );
```

Attributes

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

Returned Values Largest data cache line size in bytes if known, 0 otherwise.

Function

bp_cache_dcache_min_line_get()

<cache/bp_cache.h>

Returns the smallest effective data cache line size. Usually this would be the smallest cache line size in the data cache hierarchy.

The special value 0 is returned when no cache is present or if the data cache line size is unknown.

Caches are usually assumed to be fully enabled. The return value of this function reflects the smallest data cache line size as if the entire data cache hierarchy was enabled.

When considering the minimum alignment of DMA buffers, the largest cache line size should usually be used. See `bp_cache_dcache_max_line_get()`

Prototype

```
uint32_t bp_cache_dcache_min_line_get ( );
```

Attributes

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

Returned Values Smallest cache line size in bytes if known, 0 otherwise.

Function

bp_cache_dcache_range_clean()

<cache/bp_cache.h>

Cleans an address range from the data cache.

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

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

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

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

Prototype

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

Attributes

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

Parameters

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

Function

bp_cache_dcache_range_cleaninv()

<cache/bp_cache.h>

Cleans and invalidates an address range from the data cache.

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

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

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

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

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

Prototype

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

Attributes

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

Parameters

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

Function

bp_cache_dcache_range_inv()

<cache/bp_cache.h>

Invalidates an address range from the data cache.

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

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

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

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

Prototype

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

Attributes

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

Parameters

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

Function

bp_cache_icache_inv_all()

<cache/bp_cache.h>

Cleans the entire instruction cache. `bp_cache_icache_inv_all()` will clean the entire instruction cache hierarchy.

`bp_cache_icache_inv_all()` will not invalidate unified caches when present. It is the caller's responsibility of correctly handling any code that could be stored in the unified cache(s).

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

Prototype

```
void bp_cache_icache_inv_all ( );
```

Attributes

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

Chapter

4 Spinlocks

The spinlock module, shortened to slock, provides spinlocks and critical sections enabling atomic operations on both uni-processor and symmetric multiprocessor systems.

On uni-processor systems, the spinlocks reduces to simple critical sections, as such they can be used to write code compatible with both uni- and multi-processor.

Function

bp_critical_section_enter()

<slock/bp_slock.h>

Enters a critical section, disabling the interrupts and returning the CPU's interrupt flag state prior to the call to `bp_critical_section_enter()`. An appropriate memory barrier will be executed by the implementation to ensure proper synchronization.

The exact return value is implementation specific and should not be manipulated by the calling code.

`bp_critical_section_enter()` and `bp_critical_section_exit()` are compatible with bare-metal, single core RTOS and SMP RTOSes and can be used as a simpler alternative to spinlocks. However for maximum performance under SMP RTOSes, spinlocks are recommended.

Prototype

```
bp_irq_flag_t  bp_critical_section_enter  (  );
```

Attributes

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

Returned Values

Interrupt status flag prior to calling `bp_critical_section_enter()`.

Function

bp_critical_section_exit()

<slock/bp_slock.h>

Exits a critical section, restoring the interrupt state from the `flag` argument. An appropriate memory barrier will be executed by the implementation to ensure proper synchronization.

The exact values that `flag` can take is implementation specific and should not be manipulated by the calling code. The result of passing any value except one returned by a previous call to `bp_critical_section_enter()` is undefined.

`bp_critical_section_enter()` and `bp_critical_section_exit()` are compatible with bare-metal, single core RTOS and SMP RTOSes and can be used as a simpler alternative to spinlocks. However for maximum performance under SMP RTOSes, spinlocks are recommended.

Prototype `void bp_critical_section_exit ( );`

Attributes

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

Function

bp_slock_acquire()

<slock/bp_slock.h>

Acquires a spinlock. Under an SMP RTOS, `bp_slock_acquire()` will busy wait (spin) until the lock is available. In a single core system `bp_slock_acquire()` will be reduced to a memory barrier.

Note that `bp_slock_acquire()` will not disable interrupts which is necessary to guarantee atomicity and prevent deadlocks. `bp_slock_acquire_irq_save()` and `bp_slock_acquire_irq_dis()` can be used instead when interrupts need to be disabled.

Prototype `void bp_slock_acquire (bp_slock_t * p_lock);`

Attributes

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

Parameters `p_lock` Pointer to the spinlock.

Function

bp_slock_acquire_irq_dis()

<slock/bp_slock.h>

Acquires a spinlock and disables interrupts. Under an SMP RTOS, `bp_slock_acquire_irq_dis()` will busy wait (spin) until the lock is available. In a single core system `bp_slock_acquire_irq_dis()`

will disable interrupts and execute a memory barrier to enforce synchronization.

bp_slock_acquire_irq_dis() and bp_slock_release_irq_en() used in pairs will unconditionally disable and enable interrupts on entry and exit of the critical section. They can be used as a leaner version of spinlocks when saving the interrupt flag state is unnecessary. Otherwise bp_slock_acquire_irq_save() and bp_slock_release_irq_restore() should be used when calling from within a critical section where interrupts could be disabled.

Prototype `void bp_slock_acquire_irq_dis ( bp_slock_t * p_lock );`

Attributes

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

Parameters `p_lock` Pointer to the spinlock.

Function

bp_slock_acquire_irq_save()

<slock/bp_slock.h>

Acquires a spinlock, disables interrupts and returns the CPU's interrupt flag state. Under an SMP RTOS, bp_slock_acquire_irq_save() will busy wait (spin) until the lock is available. In a single core system bp_slock_acquire_irq_save() will disable the interrupts and return the interrupt status flag as well as executing a memory barrier to enforce synchronization.

The exact return value is implementation specific and should not be manipulated by the calling code.

Prototype `bp_irq_flag_t bp_slock_acquire_irq_save ( bp_slock_t * p_lock );`

Attributes

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

Parameters `p_lock` Pointer to the spinlock.

Returned Values Interrupt status flag prior to calling bp_slock_acquire_irq_save().

Function

bp_slock_release()

<slock/bp_slock.h>

Releases a spinlock. Under an SMP RTOS, bp_slock_release() will release the spinlock and signal other cores which may be waiting on the lock. In a single core system bp_slock_release() will be reduced to a memory barrier.

Prototype `void bp_slock_release (bp_slock_t * p_lock);`

Attributes

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

Parameters `p_lock` Pointer to the spinlock.

Function

bp_slock_release_irq_en()

<slock/bp_slock.h>

Releases a spinlock and enables interrupts. Under an SMP RTOS, `bp_slock_release_irq_en()` will release the spinlock and signal other cores which may be waiting on the lock. In a single core system `bp_slock_release_irq_en()` will enable interrupts and execute a memory barrier to enforce synchronization.

`bp_slock_acquire_irq_dis()` and `bp_slock_release_irq_en()` in a pair will unconditionally disable and enable interrupts on entry and exit of the critical section. They can be used as a leaner version of spinlocks when saving the interrupt flag state is unnecessary. Otherwise `bp_slock_acquire_irq_save()` and `bp_slock_release_irq_restore()` should be used when calling from within a critical section where interrupts are disabled.

Prototype `void bp_slock_release_irq_en (bp_slock_t * p_lock);`

Attributes

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

Parameters `p_lock` Pointer to the spinlock.

Function

bp_slock_release_irq_restore()

<slock/bp_slock.h>

Releases a spinlock and restores the interrupt state. Under an SMP RTOS, `bp_slock_release_irq_restore()` will release the spinlock and signal other cores which may be waiting on the lock. In a single core system `bp_slock_release_irq_restore()` will restore the interrupts as well as execute a memory barrier to enforce synchronization.

The exact values that `flag` can take is implementation specific and should not be manipulated by the calling code. The result of passing any value except one returned by a previous call to `bp_slock_acquire_irq_save()` is undefined.

Prototype

```
void bp_slock_release_irq_restore ( bp_slock_t *    p_lock,
                                    bp_irq_flag_t   flag );
```

Attributes

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

Parameters

`p_lock`	Pointer to the spinlock.
`flag`	Saved interrupt flag to restore.

Data Type

bp_slock_t

<slock/bp_slock.h>

Spinlock datatype. Any spinlock variable should be cleared by setting them* to 0 prior to use.

Chapter

5 Time

The time module is responsible for the system's primary timebase as well as providing high resolution time delays and time measurements. It is also the time base used by the generic timer module.

When running with an RTOS, the time module usually provides the kernel reference tick, with support for dynamic or tickless mode for RTOSes that supports it.

Additionally, when running within and RTOS, the time delays provided by the time module are implemented independently of the kernel software timers and delays. As such, they usually support a higher resolution than the kernel offers and can be used where fine timing is required.

Function

bp_time_freq_get()

<time/bp_time.h>

Returns the frequency of the primary time base.

This function cannot fail and in normal operation should always return a non-zero value. In special cases where the frequency is unknown, 0 is returned.

Prototype

```
uint32_t bp_time_freq_get ( );
```

Attributes

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

Returned Values

Frequency of the primary time base in hertz.

Function

bp_time_get()

<time/bp_time.h>

Returns the raw value of the primary time base counter.

This function cannot fail and in normal operation will always return a non-zero value. In special cases where there is no active timebase, 0 is returned.

Prototype
```
uint64_t bp_time_get ( );
```

Attributes

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

Returned Values Raw 64-bit value of the primary counter.

Function

bp_time_get32()

<time/bp_time.h>

Returns the raw value of the primary time base counter, 32-bit version. The value returned is the same as would result from truncating the returned value of `bp_time_get()` to the least significant 32 bits.

This function cannot fail and in normal operation will always return a non-zero value. In special cases where there is no active timebase, 0 is returned.

Prototype
```
uint32_t bp_time_get32 ( );
```

Attributes

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

Returned Values Raw 32-bit value of the primary counter.

Function

bp_time_get_ms()

<time/bp_time.h>

Returns the current value of the primary time base counter in milliseconds.

This function cannot fail and in normal operation will always return a non-zero value. In special cases where there is no active timebase, 0 is returned.

Prototype `uint64_t bp_time_get_ms ( );`

Attributes

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

Returned Values 64-bit counter value in milliseconds.

Function

bp_time_get_ms32()

<time/bp_time.h>

Returns the current value of the primary time base counter in milliseconds, 32-bit version.

This function cannot fail and in normal operation will always return a non-zero value. In special cases where there is no active timebase, 0 is returned.

Prototype `uint32_t bp_time_get_ms32 ( );`

Attributes

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

Returned Values 32-bit counter value in milliseconds.

Function

bp_time_get_ns()

<time/bp_time.h>

Returns the current value of the primary time base counter in nanoseconds.

This function cannot fail and in normal operation will always return a non-zero value. In special cases where there is no active timebase, 0 is returned.

Prototype `uint64_t bp_time_get_ns ( );`

Attributes

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

Returned Values 64-bit counter value in nanoseconds.

Function

bp_time_get_ns32()

<time/bp_time.h>

Returns the current value of the primary time base counter in nanoseconds. 32-bit version.

This function cannot fail and in normal operation will always return a non-zero value. In special cases where there is no active timebase, 0 is returned.

Prototype

```
uint32_t bp_time_get_ns32 ( );
```

Attributes

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

Returned Values 32-bit counter value in nanoseconds.

Function

bp_time_halt()

<time/bp_time.h>

Halts the primary time base. The primary timebase is halted until `bp_time_resume()` is called.

Halting and resuming the primary time base should be done for testing and debugging purpose only.

Prototype

```
int bp_time_halt ( );
```

Attributes

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

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_time_init()

<time/bp_time.h>

Initializes the time module and the primary time base.

bp_time_init() should be called before any other services that is dependent on the system timebase are used.

bp_time_init() should only be called once. The result of subsequent calls after the first is undefined.

Prototype `int bp_time_init ( );`

Attributes

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

Returned Errors
RTNC_SUCCESS
RTNC_FATAL

Function

bp_time_ms_to_raw()

<time/bp_time.h>

Converts milliseconds to the raw time base unit.

This function cannot fail and in normal operation should always return a non-zero value for a non-zero input. In special cases where the frequency is unknown, 0 is returned.

Prototype `uint64_t bp_time_ms_to_raw ( );`

Attributes

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

Parameters

Returned Values Time value in the raw time base unit.

Function

bp_time_ms_to_raw32()

<time/bp_time.h>

Converts milliseconds to the raw time base unit, 32-bit version.

This function cannot fail and in normal operation should always return a non-zero value for a non-zero input. In special cases where the frequency is unknown, 0 is returned.

Prototype `uint32_t bp_time_ms_to_raw32 (uint32_t time_ms );`

Attributes

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

Parameters `time_ms` Time value in milliseconds.

Returned Values Time value in the raw time base unit.

Function

bp_time_ns_to_raw()

<time/bp_time.h>

Converts nanoseconds to the raw time base unit.

This function cannot fail and in normal operation should always return a non-zero value for a non-zero input. In special cases where the frequency is unknown, 0 is returned.

Prototype

```
uint64_t bp_time_ns_to_raw (uint64_t time_ns);
```

Attributes

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

Parameters `time_ns` Time value in milliseconds.

Returned Values Time value in the raw time base unit.

Function

bp_time_ns_to_raw32()

<time/bp_time.h>

Converts nanoseconds to the raw time base unit, 32-bit version.

This function cannot fail and in normal operation should always return a non-zero value for a non-zero input. In special cases where the frequency is unknown, 0 is returned.

Prototype

```
uint32_t bp_time_ns_to_raw32 (uint32_t time_ns);
```

Attributes

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

Parameters `time_ns` Time value in milliseconds.

Returned Values Time value in the raw time base unit.

Function

bp_time_raw_to_ms()

<time/bp_time.h>

Converts a time value from the raw time base unit to milliseconds.

This function cannot fail and in normal operation should always return a non-zero value for a non-zero input. In special cases where the frequency is unknown, 0 is returned.

Prototype

```
uint64_t bp_time_raw_to_ms (uint64_t time_raw);
```

Attributes

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

Parameters `time_raw` Time value in the unit of the system time base.

Returned Values Time value in milliseconds.

Function

bp_time_raw_to_ms32()

<time/bp_time.h>

Converts a time value in the raw time base unit to milliseconds, 32-bit version.

This function cannot fail and in normal operation should always return a non-zero value for a non-zero input. In special cases where the frequency is unknown, 0 is returned.

Prototype

```
uint32_t bp_time_raw_to_ms32 (uint32_t time_raw);
```

Attributes

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

Parameters `time_raw` Time value in the unit of the system time base.

Returned Values Time value in milliseconds.

Function

bp_time_raw_to_ns()

<time/bp_time.h>

Converts a time value in the raw time base unit to nanoseconds.

This function cannot fail and in normal operation should always return a non-zero value for a non-zero input. In special cases where the frequency is unknown, 0 is returned.

Prototype

```
uint64_t bp_time_raw_to_ns (uint64_t time_raw);
```

Attributes

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

Parameters `time_raw` Time value in the unit of the system time base.

Returned Values Time value in nanoseconds.

Function

bp_time_raw_to_ns32()

<time/bp_time.h>

Converts a time value in the raw time base unit to nanoseconds, 32-bit version.

This function cannot fail and in normal operation should always return a non-zero value for a non-zero input. In special cases where the frequency is unknown, 0 is returned.

Prototype

```
uint32_t bp_time_raw_to_ns32 (uint32_t time_raw);
```

Attributes

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

Parameters `time_raw` Time value in the unit of the system time base.

Returned Values Time value in nanoseconds.

Function

bp_time_resume()

<time/bp_time.h>

Resumes the primary time base. Resumes the primary time base from where it was stopped by `bp_time_halt()`. The result of calling resume when the timebase isn't halted is undefined.

Halting and resuming the primary time base should be done for testing and debugging purpose only.

Prototype

```
int bp_time_resume ( );
```

Attributes

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

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Function

bp_time_sleep()

<time/bp_time.h>

Sleeps for a specified amount of time in the platform's raw timebase unit.

The wait method is chosen by the underlying implementation and will usually be a busy loop for small delays and a timer interrupt for larger delays.

The amount of time slept is guaranteed to be at least the specified amount.

`bp_time_sleep()` should not be called from an interrupt service routine or with the interrupts disabled. `bp_time_sleep_busy()` should be used instead. However long delays within interrupt service routines or critical section could have a negative impact on the system performance and should be used sparingly.

Prototype

```
int bp_time_sleep (uint64_t time_raw);
```

Attributes

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

Parameters

time_raw	Amount of time to sleep in the platform's raw timebase unit.

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Function

bp_time_sleep32()

<time/bp_time.h>

Sleeps for a specified amount of time in the platform's raw timebase unit, 32-bit version.

The wait method is chosen by the underlying implementation and will usually be a busy loop for small delays and a timer interrupt for larger delays.

The amount of time slept is guaranteed to be at least the specified amount.

`bp_time_sleep32()` should not be called from an interrupt service routine or with the interrupts disabled. `bp_time_sleep_busy32()` should be used instead. However long delays within interrupt service routines or critical section could have a negative impact on the system performance and should be used sparingly.

Prototype `int bp_time_sleep32 (uint32_t time_raw);`

Attributes

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

Parameters

`time_raw` Amount of time to sleep in the platform's raw timebase unit.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_time_sleep_busy()

<time/bp_time.h>

Busy wait for a specified amount of time.

Contrary to `bp_time_sleep()`, `bp_time_sleep_busy()` will always perform a busy loop for short and long delays. As such `bp_time_sleep_busy()` can always be called from an interrupt service routine or with the interrupts disabled.

Interrupts are not disabled while waiting unless they are disabled prior to calling `bp_time_sleep_busy()`.

The amount of time slept is guaranteed to be at least the specified amount.

Long busy delays should usually be avoided, especially when running under an RTOS.

Prototype `int bp_time_sleep_busy (uint64_t time_raw);`

Attributes

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

Parameters `time_raw` Amount of time to sleep in the raw timebase unit.

Returned Errors
`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_time_sleep_busy32()

<time/bp_time.h>

Busy wait for a specific amount of time, 32-bit version.

Contrary to `bp_time_sleep()`, `bp_time_sleep_busy32()` will always perform a busy loop for short and long delays. As such `bp_time_sleep_busy32()` can always be called from an interrupt service routine or with the interrupts disabled.

Interrupts are not disabled while waiting unless they are disabled prior to calling `bp_time_sleep_busy32()`.

The amount of time slept is guaranteed to be at least the specified amount.

Long busy delays should usually be avoided, especially when running under an RTOS.

Prototype

```
int bp_time_sleep_busy32 (uint32_t time_raw);
```

Attributes

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

Parameters `time_raw` Amount of time to sleep in the raw timebase unit.

Returned Errors
`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_time_sleep_busy_ms()

<time/bp_time.h>

Busy wait for a specific amount of time in milliseconds.

Contrary to `bp_time_sleep()`, `bp_time_sleep_busy_ms()` will always perform a busy loop for short and long delays. As such `bp_time_sleep_busy_ms()` can always be called from an interrupt service routine or with the interrupts disabled.

Interrupts are not disabled while waiting unless they are disabled prior to calling `bp_time_sleep_busy_ms()`.

The amount of time slept is guaranteed to be at least the specified amount.

Long busy delays should usually be avoided, especially when running under an RTOS.

Prototype `int bp_time_sleep_busy_ms (uint32_t time_ms);`

Attributes

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

Parameters `time_ms` Amount of time to sleep in milliseconds.

Returned Errors
`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_time_sleep_busy_ns()

<time/bp_time.h>

Busy wait for a specific amount of time in nanoseconds.

Contrary to `bp_time_sleep()`, `bp_time_sleep_busy_ns()` will always perform a busy loop for short and long delays. As such `bp_time_sleep_busy_ns()` can always be called from an interrupt service routine or with the interrupts disabled.

Interrupts are not disabled while waiting unless they are disabled prior to calling `bp_time_sleep_busy_ns()`.

The amount of time slept is guaranteed to be at least the specified amount.

Long busy delays should usually be avoided, especially when running under an RTOS.

Prototype `int bp_time_sleep_busy_ns (uint32_t time_ns);`

Attributes

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

Parameters `time_ns` Amount of time to sleep in nanoseconds.

Returned Errors
`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_time_sleep_ms()

<time/bp_time.h>

Sleeps for a specified amount of time in milliseconds.

The wait method is chosen by the underlying implementation and will usually be a busy loop for small delays and a timer interrupt for larger delays.

The amount of time slept is guaranteed to be at least the specified amount.

`bp_time_sleep_ms()` should not be called from an interrupt service routine or with the interrupts disabled. `bp_time_sleep_busy_ms()` should be used instead. However long delays within interrupt service routines or critical section could have a negative impact on the system performance and should be used sparingly.

Prototype

```
int bp_time_sleep_ms (uint32_t time_ms);
```

Attributes

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

Parameters

`time_ms`	Amount of time to sleep in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_time_sleep_ns()

<time/bp_time.h>

Sleeps for a specified amount of time in nanoseconds.

The wait method is chosen by the underlying implementation and will usually be a busy loop for small delays and a timer interrupt for larger delays.

The amount of time slept is guaranteed to be at least the specified amount.

`bp_time_sleep_ns()` should not be called from an interrupt service routine or with the interrupts disabled. `bp_time_sleep_busy_ns()` should be used instead. However long delays within interrupt service routines or critical section could have a negative impact on the system performance and should be used sparingly.

Prototype

```
int bp_time_sleep_ns (uint32_t time_ns);
```

Attributes

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

Parameters `time_ns` Amount of time to sleep in nanoseconds.

Returned Errors RTNC_SUCCESS
RTNC_FATAL

Function

bp_time_sleep_yield()

<time/bp_time.h>

Yields and wait for a specific amount of time in the raw timebase unit.

Contrary to bp_time_sleep(), bp_time_sleep_yield() will always perform an interrupt based delay even for small delays. When running with an RTOS it is guaranteed to generate a context switch.

bp_time_sleep_yield() must not be called from an interrupt service routine or with the interrupts disabled.

The amount of time slept is guaranteed to be at least the specified amount.

Prototype
```
int bp_time_sleep_yield (uint64_t time_raw);
```

Attributes

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

Parameters `time_raw` Amount of time to sleep in the raw timebase unit.

Returned Errors RTNC_SUCCESS
RTNC_FATAL

Function

bp_time_sleep_yield32()

<time/bp_time.h>

Yields and wait for a specific amount of time, 32-bit version.

Contrary to bp_time_sleep32(), bp_time_sleep_yield32() will always perform an interrupt based delay even for small delays. When running with an RTOS it is guaranteed to generate a context switch.

bp_time_sleep_yield32() must not be called from an interrupt service routine or with the interrupts disabled.

The amount of time slept is guaranteed to be at least the specified amount.

Prototype
```
int bp_time_sleep_yield32 (uint32_t time_raw);
```

Attributes

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

Parameters `time_raw` Amount of time to sleep in the raw timebase unit.

Returned Errors
RTNC_SUCCESS
RTNC_FATAL

Function

bp_time_sleep_yield_ms()

<time/bp_time.h>

Yields and wait for a specific amount of time in milliseconds.

Contrary to bp_time_sleep_ms(), bp_time_sleep_yield_ms() will always perform an interrupt based delay even for small delays. When running with an RTOS it is guaranteed to generate a context switch.

bp_time_sleep_yield_ms() must not be called from an interrupt service routine or with the interrupts disabled.

The amount of time slept is guaranteed to be at least the specified amount.

Prototype

```
int bp_time_sleep_yield_ms (uint32_t time_ms);
```

Attributes

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

Parameters `time_ms` Amount of time to sleep in milliseconds.

Returned Errors
RTNC_SUCCESS
RTNC_FATAL

Function

bp_time_sleep_yield_ns()

<time/bp_time.h>

Yields and wait for a specific amount of time in nanoseconds.

Contrary to bp_time_sleep(), bp_time_sleep_yield_ns() will always perform an interrupt based delay even for small delays. When running with an RTOS it is guaranteed to generate a context switch.

bp_time_sleep_yield_ns() must not be called from an interrupt service routine or with the interrupts disabled.

The amount of time slept is guaranteed to be at least the specified amount.

Prototype

```
int bp_time_sleep_yield_ns (uint32_t time_ns);
```

Attributes

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

Parameters

`time_ns` Amount of time to sleep in nanoseconds.

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Chapter

6 Timers

The timer module offers generic high resolution timers based on a hardware time base provided by the time module. Being independent of any RTOS the timers are available across all platforms supported by the BASEplatform, including bare-metal. In addition, being derived from the primary timebase, the generic timer's resolution is usually higher than the kernel's software timers.

Function

bp_timer_create()

<timer/bp_timer.h>

Creates a new timer. When successful the newly created timer handle is returned through the p_hndl argument.

When returning with an RTNC_NO_RESOURCE error, it is guaranteed that no resource has been permanently allocated to prevent leaking.

Prototype

```
int bp_timer_create ( bp_timer_hndl_t * p_hndl );
```

Attributes

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

Parameters

p_hndl Pointer to the returned timer handle.

Returned Errors

RTNC_SUCCESS
RTNC_NO_RESOURCE
RTNC_FATAL

Function

bp_timer_destroy()

<timer/bp_timer.h>

Destroys a timer. The timer is either returned to a pool of timers that can be reused or freed if the memory allocator allows freeing memory.

Prototype `int bp_timer_destroy ( bp_timer_hndl_t hndl );`

Attributes

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

Parameters `hndl` Handle of the timer to destroy.

Returned Errors
`RTNC_SUCCESS`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

bp_timer_halt()

<timer/bp_timer.h>

Halts the BASEplatform timer processing. This function should be used for testing and debugging only to temporarily halt timer processing until `bp_timer_resume()` is called.

Prototype `int bp_timer_halt ( );`

Attributes

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

Returned Errors
`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_timer_init()

<timer/bp_timer.h>

Initializes the timer facility. `bp_timer_init()` should be called before any other services that are dependent on the timers are used. In most cases, the time module should be initialized before the timer module. See `bp_time_init()` for details.

`bp_timer_init()` should only be called once. The result of subsequent calls after the first is undefined.

Prototype

```
int bp_timer_init ( );
```

Attributes

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

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Function

bp_timer_restart()

<timer/bp_timer.h>

Restarts a timer. The timer will be restarted and set to expire after `time_raw` has passed in the system's primary timebase from the last time it expired. Upon expiration, the original callback will be called.

To start a timer from the current time instead of the last expiration `bp_timer_start()` should be used.

Prototype

```
int bp_timer_restart ( bp_timer_hndl_t  hndl,
                       uint64_t         time_raw,
                       void *           p_arg );
```

Attributes

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

Parameters

`hndl`	Handle of the timer to restart.
`time_raw`	Time to wait in the raw timebase unit.
`p_arg`	Optional argument passed to the timer callback.

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Function

bp_timer_restart_ms()

<timer/bp_timer.h>

Restarts a timer. The timer will be started and set to expire after `time_ms` milliseconds has passed from the last time it expired. Upon expiration, the original callback will be called.

To start a timer from the current time instead of the last expiration `bp_timer_start_ms()` should be used.

Prototype

```
int bp_timer_restart_ms ( bp_timer_hndl_t hndl,
                          uint32_t        time_ms,
                          void *          p_arg );
```

Attributes

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

Parameters

`hndl`	Handle of the timer to restart.
`time_ms`	Time to wait in milliseconds.
`p_arg`	Optional argument passed to the timer callback.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_timer_restart_ns()

<timer/bp_timer.h>

Restarts a timer. The timer will be started and set to expire after `time_ns` nanoseconds has passed from the last time it expired. Upon expiration, the original callback will be called.

To start a timer from the current time instead of the last expiration `bp_timer_start_ns()` should be used.

Prototype

```
int bp_timer_restart_ns ( bp_timer_hndl_t hndl,
                          uint32_t        time_ns,
                          void *          p_arg );
```

Attributes

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

Parameters

`hndl`	Handle of the timer to restart.
`time_ns`	Time to wait in nanoseconds.
`p_arg`	Optional argument passed to the timer callback.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_timer_resume()

<timer/bp_timer.h>

Resumes the BASEplatform timer processing. This function should be used for testing and debugging only to resume timer processing after a call to `bp_timer_halt()`.

Prototype

```
int bp_timer_resume ( );
```

Attributes

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

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_timer_start()

<timer/bp_timer.h>

Starts a timer. The timer will be started and set to expire after the specified amount of time has passed on the system raw timebase. Upon expiration `p_callback` will be called with `p_arg` passed as an optional argument.

See `bp_timer_cb_t` for details about the callback functionality.

Prototype

```
int bp_timer_start ( bp_timer_hndl_t  hndl,
                     uint64_t         time_raw,
                     void *           p_arg );
```

Attributes

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

Parameters

`hndl`	Handle of the timer to start.
`time_raw`	Timer delay in the raw timebase unit.
`p_arg`	Optional argument passed to the timer callback.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_timer_start_ms()

<timer/bp_timer.h>

Starts a timer. The timer will be started and set to expire after `time_ms` has passed in milliseconds. Upon expiration `p_callback` will be called with `p_arg` passed as an optional argument.

See `bp_timer_cb_t` for details about the callback functionality.

Prototype

```
int bp_timer_start_ms ( bp_timer_hndl_t hndl,
                        uint32_t        time_ms,
                        void *          p_arg );
```

Attributes

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

Parameters

`hndl`	Handle of the timer to start.
`time_ms`	Timer delay in milliseconds.
`p_arg`	Optional argument passed to the timer callback.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_timer_start_ns()

<timer/bp_timer.h>

Starts a timer. The timer will be started and set to expire after `time_ns` has passed in nanoseconds. Upon expiration `p_callback` will be called with `p_arg` passed as an optional argument.

See `bp_timer_cb_t` for details about the callback functionality.

Prototype

```
int bp_timer_start_ns ( bp_timer_hndl_t hndl,
                        uint32_t        time_ns,
                        void *          p_arg );
```

Attributes

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

Parameters

`hndl`	Handle of the timer to start.
`time_ns`	Timer delay in nanoseconds.
`p_arg`	Optional argument passed to the timer callback.

Returned Errors RTNC_SUCCESS
RTNC_FATAL

Function

bp_timer_stop()

<timer/bp_timer.h>

Stops a timer. The timer will be stopped without calling its expiration callback. If the timer is not started or has expired already bp_timer_stop() will return RTNC_SUCCESS without affecting the timer.

Prototype

```
int bp_timer_stop ( bp_timer_hndl_t hndl );
```

Attributes

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

Parameters

hndl	Handle of the timer to stop.

Returned Errors RTNC_SUCCESS
RTNC_FATAL

Function

bp_timer_target_get()

<timer/bp_timer.h>

Returns the timer target in the raw timebase unit. If successful the timer's target expiration time is returned through p_target;

Prototype

```
int bp_timer_target_get ( bp_timer_hndl_t hndl,
                          uint64_t *      p_target );
```

Attributes

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

Parameters

hndl	Handle of the timer to query.
p_target	Pointer to the returned target time.

Returned Errors RTNC_SUCCESS
RTNC_FATAL

Data Type

bp_timer_action_t

<timer/bp_timer.h>

Action that can be returned from a timer's callback function. See `bp_timer_cb_t` for details.

Values

`BP_TIMER_STOP`	Stops the timer.
`BP_TIMER_PERIODIC`	Restarts a timer with the same settings counting from the last timer expiry.
`BP_TIMER_RESTART`	Restarts a timer with new settings.

Data Type

bp_timer_cb_t

<timer/bp_timer.h>

Timer callback function signature type. The `hndl` argument is a handle to the expired timer. The argument `p_arg` is set when creating the timer, see `bp_timer_create()` for details.

Three actions are possible when returning.

- `BP_TIMER_STOP` Stops the timer, removing it from the active timer list.
- `BP_TIMER_PERIODIC` Restart the timer using the same settings starting from the last timer expiry.
- `BP_TIMER_RESTART` Restart the timer with new settings.

Prototype

```
bp_timer_action_t bp_timer_cb_t ( bp_timer_hndl_t hndl,
                                  void *          p_arg );
```

Attributes

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

Parameters

`hndl`	
`p_arg`	Callback argument set when creating the timer.

Returned Values

Action of type `bp_timer_action_t` to perform with the timer once returning.

Data Type

bp_timer_hndl_t

<timer/bp_timer.h>

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

Members

`p_tmr`	`bp_timer_t *`	Pointer to the internal timer structure.

Chapter

7 Platform Clocks

The clock module offers a unified clock control interface to other BASEplatform modules and drivers as well as the application across different platforms. This enables drivers and application code to be aware of core and peripherals clock speed, state and control clock gating using a portable API.

The mapping of clock id and clock gates is SoC specific, details can be found in the platform's documentation.

Function

bp_clock_dis()

<clock/bp_clock.h>

Disables a clock gate.

Disabling an already disabled clock should be without side effects.

Clock and gate id are implementation specific, the list of clocks and gates can be found in the platform's documentation.

It is implementation defined whether or not a clock and gate id with the same numerical value corresponds to the same clock line.

Prototype

```
int bp_clock_dis (int clock_gate_id);
```

Attributes

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

Parameters

`clock_gate_id`	Clock gate id of the clock gate to disable.

Returned Errors `RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_clock_en()

<clock/bp_clock.h>

Enables a clock gate.

Enabling an already enabled clock should be without side effects.

Clocks and gates id are implementation specific, the clock and gate lines can be found in the platform's documentation.

It is implementation defined whether or not a clock and gate id with the same numerical value corresponds to the same clock line.

Prototype

```
int bp_clock_en (int clock_gate_id);
```

Attributes

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

Parameters

`clock_gate_id` Clock gate id of the clock gate to enable.

Returned Errors `RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_clock_freq_get()

<clock/bp_clock.h>

Returns the clock frequency of clock `clock_id` when known, otherwise 0 is returned.

When a clock is gated, `bp_clock_freq_get()` will return the clock frequency as if the clock wasn't gated, if possible, instead of 0. `bp_clock_is_en()` can be called to query if the clock is gated or not.

Clocks and gates id are implementation specific, the clock and gate mapping can be found in the platform's documentation.

It is implementation defined whether or not a clock and gate id with the same numerical value corresponds to the same clock line.

Prototype

```
int bp_clock_freq_get (int        clock_id,
                       uint32_t * p_freq);
```

Attributes

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

Parameters

`clock_id` Clock id of the clock to query.
`p_freq` Returned frequency in hertz.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_clock_gate_id_is_valid()

<clock/bp_clock.h>

Checks if a clock gate id is valid for the current platform. The validity of the clock gate `id` is returned as the function return value for brevity since the function cannot fail.

Prototype

```
bool bp_clock_gate_id_is_valid ( clock_id );
```

Attributes

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

Parameters

`clock_id` Clock gate id to check.

Returned Values

`true` if the clock gate id is valid, `false` otherwise.

Function

bp_clock_id_is_valid()

<clock/bp_clock.h>

Checks if a clock id is valid for the current platform. The validity of the clock `id` is returned as the function return value for brevity since the function cannot fail.

Prototype

```
bool bp_clock_id_is_valid (int clock_id );
```

Attributes

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

Parameters

`clock_id` Clock id to check.

Returned Values `true` if the clock id is valid, `false` otherwise.

Function

bp_clock_is_en()

<clock/bp_clock.h>

Returns the enabled or disabled state of a clock gate.

Clocks and gates id are implementation specific, the list of clocks and gate lines can be found in the platform's documentation.

It is implementation defined whether or not a clock and gate id with the same numerical value corresponds to the same clock line.

Prototype

```
int  bp_clock_is_en  ( int     clock_gate_id,
                       bool *  p_state );
```

Attributes

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

Parameters

`clock_gate_id`	Clock gate id of the clock gate to query.
`p_state`	Returned state, `true` if enabled `false` otherwise.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Chapter

8 Platform Resets

The reset module provides a unified reset interface to other BASEplatform modules and drivers as well as the application. This enables drivers and application code to control peripheral reset lines using a portable API.

Peripheral reset ids are platform specific, the exact mapping can be found in the platform documentation.

Not all platforms have a way to control individual peripheral reset lines. With those platforms the API calls are still defined but have no effect.

Function

bp_periph_reset_assert()

<reset/bp_reset.h>

Asserts a peripheral reset.

Asserting an already asserted reset lines should be without side effects.

Peripheral reset ids are implementation specific, the list of reset lines can be found in the platform's documentation.

Prototype

```
int bp_periph_reset_assert ( int periph_reset_id );
```

Attributes

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

Parameters

`periph_reset_id`	Peripheral reset line id to assert.

Returned Errors `RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_periph_reset_deassert()

<reset/bp_reset.h>

Deasserts a peripheral reset.

Deasserting an already deasserted reset lines should be without side effects.

Peripheral reset ids are implementation specific, the list of peripheral reset lines can be found in the platform's documentation.

Prototype

```
int bp_periph_reset_deassert ( int periph_reset_id );
```

Attributes

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

Parameters

`periph_reset_id` Peripheral reset line id to deassert.

Returned Errors `RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_periph_reset_id_is_valid()

<reset/bp_reset.h>

Checks if a peripheral reset id is valid for the current platform. The validity of the reset `periph_reset_id` is returned as the function return value for brevity since the function cannot fail.

Prototype

```
bool bp_periph_reset_id_is_valid ( int periph_reset_id );
```

Attributes

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

Parameters

`periph_reset_id` Peripheral reset id to check.

Returned Values `true` if the peripheral reset id is valid, `false` otherwise.

Function

bp_periph_reset_is_asserted()

<reset/bp_reset.h>

Returns the state of a peripheral reset line. If successful, the state of the reset line `periph_reset_id` will be returned through `p_is_asserted`.

Peripheral reset ids are implementation specific, the list of peripheral reset lines can be found in the platform's documentation.

Prototype

```
int  bp_periph_reset_is_asserted  ( int     periph_reset_id,
                                    bool *  p_is_asserted );
```

Attributes

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

Parameters

`periph_reset_id`	Peripheral reset line id to query.
`p_is_asserted`	Pointer to the returned state, set to `true` if the peripheral is in reset, `false` otherwise.

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Chapter

9 Interrupt Management

The interrupt management module handles the platform's interrupt controller as well as the list of interrupt service routines, also known as interrupt handlers.

By default, interrupts are initialized to their lowest priority. The interrupt default type, either edge or level, as well as its default polarity are implementation dependent.

When registering an interrupt, it is automatically configured to target the current core on multi-core architectures.

Function

bp_int_arg_get()

<int/bp_int.h>

Returns the argument of the current interrupt.

Prototype

```
void * bp_int_arg_get ( );
```

Attributes

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

Returned Values

Argument of the current interrupt.

Function

bp_int_dis()

<int/bp_int.h>

Disables the interrupt controller. This function should be used for testing and debugging only. The interrupt controller is usually enabled automatically after it is initialized and stays enabled permanently until the platform is shutdown or reset. To temporarily disable and re-enable interrupts the architecture interrupt disable functions should be used. See BP_ARCH_INT_DIS() and BP_ARCH_INT_EN() for details.

To enable or disable a single interrupt id use bp_int_src_en() and bp_int_src_dis().

Prototype

```
int bp_int_dis ( );
```

Attributes

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

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Function

bp_int_en()

<int/bp_int.h>

Enables the interrupt controller. This function should be used for testing and debugging only. The interrupt controller is usually enabled automatically after it is initialized and stays enabled permanently until a platform shutdown or reset is performed. To temporarily disable and re-enable interrupts the architecture interrupt disable functions should be used. See BP_ARCH_INT_DIS() and BP_ARCH_INT_EN() for details.

To enable or disable a single interrupt id use bp_int_src_en() and bp_int_src_dis().

Prototype

```
int bp_int_en ( );
```

Attributes

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

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Function

bp_int_id_is_valid()

<int/bp_int.h>

Checks if an interrupt id is valid for the current platform. The validity of the interrupt `id` is returned as the function return value for brevity since the function cannot fail.

Prototype

```
bool bp_int_id_is_valid ( int id );
```

Attributes

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

Parameters

`id` Interrupt id to check.

Returned Values

`true` if the interrupt id is valid, `false` otherwise.

Function

bp_int_init()

<int/bp_int.h>

Initializes and enables the platform's interrupt controller. `bp_int_init()` should usually be called early in the platform initialization process before the OS or bare-metal environment is initialized.

Most interrupt controller implementations will use statically allocated resources at compile time. For the implementations that do require run-time allocation, `bp_int_init()` could return an `RTNC_NO_RESOURCE` error. See the implementation's documentation for details.

Prototype

```
int bp_int_init ( );
```

Attributes

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

Returned Errors

`RTNC_SUCCESS`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

bp_int_prio_get()

<int/bp_int.h>

Retrieves the priority of an interrupt source. The priority of the interrupt source will be returned through p_priority.

The range, meaning and order of interrupt priorities is implementation defined and usually follows the platform's convention.

Prototype

```
int bp_int_prio_get ( int        id,
                      uint32_t * p_priority );
```

Attributes

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

Parameters

id	Interrupt id to query.
p_priority	Pointer to the returned interrupt priority.

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Function

bp_int_prio_highest_get()

<int/bp_int.h>

Returns the numerical value of the highest possible interrupt priority.

Prototype

```
uint32_t bp_int_prio_highest_get ( );
```

Attributes

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

Returned Values

Numerical value of the highest interrupt priority.

Function

bp_int_prio_lowest_get()

<int/bp_int.h>

Returns the numerical value of the lowest possible interrupt priority.

Prototype `uint32_t bp_int_prio_lowest_get ( );`

Attributes

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

Returned Values Numerical value of the lowest interrupt priority.

Function

bp_int_prio_next_get()

<int/bp_int.h>

Returns the numerical value of the next interrupt priority level higher than `prio`.

In case the next highest priority level is higher than the maximum possible, the maximum interrupt priority level will be returned.

Prototype `uint32_t bp_int_prio_next_get (uint32_t prio);`

Attributes

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

Parameters `prio` Interrupt priority.

Returned Values Numerical value of the next interrupt priority.

Function

bp_int_prio_prev_get()

<int/bp_int.h>

Returns the numerical value of the previous interrupt priority level lower than `prio`.

In case the previous lowest priority level is lower than the minimum possible, the minimum interrupt priority level will be returned.

Prototype `uint32_t bp_int_prio_prev_get (uint32_t prio);`

Attributes

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

Parameters	`prio`	Interrupt priority.

Returned Values Numerical value of the previous interrupt priority.

Function

bp_int_prio_set()

<int/bp_int.h>

Sets the priority of an interrupt source. The interrupt id's priority will be set to `priority`. Attempting to configure an invalid priority level for the current interrupt controller will return an `RTNC_FATAL` error.

The range, meaning and order of interrupt priorities is implementation defined and usually follows the platform's convention.

It is implementation specific whether changing the priority of a pending interrupt will be effective immediately.

Prototype

```
int  bp_int_prio_set  ( int        id,
                        uint32_t  priority );
```

Attributes

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

Parameters

`id`	Interrupt id to set.
`priority`	Interrupt priority value.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_int_reg()

<int/bp_int.h>

Registers an interrupt service routine. Sets the ISR handler of the interrupt source `id` to the function `handler`. The optional argument `p_arg` will be passed to the interrupt handler when invoked. See the `bp_int_handler_t` documentation for details.

Setting a NULL `handler` will effectively unregister any ISR registered to that interrupt id. It is the caller's responsibility to make sure the interrupt source is disabled prior to unregistering an ISR.

The result of an interrupt firing without a registered handler is implementation specific. See the implementation's documentation for details.

Prototype

```
int bp_int_reg ( int                id,
                 bp_int_handler_t   handler,
                 void *             p_arg );
```

Attributes

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

Parameters

`id`	Interrupt id to register.
`handler`	Function pointer to the interrupt handler.
`p_arg`	Argument passed to the interrupt handler.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

`bp_int_src_dis()`

<int/bp_int.h>

Disables an interrupt source.

It is implementation specific whether disabling a pending interrupt before it is executed will cancel the pending interrupt.

Prototype

```
int bp_int_src_dis ( int id );
```

Attributes

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

Parameters

`id`	Interrupt id to disable.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

`bp_int_src_en()`

<int/bp_int.h>

Enables an interrupt source. The interrupt source `id` will be enabled even if no ISR is registered for that interrupt id. It is the caller's responsibility to make sure that an ISR is registered to that particular interrupt id before enabling the interrupt. See `bp_int_reg()` for details.

Prototype

```
int bp_int_src_en ( int id );
```

Attributes

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

Parameters `id` Interrupt id to enable.

Returned Errors
RTNC_SUCCESS
RTNC_FATAL

Function

bp_int_src_is_en()

<int/bp_int.h>

Checks if an interrupt source is enabled. Returns the enabled status of the interrupt through `p_is_en`.

Prototype

```
int bp_int_src_is_en ( int    id,
                       bool * p_is_en );
```

Attributes

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

Parameters

`id`	Interrupt id to query.
`p_is_en`	Pointer to the result, set to `true` if the interrupt is enabled, `false` otherwise.

Returned Errors
RTNC_SUCCESS
RTNC_FATAL

Function

bp_int_trig()

<int/bp_int.h>

Triggers a software interrupt.

It is implementation defined whether or not an interrupt can be triggered in software.

Prototype

```
int bp_int_trig ( int id );
```

Attributes

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

Parameters `id` Interrupt id to trigger.

Returned Errors RTNC_SUCCESS
RTNC_FATAL

Function

bp_int_type_get()

<int/bp_int.h>

Gets the trigger type of an interrupt source. The trigger type will be returned through p_type.

Prototype

```
int bp_int_type_get ( int              id,
                      bp_int_type_t *  p_type );
```

Attributes

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

Parameters

id	Interrupt id to query.
p_type	Pointer to the returned interrupt type.

Returned Errors RTNC_SUCCESS
RTNC_FATAL

Function

bp_int_type_set()

<int/bp_int.h>

Sets the trigger type of an interrupt source.

Not all trigger types may be supported on an interrupt controller. It is implementation dependent whether or not an RTNC_NOT_SUPPORTED error is returned when attempting to set an unsupported trigger type. Implementations are free to set a different trigger type when appropriate. Calling bp_int_type_get() will return the actual type when known.

Implementations that do not support changing the interrupt trigger type at runtime will usually ignore the configuration and return successfully.

Prototype

```
int bp_int_type_set ( int             id,
                      bp_int_type_t   type );
```

Attributes

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

Parameters

id	Interrupt id to configure.
type	Interrupt type.

Returned Errors RTNC_SUCCESS
RTNC_FATAL

Data Type

bp_int_type_t

<int/bp_int.h>

Interrupt type used to set both the sensitivity type, either edge or level and polarity. Not all values may be supported by a specific interrupt controller.

See bp_int_type_set() and bp_int_type_get() for details.

Values

BP_INT_TYPE_LEVEL_HIGH	High-level sensitivity.
BP_INT_TYPE_LEVEL_LOW	Low-level sensitivity.
BP_INT_TYPE_EDGE_RISING	Rising edge sensitivity.
BP_INT_TYPE_EDGE_FALLING	Falling edge sensitivity.
BP_INT_TYPE_EDGE_ANY	Any edge or toggle type interrupt sensitivity.

Data Type

bp_int_handler_t

<int/bp_int.h>

Interrupt handler function signature type.

The argument p_int_arg is taken from the p_arg argument used when registering an interrupt handler with bp_int_reg().

The interrupt id int_id is passed to the interrupt handler if know.

The source argument is the id of the CPU core that triggered the interrupt on SMP platforms otherwise it is set to 0.

Prototype

```
void bp_int_handler_t ( void *    p_int_arg,
                        int       int_id,
                        uint32_t  source );
```

Attributes

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

Parameters

p_int_arg	User-defined interrupt argument.
int_id	Interrupt id of the current interrupt if known.
source	Core id of the signaling core for inter-core interrupts.

Macro

BP_INT_ID_NONE

<int/bp_int.h>

Special invalid interrupt value.

Chapter

10

Interrupt SMP Extension

SMP extension of the interrupt management API. The SMP extensions are used to fine-tune interrupt behaviour on SMP platforms. Note that the SMP extension API will work in an AMP configuration on an SMP platform as well to control interrupt targeting and triggering between cores.

Function

bp_int_smp_src_dis()

<int/bp_int_smp.h>

Disables an interrupt source on a specific core.

It is implementation specific whether disabling a pending interrupt before it is executed will cancel the pending interrupt.

Prototype

```
int  bp_int_smp_src_dis  ( int        id,
                           uint32_t  core_id );
```

Attributes

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

Parameters

`id`	Interrupt id to disable.
`core_id`	ID of the core to target.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_int_smp_src_en()

<int/bp_int_smp.h>

Enables an interrupt source on a specific core. The interrupt source `id` will be enabled even if no ISR is registered for that interrupt id. It is the caller's responsibility to make sure that an ISR is registered to that particular interrupt id before enabling the interrupt. See `bp_int_reg()` for details.

Prototype

```
int  bp_int_smp_src_en  ( int       id,
                          uint32_t  core_id );
```

Attributes

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

Parameters

`id`	Interrupt id to enable.
`core_id`	ID of the core to target.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_int_smp_trig()

<int/bp_int_smp.h>

Triggers a software interrupt targeting a specific core.

It is implementation defined whether or not an interrupt can be triggered in software. It is also implementation defined which interrupts can be targeted to a specific core. In case an interrupt can be triggered by software but cannot be targeted to a specific core the behaviour will be the same as if `bp_int_trig()` was called.

For maximum portability, `bp_int_trig()` should be used to trigger a peripheral interrupt.

Prototype

```
int  bp_int_smp_trig  ( int       id,
                        uint32_t  core_id );
```

Attributes

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

Parameters

`id`	Interrupt id to trigger.
`core_id`	ID of the core to target.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Chapter

11 GPIO

The GPIO module allows control over a platform's General Purpose I/Os. It can also be used to access various types of external I/O expanders.

In contrast to the majority of the BASEplatform peripheral interface modules, the GPIO module API is non-blocking since driver implementations are usually atomic by design. Most of the GPIO module API can be called from a critical or interrupt context. However, as a general exception, drivers for external I/O expanders can be blocking, especially if accessing an I2C or SPI expander.

The meaning of the bank and pin numbers are platform specific, and usually follows the MCU or SoC's numbering as documented in the manufacturer's manuals. Additional details about each GPIO implementation can be found by consulting the individual driver's documentation.

Function

`bp_gpio_create()`

<gpio/bp_gpio.h>

Creates a new GPIO module instance. The created GPIO instance is associated with the GPIO peripheral definition `p_def`. If successful, a handle to the newly created instance is returned through the `p_hndl` argument. After returning from a successful call to `bp_gpio_create()` the newly created instance is in the created state and should subsequently be enabled to be fully functional. See `bp_gpio_en()` for details.

The GPIO definition structure `p_def` must be unique and can only be associated with a single GPIO instance. Once created, the UART instance is assigned a name that can be used afterward to retrieve the interface handle by calling `bp_gpio_hndl_get()`. The assigned name is set from the board definition structure `p_def` and must be unique.

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

The board definition p_def passed to bp_gpio_create() must be kept valid for the lifetime of the application once the GPIO interface is open.

When bp_gpio_create() returns with either an RTNC_NO_RESOURCE or RTNC_ALREADY_EXIST error, the destination of p_hndl is left unmodified.

Unless specified otherwise in the driver documentation, opening, enabling or disabling a GPIO interface will not alter or clear the direction and pin state of the GPIO interface.

Prototype

```
int  bp_gpio_create  ( const bp_gpio_board_def_t *  p_def,
                       bp_gpio_hndl_t *             p_hndl );
```

Attributes

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

Parameters

p_def	Board definition of the GPIO peripheral to initialize.
p_hndl	Handle to the created GPIO module instance.

Returned Errors

RTNC_SUCCESS
RTNC_ALREADY_EXIST
RTNC_NO_RESOURCE
RTNC_FATAL

Example

```
extern bp_gpio_board_def_t g_gpio0;
bp_gpio_hndl_t gpio_hndl

bp_gpio_create(\&g_gpio0, \&gpio_hndl);
```

Function

bp_gpio_data_get()

<gpio/bp_gpio.h>

Gets the state of a GPIO pin. Returns the data state of pin number pin of bank bank through the argument p_data. p_data will be set to either 0 or 1.

Prototype

```
int  bp_gpio_data_get  ( bp_gpio_hndl_t  hndl,
                         uint32_t        bank,
                         uint32_t        pin,
                         uint32_t *      p_data );
```

Attributes

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

Parameters

`hndl`	Handle of the GPIO interface to query.
`bank`	Bank number of the pin to query.
`pin`	Pin number of the pin to query.
`p_data`	Pointer to the returned data state.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_gpio_data_set()

<gpio/bp_gpio.h>

Sets the state of a GPIO pin. Set the state of pin number `pin` of bank `bank` to the data specified by `data`. Data should be either 0 or 1.

Prototype

```
int  bp_gpio_data_set  ( bp_gpio_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 GPIO 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`

Function

bp_gpio_data_tog()

<gpio/bp_gpio.h>

Toggles the state of a GPIO pin. Toggle the the data value from low to high or from high to low of pin number `pin` of bank `bank`.

Prototype

```
int  bp_gpio_data_tog  ( bp_gpio_hndl_t  hndl,
                         uint32_t        bank,
                         uint32_t        pin );
```

Attributes

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

Parameters

`hndl`	Handle of the GPIO 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`

Function

bp_gpio_destroy()

<gpio/bp_gpio.h>

Destroys a GPIO module instance. When supported, `bp_gpio_destroy()` will free up all the resources allocated to the GPIO module instance, including the peripheral driver and internal data structures. Depending on the memory allocation policy of the default memory allocator, it may not be possible to free previously allocated memory, in that case `RTNC_NOT_SUPPORTED` is returned and the GPIO module instance is left unaffected.

It is not necessary, but strongly recommended, to disable a GPIO instance by calling `bp_gpio_dis()` before attempting to destroy it. This helps ensure that no race condition exists between the instance destruction and ongoing operations.

The result of using a GPIO module handle after its underlying instance is destroyed is undefined.

Prototype

```
int bp_gpio_destroy (bp_gpio_hndl_t hndl);
```

Attributes

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

Parameters

`hndl`	Handle of the GPIO module instance to destroy.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_SUPPORTED`
`RTNC_FATAL`

Function

bp_gpio_dir_get()

<gpio/bp_gpio.h>

Gets the direction of a GPIO pin. Returns the direction of pin number `pin` of bank `bank` through the argument `p_dir`.

Prototype

```
int  bp_gpio_dir_get  ( bp_gpio_hndl_t   hndl,
                        uint32_t         bank,
                        uint32_t         pin,
                        bp_gpio_dir_t *  p_dir );
```

Attributes

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

Parameters

`hndl`	Handle of the GPIO interface to query.
`bank`	Bank number of the pin to query.
`pin`	Pin number of the pin to query.
`p_dir`	Pointer to the returned direction.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

`bp_gpio_dir_set()`

<gpio/bp_gpio.h>

Sets the direction of a GPIO pin. Sets the direction of pin number `pin` of bank `bank` to the direction specified by `dir`.

Prototype

```
int  bp_gpio_dir_set  ( bp_gpio_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 GPIO 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_gpio_dis()

<gpio/bp_gpio.h>

Disables a GPIO interface. The exact side effects of disabling an interface is driver dependent. In general, the peripheral is disabled at the peripheral level, and, when possible, the module clock is gated.

The result of calling `bp_gpio_dis()` or any other functions other than `bp_gpio_en()` or `bp_gpio_reset()` on an already disabled interface is undefined. With assertion checking enabled, some drivers will return an `RTNC_FATAL` error when attempting to access a disabled interface. The current enabled/disabled state of an interface can be queried using `bp_gpio_is_en()`.

Unless specified otherwise in the driver documentation, opening, enabling or disabling a GPIO interface will not alter or clear the direction and pin state of the GPIO interface.

To optimize performance and footprint, GPIO drivers are allowed to ignore the calls to `bp_gpio_en()` and `bp_gpio_dis()` and be in the enabled state permanently after being opened. For compatibility with future releases and portability between GPIO drivers `bp_gpio_en()` should be called before attempting to use a newly opened GPIO interface.

Prototype

```
int bp_gpio_dis ( bp_gpio_hndl_t hndl );
```

Attributes

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

Parameters

`hndl` Handle of the GPIO module instance to disable.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_gpio_drv_hndl_get()

<gpio/bp_gpio.h>

Returns the driver handle associated with a GPIO module instance. The underlying driver handle will be returned through `p_drv_hndl`. The driver handle can be used to perform operations with the driver interface directly or to access driver specific features.

Prototype

```
int bp_gpio_drv_hndl_get ( bp_gpio_hndl_t      hndl,
                           bp_gpio_drv_hndl_t *  p_drv_hndl );
```

Attributes

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

Parameters	`hndl`	Handle of the GPIO module instance to query.
	`p_drv_hndl`	Pointer to the GPIO driver handle.

Returned Errors `RTNC_SUCCESS`
`RTNC_FATAL`

Function

`bp_gpio_en()`

<gpio/bp_gpio.h>

Enables a GPIO interface. Enabling an interface in the disabled state will, depending on the driver, enable the peripheral clock, de-assert reset, if asserted, and enable modifications of the GPIO states.

Calling `bp_gpio_en()` on an enabled interface should be without side effect.

Unless specified otherwise in the driver documentation, opening, enabling or disabling a GPIO interface will not alter or clear the direction and pin state of the GPIO interface.

To optimize performance and footprint, GPIO drivers are allowed to ignore the calls to `bp_gpio_en()` and `bp_gpio_dis()` and be in the enabled state permanently after being opened. For compatibility with future releases and ensure portability between GPIO drivers, `bp_gpio_en()` should be called before attempting to use a newly opened GPIO module instance.

Prototype

```
int bp_gpio_en (bp_gpio_hndl_t hndl);
```

Attributes

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

Parameters	`hndl`	Handle of the GPIO module instance to enable.

Returned Errors `RTNC_SUCCESS`
`RTNC_FATAL`

Function

`bp_gpio_hndl_get()`

<gpio/bp_gpio.h>

Retrieves a previously created GPIO instance handle by name. If found, the result is returned through the `p_hndl` argument, otherwise `RTNC_NOT_FOUND` is returned and `p_hndl` is left as it was before the call to `bp_gpio_hndl_get()`.

The name of an instance is set in the `bp_gpio_board_def_t` board definition passed to `bp_gpio_create()`.

Prototype

```
int bp_gpio_hndl_get ( const char *      p_name,
                       bp_gpio_hndl_t *  p_hndl );
```

Attributes

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

Parameters

`p_name`	Name of the GPIO instance to retrieve.
`p_hndl`	Pointer to the GPIO interface handle.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_FATAL`

Function

bp_gpio_is_en()

<gpio/bp_gpio.h>

Returns the enabled/disabled state of a GPIO interface. If successful, the state of the GPIO interface `hndl` will be returned through argument `p_is_en`.

The state of an interface is checked atomically in a non-blocking way. As such, `bp_gpio_is_en()` can be called while another operation is in progress without blocking or from an interrupt service routine.

Prototype

```
int bp_gpio_is_en ( bp_gpio_hndl_t  hndl,
                    bool *          p_is_en );
```

Attributes

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

Parameters

`hndl`	Handle of the GPIO interface to check.
`p_is_en`	Returned interface state, `true` if enabled `false` otherwise.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_gpio_reset()

<gpio/bp_gpio.h>

Resets a GPIO module instance. Upon a successful call to `bp_gpio_reset()` the GPIO interface is left in the created state, equivalent to the state a newly created instance. Before using the instance again it must be re-enabled, see `bp_gpio_en()`.

Pin states are likely to be lost after a reset, reset a platform's GPIO peripheral should be done with care.

Prototype `int bp_gpio_reset (bp_gpio_hndl_t hndl);`

Attributes

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

Parameters `hndl` Handle of the GPIO interface to reset.

Returned Errors `RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

bp_gpio_dir_t

<gpio/bp_gpio.h>

GPIO direction. Enumeration of the possible GPIO direction values used by the GPIO module and drivers.

See `bp_gpio_dir_set()` and `bp_gpio_dir_get()` for usage details.

Values

BP_GPIO_DIR_NONE	Special NULL value.
BP_GPIO_DIR_IN	GPIO pin configured as input.
BP_GPIO_DIR_OUT	GPIO pin configured as output.

Data Type

bp_gpio_board_def_t

<gpio/bp_gpio.h>

GPIO board level hardware definition. Complete definition of a GPIO interface, including the name, BSP as well as the SoC level definition structure of type `bp_gpio_soc_def_t` providing the driver and driver specific parameters. The overall definition of a GPIO interface should be unique, including the name, for each GPIO module instance to prevent conflicts.

BSP definitions are driver specific an usually not required, when that is the case `p_bsp_def` should be set to NULL. See the driver's documentation for details.

See `bp_gpio_create()` for usage details.

Members

p_soc_def	const bp_gpio_soc_def_t *	SoC level hardware definition.

p_bsp_def	const void *	Board and application-specific definition.
p_name	const char *	GPIO instance name.

Data Type

bp_gpio_drv_hndl_t

<gpio/bp_gpio.h>

GPIO driver handle. GPIO driver handle returned by a driver's create function. The pointer contained in the handle is private and should not be accessed by calling code. See bp_gpio_drv_create_t for a generic description of a driver's create function.

Most GPIO drivers are single instance drivers that handles all the GPIOs of a chip with a single driver instance to save resources. Those driver can be passed a BP_GPIO_DRV_NULL_HNDL to use the default instance.

Members

p_hndl	void *	Private pointer to the driver instance.

Data Type

bp_gpio_hndl_t

<gpio/bp_gpio.h>

GPIO handle. GPIO handle returned by bp_gpio_create() and used for subsequent access to a GPIO module instance. The pointer contained in the handle is private and should not be accessed by calling code.

See bp_gpio_create() for usage details.

Members

p_hndl	bp_gpio_inst_t *	Private pointer to the GPIO module instance internal data.

Data Type

bp_gpio_soc_def_t

<gpio/bp_gpio.h>

GPIO module SoC level hardware definition structure.

The GPIO hardware definition structure is used to describe the peripheral at the SoC level. The structure specifies the driver to be used as well as a driver specific definition structure usually specifying the location, clock, interrupt and various other parameters required by each GPIO drivers.

To be complete, a GPIO hardware instance also requires a board specific portion. Both this structure and the BSP structures are referenced by a `bp_gpio_board_def_t` structure to describe a form a complete GPIO interface definition.

Members

`p_drv`	`const bp_gpio_drv_t *`	Driver associated with this peripheral.
`p_drv_def`	`const void *`	Driver specific hardware definition.

Macro

BP_GPIO_HNDL_IS_NULL()

<gpio/bp_gpio.h>

Evaluates if a GPIO handle is NULL.

Prototype `BP_GPIO_HNDL_IS_NULL ( hndl );`

Parameters `hndl` Handle to be checked.

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

Macro

BP_GPIO_NULL_HNDL

<gpio/bp_gpio.h>

NULL GPIO module handle.

Chapter

12

I2C

The I2C module allows access to Inter-Integrated Circuit (I2C) compatible peripherals in both master and slave configurations.

I2C drivers are usually written to minimize the number of interrupts and context switches generated by I2C operations.

Considering the wide varieties of I2C compatible peripherals, it would be impossible to design a high-level API that could leverage the unique features of many peripherals. To alleviate this, drivers are allowed to implement driver-specific functionalities to extend the features of the I2C module. Details of these features can be found in each driver's documentation.

In addition to directly accessing an external i2c peripherals, the BASEplatform also includes many boards component modules and drivers for popular parts such as IO expanders, EEPROMs, sensors and more.

Function

bp_i2c_acquire()

<i2c/bp_i2c.h>

Acquires exclusive access to an I2C interface. Upon a successful call the I2C module instance will be accessible exclusively from the current thread.

`bp_i2c_acquire()` has no effect in a bare-metal environment.

Prototype

```
int  bp_i2c_acquire  ( bp_i2c_hndl_t  hndl,
                       uint32_t       timeout_ms );
```

Attributes

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

Parameters	`hndl`	Handle of the I2C module instance to acquire.
	`timeout_ms`	Timeout in milliseconds.

Returned Errors
`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

bp_i2c_addr_is_10b()

<i2c/bp_i2c.h>

Checks if an I2C address is in the 10-bit I2C address range. By the standard a valid 10-bit I2C address ranges from 0x78 (120 decimal) to 0x3FB (1019 decimal) inclusively.

Prototype

```
bool bp_i2c_addr_is_10b (uint32_t addr);
```

Attributes

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

Parameters `addr` Address to validate.

Returned Values Returns `true` if the address is a 10-bit address `false` otherwise.

Function

bp_i2c_addr_is_valid()

<i2c/bp_i2c.h>

Checks the validity of an I2C slave address. Validates that the I2C address `addr` is valid according to the I2C specifications.

Prototype

```
bool bp_i2c_addr_is_valid (uint32_t addr);
```

Attributes

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

Parameters `addr` Address to validate.

Returned Values Returns `true` if the address is valid `false` otherwise.

Function

bp_i2c_cfg_get()

<i2c/bp_i2c.h>

Retrieves the current configuration of an I2C interface. Returns the configuration of the I2C interface through `p_cfg`. The configuration returned is derived from the hardware registers and reflects the actual configuration regardless of the last configuration set by `bp_i2c_cfg_set()`.

The clock frequency returned is the actual frequency when known, otherwise the `clk_freq` member of the `p_cfg` argument is set to 0.

It is driver specific whether the slave address specified in the `p_cfg` configuration structure is saved or set when the `master` field is true. This means that some drivers will return a slave address of 0 when calling `bp_i2c_cfg_get()` when configured as a master. For compatibility application code should not rely on `bp_i2c_cfg_get()` returning a valid i2c address when configured as a master.

When `bp_i2c_cfg_get()` returns with an `RTNC_TIMEOUT` error, the destination of `p_cfg` is left unmodified.

Prototype

```
int bp_i2c_cfg_get ( bp_i2c_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 module instance to query.
`p_cfg`	Pointer to the returned I2C configuration.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

bp_i2c_cfg_set()

<i2c/bp_i2c.h>

Configures an I2C interface. Configures the I2C interface using configuration `p_cfg`. If the interface was in the opened state, it will transition to the configured state. Otherwise the interface configuration is updated.

The underlying driver will attempt to configure the closest clock frequency to the specified frequency. Calling `bp_i2c_cfg_get()` will return the actual frequency configured.

It is driver specific whether the slave address specified in the `p_cfg` configuration structure is saved or set when the `master` field is true. This means that some drivers will return a slave address of 0 when calling bp_i2c_cfg_get() when configured as a master. For compatibility application code should not rely on bp_i2c_cfg_get() returning a valid i2c address when configured as a master.

When bp_i2c_cfg_set() returns with a RTNC_NOT_SUPPORTED or RTNC_TIMEOUT error, it is guaranteed that the current configuration is unaffected.

Not all peripherals support both master and slave modes. Attempting to set an unsupported mode will return RTNC_NOT_SUPPORTED.

Drivers for peripherals that do not support changing the clock speed will ignore the bit_rate argument. bp_i2c_cfg_get() will return the fixed speed if known.

It is driver specific whether or not an RTNC_NOT_SUPPORTED error is returned on configurations not supported by the underlying peripheral. Unless specified differently by the driver documentation, the following holds true.

- A clock speed of 0 will return an RTNC_FATAL error, unless it has a special meaning for the hardware.
- Specifying a clock speed outside of the peripheral's supported range will configure the closest supported rate.
- Specifying an unsupported mode will return RTNC_NOT_SUPPORTED.
- Drivers for peripherals with a fixed hardware configuration such as soft IPs for FPGAs, will usually ignore any configuration parameters and return successfully.

Prototype

```
int  bp_i2c_cfg_set  ( bp_i2c_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 module instance to configure.
p_cfg	I2C configuration to apply.
timeout_ms	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_NOT_SUPPORTED
RTNC_FATAL

Example

```
bp_i2c_hndl_t i2c_hndl;
bp_i2c_cfg_t i2c_cfg;

i2c_cfg.bit_rate = 400000u;
i2c_cfg.master = true;

bp_i2c_cfg_set(i2c_hndl, \&i2c_cfg, TIMEOUT_INF);
```

Function

`bp_i2c_create()`

<i2c/bp_i2c.h>

Creates an I2C module instance. The created I2C instance is associated with the I2C peripheral definition `p_def`. If successful, a handle to the newly created instance is returned through the `p_hndl` argument. After returning from a successful call to `bp_i2c_create()` the newly created instance is in the created state and should subsequently be configured and enabled to be fully functional. See `bp_i2c_cfg_set()` and `bp_i2c_en()` for details.

The I2C definition structure `p_def` must be unique and can only be associated with a single I2C instance. Once created, the I2C instance is assigned a name that can be used afterward to retrieve the interface handle by calling `bp_i2c_hndl_get()`. The assigned name is set from the board definition structure `p_def` and must be unique.

An I2C peripheral cannot be opened more than once. If an attempt is made to open the same interface twice, `bp_i2c_create()` returns an `RTNC_ALREADY_EXIST` error without affecting the already opened interface.

The board definition `p_def` passed to `bp_i2c_create()` must be kept valid for the lifetime of the I2C module instance.

When `bp_i2c_create()` returns with either an `RTNC_NO_RESOURCE` or `RTNC_ALREADY_EXIST` error, the destination of `p_hndl` is left unmodified.

Prototype

```
int bp_i2c_create ( const bp_i2c_board_def_t *  p_def,
                    bp_i2c_hndl_t *             p_hndl );
```

Attributes

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

Parameters

`p_def`	Definition of the I2C peripheral to initialize.
`p_hndl`	Pointer to the newly created I2C module instance.

Returned Errors

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

Example

```
extern bp_i2c_board_def_t g_i2c0;
bp_i2c_hndl_t i2c_hndl;

bp_i2c_create(\&g_i2c0, \&i2c_hndl);
```

Function

bp_i2c_destroy()

<i2c/bp_i2c.h>

Destroys an I2C module instance. When supported, `bp_i2c_destroy()` will free up all the resources allocated to the I2C module instance, including the peripheral driver and internal data structures. Depending on the memory allocation policy of the default memory allocator it may not be possible to free previously allocated memory, in that case `RTNC_NOT_SUPPORTED` is returned and the I2C module instance is left unaffected.

It is not necessary, but strongly recommended, to disable an I2C interface by calling `bp_i2c_dis()` before attempting to destroy it. This helps ensure that no race condition exists between the instance destruction and ongoing transfers.

The result of using an I2C module handle after its underlying instance is destroyed is undefined.

Prototype

```
int  bp_i2c_destroy  ( bp_i2c_hndl_t  hndl,
                       uint32_t       timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the I2C instance to destroy.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_NOT_SUPPORTED`
`RTNC_FATAL`

Function

bp_i2c_dis()

<i2c/bp_i2c.h>

Disables an I2C interface. `bp_i2c_dis()` will wait for the interface to be idle before disabling it.

The exact side effect of disabling an interface is driver dependent. In general the peripheral is disabled at the peripheral level, and, when possible, the module clock is gated.

The result of calling `bp_i2c_dis()` or any other functions other than `bp_i2c_en()` or `bp_i2c_reset()` on an already disabled interface is undefined. With assertion checking enabled, some drivers will return `RTNC_FATAL` when attempting to access a disabled interface. The current enabled/disabled state of an interface can be queried using `bp_i2c_is_en()`.

Prototype

```
int bp_i2c_dis ( bp_i2c_hndl_t hndl,
                 uint32_t      timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the I2C module instance to disable.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

bp_i2c_drv_hndl_get()

<i2c/bp_i2c.h>

Returns the driver handle associated with an I2C module instance. The underlying driver handle will be returned through `p_drv_hndl`. The driver handle can be used to perform operations with the driver interface directly or to access driver specific features.

Prototype

```
int bp_i2c_drv_hndl_get ( bp_i2c_hndl_t       hndl,
                          bp_i2c_drv_hndl_t * p_drv_hndl );
```

Attributes

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

Parameters

`hndl`	Handle of the I2C module instance to query.
`p_drv_hndl`	Pointer to the received I2C driver handle.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_i2c_en()

<i2c/bp_i2c.h>

Enables an I2C interface. Enabling an interface in the disabled or configured state will, depending on the driver, enable the peripheral clock, de-assert reset, if asserted, and enable transmission and reception through the I2C interface.

Calling `bp_i2c_en()` on an enabled interface should be without side effect.

Prototype

```
int bp_i2c_en ( bp_i2c_hndl_t hndl,
                uint32_t      timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the I2C module instance to enable.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

bp_i2c_flush()

<i2c/bp_i2c.h>

Flushes the transmit and receive paths. Flush the transmit and receive paths of an I2C interface. It is unspecified whether any data written but not yes transmitted is sent or dropped.

Prototype

```
int bp_i2c_flush ( bp_i2c_hndl_t hndl,
                   uint32_t      timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the I2C module instance to flush.
`timeout_ms`	Timeout in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

bp_i2c_hndl_get()

<i2c/bp_i2c.h>

Retrieves a previously created I2C instance handle by name. If found, the result is returned through the p_hndl argument, otherwise RTNC_NOT_FOUND is returned and p_hndl is left as it was before the call to bp_i2c_hndl_get().

The name of an interface is set in the bp_i2c_board_def_t board description passed to bp_i2c_create().

Prototype

```
int bp_i2c_hndl_get ( const char *      p_name,
                      bp_i2c_hndl_t *   p_hndl );
```

Attributes

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

Parameters

p_name	Name of the I2C instance to retrieve.
p_hndl	Pointer to the returned I2C interface handle.

Returned Errors

RTNC_SUCCESS
RTNC_NOT_FOUND
RTNC_FATAL

Function

bp_i2c_idle_wait()

<i2c/bp_i2c.h>

Waits for an I2C interface to be idle.

Prototype

```
int bp_i2c_idle_wait ( bp_i2c_hndl_t  hndl,
                       uint32_t       timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the I2C module instance to wait on.
timeout_ms	Timeout in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

`bp_i2c_is_en()`

<i2c/bp_i2c.h>

Returns the enabled/disabled state of an I2C interface. If the call is successful, the state of the I2C interface `hndl` through argument `p_is_en`.

The state of an interface is checked atomically in a non-blocking way. As such `bp_i2c_is_en()` can be called while another operation is in progress without blocking or from an interrupt service routine.

Prototype

```
int bp_i2c_is_en ( bp_i2c_hndl_t hndl,
                   bool *         p_is_en );
```

Attributes

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

Parameters

`hndl`	Handle of the I2C module instance to query.
`p_is_en`	Interface state, `true` if enabled `false` otherwise.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

`bp_i2c_release()`

<i2c/bp_i2c.h>

Releases exclusive access to an I2C interface.

`bp_i2c_release()` has no effect in a bare-metal environment.

Prototype

```
int bp_i2c_release ( bp_i2c_hndl_t hndl );
```

Attributes

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

Parameters

`hndl`	Handle of the I2C module instance to release.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

bp_i2c_reset()

<i2c/bp_i2c.h>

Resets an I2C module instance. Upon a successful call to bp_i2c_reset() the I2C interface is returned to the created state. Before using the interface again it must be configured and enabled, see bp_i2c_cfg_set() and bp_i2c_en().

Any asynchronous transfers in progress will be aborted without calling their callback functions.

Prototype

```
int bp_i2c_reset ( bp_i2c_hndl_t hndl,
                   uint32_t      timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the I2C interface to reset.
timeout_ms	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_i2c_xfer()

<i2c/bp_i2c.h>

Performs an I2C operation. Transmit or receive through I2C interface according to the p_tf transfer descriptor. See the bp_i2c_tf_t documentation for details of the individual fields.

The callback member of the p_tf argument, which is only used for asynchronous transfers, should be set to NULL.

In slave mode the pointee of argument p_tf_len will be the actual number of bytes received in case of a successful transfer or a receive timeout.

In slave mode RTNC_WANT_READ and RTNC_WANT_WRITE will be returned when the requested operation, either a transmit or a receive, doesn't match the operation requested by the I2C master. In those cases nothing is performed the application should setup a new I2C transfer with the correct direction.

In master mode the hold_nack member of the transfer description structure can be set to true to hold the bus after the master operation, allowing for a repeated start at the next operation. Note that the bus will be held indefinitely if no other master operation is performed with hold_nack set to false. To prevent contention issues in multi-master operation or possible slave timeout it is recommended to minimize the delay between master operations with the bus held.

The timeout value is the amount of time to wait for the channel to be available. The time spent to perform the transfer is not counted to consider a timeout condition. Drivers that support querying the bit rate of the interface in master mode can return RTNC_FATAL in case the transfer operation is taking longer than expected.

Prototype

```
int bp_i2c_xfer ( bp_i2c_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 I2C module instance to use for the transfer.
p_tf	Pointer to an bp_i2c_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_WANT_READ
RTNC_WANT_WRITE
RTNC_IO_ERR
RTNC_FATAL

Example

```
bp_i2c_tf_t tf;
size_t tf_len

tf.p_buf = p_buf;
tf.buf_len = 0u;
tf.dir = BP_I2C_DIR_RX;
tf.slave_addr = 0xA;
tf.hold_nack = false;
tf.callback = NULL;

bp_i2c_xfer(i2c_hndl, \&tf, \&tf_len, TIMEOUT_INF);
```

Function

bp_i2c_xfer_async()

<i2c/bp_i2c.h>

Transfers data asynchronously. Performs an asynchronous transfer operation according to the parameters of the p_tf argument, see the bp_i2c_tf_t documentation for an explanation of the transfer parameters. Upon successfully starting a transfer, the function returns immediately. The callback specified in the p_tf structure will be called when the transfer is finished. If no callback is specified a fire and forget transfer will be performed, where the entire operation will be executed in the background. Care should be taken when using such transfers as it's not possible for the application to know if the transfer succeeded.

The timeout argument timeout_ms specifies the amount of time to wait for the channel to be available. The timeout value has no impact on the asynchronous transfer operation once started.

In master mode the hold_nack member of the transfer description structure can be set to true to hold the bus after the master operation, allowing for a repeated start at the next operation. Note that the bus will be held indefinitely if no other master operation is performed with hold_nack set to false. To prevent contention issues in multi-master operation or possible slave timeout it is recommended to minimize the delay between master operations with the bus held.

When bp_i2c_xfer_async() returns with an RTNC_TIMEOUT, error the transfer is not started and the callback function specified in p_tf won't be called.

The structure referenced by p_tf must be valid for the entire asynchronous transfer operation and may be accessed by the I2C driver. Upon returning, the original state of the transfer descriptor will be preserved. p_tf will be passed verbatim to the callback and may be modified within the user callback to perform an additional transfer from the callback.

The p_ctxt member of the p_tf transfer descriptor can be used to pass user context information to the callback.

Prototype

```
int bp_i2c_xfer_async ( bp_i2c_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 I2C module instance to use for the transfer.
p_tf	Transfer parameters.
timeout_ms	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Example

```
bp_i2c_tf_t tf;


tf.p_buf = p_buf;
tf.buf_len = 0u;
tf.dir = BP_I2C_DIR_RX;
tf.slave_addr = 0xA;
tf.hold_nack = false;
tf.callback = cb_func;

bp_i2c_xfer_async(i2c_hndl, \&tf, TIMEOUT_INF);
```

Function

bp_i2c_xfer_async_abort()

<i2c/bp_i2c.h>

Aborts an asynchronous transfer. Aborts any running asynchronous transfer operation. The number of bytes already transmitted will be returned through `p_tf_len` if it's not NULL.

In case of a successful abort the transfer callback function of the aborted operation won't be called. It is, however, possible for the transfer to finish just before being aborted in which case `bp_i2c_xfer_async_abort()` will return with `RTNC_SUCCESS`.

When aborting a write operation `p_tf_len` may not reflect the actual number of bytes successfully written through the I2C bus.

In case no asynchronous transfer operation is in progress `bp_i2c_xfer_async_abort()` will return `RTNC_SUCCESS` and the number of bytes transmitted will be 0.

Prototype

```
int bp_i2c_xfer_async_abort ( bp_i2c_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 I2C module instance 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_action_t

<i2c/bp_i2c.h>

Asynchronous IO return action. These are the return value possible to an I2C asynchronous IO callback instructing the driver on the action to be performed. See `bp_i2c_xfer_async()` and `bp_i2c_async_cb_t` for usage details.

Values

`BP_I2C_ACTION_FINISH`	Finish normally.
`BP_I2C_ACTION_RESTART`	Restart a transfer with the data of the current transfer description structure.

Data Type

bp_i2c_dir_t

<i2c/bp_i2c.h>

I2C direction.

To be used in the `bp_i2c_tf_t` I2C operation structure. See `bp_i2c_xfer()` and `bp_i2c_xfer_async()` for details.

Values

`BP_I2C_DIR_TX`	I2C transmit/output.
`BP_I2C_DIR_RX`	I2C receive/input.

Data Type

bp_i2c_async_cb_t

<i2c/bp_i2c.h>

Asynchronous IO callback. Callback function pointer type to be used with non-blocking asynchronous transfers.

When an asynchronous transfer is finished, the callback will be called if set. The `status` argument will be one of the following, indicating the result of the transfer:

- `RTNC_SUCCESS` The transfer finished normally.
- `RTNC_IO_ERR` An I/O error occurred.
- `RTNC_WANT_READ` Slave write requested but master indicated a read.
- `RTNC_WANT_WRITE` Slave read requested but master indicated a write.
- `RTNC_FATAL` A fatal error was detected.

Two actions are possible when returning.

- `BP_I2C_ACTION_FINISH` Finish the transfer normally.

- BP_I2C_ACTION_RESTART Restart the transfer operation with the data in the p_tf transfer description structure.

The transfer description structure is the same that was passed to the initial call to bp_i2c_xfer_async(). It can be modified prior to returning BP_SPI_ACTION_RESTART to restart a transfer immediately from the callback using the updated transfer descriptor.

See bp_i2c_xfer_async() for usage details.

Prototype

```
bp_i2c_action_t  bp_i2c_async_cb_t  ( int          status,
                                      size_t       tf_len,
                                      bp_i2c_tf_t * p_tf );
```

Attributes

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

Parameters

status	Status of the asynchronous operation.
tf_len	Amount of bytes actually transferred in case of timeout or error.
p_tf	Pointer to the current transfer.

Returned Values

Return value of type bp_i2c_action_t to signal the desired operation (terminate or restart).

Data Type

bp_i2c_board_def_t

<i2c/bp_i2c.h>

I2C board-level hardware definition. Complete definition of an I2C interface, including the name, BSP as well as the SoC level definition structure of type bp_i2c_soc_def_t providing the driver and driver specific parameters. The overall definition of an I2C interface should be unique, including the name, for each I2C module instance to prevent conflicts.

BSP definitions are driver specific and usually not required, when that is the case p_bsp_def should be set to NULL. See the driver's documentation for details.

See bp_i2c_create() for usage details.

Members

p_soc_def	const bp_i2c_soc_def_t *	SoC-level hardware definition.
p_bsp_def	void *	Board and application specific definition.
p_name	const char *	I2C peripheral name.

Data Type

bp_i2c_cfg_t

<i2c/bp_i2c.h>

I2C configuration structure. Used to set or return the configuration of an I2C interface.

See bp_i2c_cfg_set() and bp_i2c_cfg_get() for usage details.

Members

bit_rate	uint32_t	Bit rate.
slave_addr	uint16_t	Slave address, ignored for master configuration.
master	bool	true for master mode false for slave.

Data Type

bp_i2c_drv_hndl_t

<i2c/bp_i2c.h>

I2C driver data handle. Pointer to driver private data. The pointer contained in the handle is private and should not be accessed by calling code.

See bp_i2c_driver_create_t and the driver documentation for details.

Members

p_hndl	void *	Pointer to the internal I2C driver's data.

Data Type

bp_i2c_hndl_t

<i2c/bp_i2c.h>

I2C handle. I2C handle returned by bp_i2c_create(). The pointer contained in the handle is private and should not be accessed by calling code.

Members

p_hndl	bp_i2c_inst_t *	Pointer to the I2C module internal instance data.

Data Type

bp_i2c_soc_def_t

<i2c/bp_i2c.h>

I2C module SoC-level hardware definition structure.

The I2C hardware definition structure is used to describe the peripheral at the SoC level. The structure specifies the driver to be used as well as driver specific definition structure usually specifying the location, clock, interrupt and various other parameters required by each I2C drivers.

To be complete, an I2C hardware instance also requires a board specific portion. Both this structure and the BSP structures are referenced by a `bp_i2c_board_def_t` structure to describe a form a complete I2C interface definition.

Members

`p_drv`	`const bp_i2c_drv_t *`	Driver associated with this peripheral.
`p_drv_def`	`const void *`	Driver specific definition structure.

Data Type

bp_i2c_tf_t

<i2c/bp_i2c.h>

I2C operation definition structure. Used to describe an I2C operation to perform. See `bp_i2c_xfer()` and `bp_i2c_xfer_async()` for usage details.

Members

`dir`	`bp_i2c_dir_t`	Direction.
`hold_nack`	`bool`	Set to true to hold the bus in master mode or to nack after the end of a transfer in slave mode.
`p_buf`	`void *`	Point to data buffer to transmit or receive.
`slave_addr`	`uint16_t`	Slave address.
`buf_len`	`uint32_t`	Length of data to transmit or receive in bytes.
`callback`	`bp_i2c_async_cb_t`	Async transfer callback. Should be set to NULL for non-async transfers.
`p_ctxt`	`void *`	Optional user context pointer passed to the asynchronous callback.

Macro

BP_I2C_10B_SLV_ADDR_MASK

<i2c/bp_i2c.h>

10-bit I2C address mask.

Macro

BP_I2C_HNDL_IS_NULL()

<i2c/bp_i2c.h>

Evaluates if an I2C module handle is NULL.

Prototype `BP_I2C_HNDL_IS_NULL ( hndl );`

Parameters `hndl` Handle to be checked.

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

Macro

BP_I2C_MAX_10B_SLV_ADDR

<i2c/bp_i2c.h>

Highest 10-bit I2C address.

Macro

BP_I2C_MAX_SLV_ADDR

<i2c/bp_i2c.h>

Highest 7-bit I2C address.

Macro

BP_I2C_MIN_10B_SLV_ADDR

<i2c/bp_i2c.h>

Lowest 10-bit I2C address.

Macro

BP_I2C_NULL_HNDL

<i2c/bp_i2c.h>

NULL I2C handle.

Macro

BP_I2C_SLV_ADDR_MASK

<i2c/bp_i2c.h>

7-bit I2C address mask.

Chapter

13

SPI

The SPI module allows transmission and reception through Serial Peripheral Interface(SPI) compatible peripherals along with optional control of the slave select lines. Operation can either be as an SPI master or slave if supported by the peripheral. The API also supports simultaneous transmission and reception in both the master and slave configuration.

The exact handling of the slave select line performed by calling `bp_spi_slave_sel()` and `bp_spi_slave_desel()` is driver and platform specific. The mapping between the slave select id and a physical slave select pin is also platform specific. Additional details are available in the driver's documentation.

Considering the wide varieties of SPI compatible peripherals, it would be impossible to design a high-level API that could leverage the unique features of many peripherals. To alleviate this, drivers are allowed to implement driver-specific functionalities to extend the features of the SPI module. Details of these features can be found in each driver's documentation.

Function

bp_spi_cfg_get()

<spi/bp_spi.h>

Retrieves the current configuration of an SPI interface. If successful, the SPI configuration is returned through `p_cfg`. The configuration returned is derived from the hardware registers and reflects the actual configuration regardless of the last configuration set by `bp_spi_cfg_set()`.

The clock frequency returned is the actual frequency when known, otherwise the `max_clk_speed` member of `p_cfg` is set to 0.

When `bp_spi_cfg_get()` returns with an `RTNC_TIMEOUT` error, the destination of `p_cfg` is left unmodified.

Prototype

```
int  bp_spi_cfg_get  ( bp_spi_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 module instance to query.
`p_cfg`	Pointer to the returned SPI configuration.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

bp_spi_cfg_set()

<spi/bp_spi.h>

Configures an SPI interface. The SPI interface configuration is set from the `p_cfg` argument. If the interface was in the created state, it will transition to the configured state and must be enabled using 'bp_spi_en() before being used. Otherwise the interface configuration is updated.

The underlying driver will attempt to configure the closest clock frequency to the specified frequency. Calling `bp_spi_cfg_get()` will return the actual frequency configured.

When `bp_spi_cfg_set()` returns with an `RTNC_NOT_SUPPORTED` or `RTNC_TIMEOUT` error, it is guaranteed that the current configuration is unaffected.

Not all peripherals and drivers support both master and slave mode. Attempting to set an unsupported mode will return `RTNC_NOT_SUPPORTED`.

Drivers for peripherals that do not support changing the clock speed will ignore the `max_clk_speed` argument. `bp_spi_cfg_get()` will return the fixed speed if known.

It is driver specific whether or not an `RTNC_NOT_SUPPORTED` error is returned on configurations not supported by the underlying peripheral. Unless specified differently by the driver documentation, the following holds true.

- A clock speed of 0 will return an `RTNC_FATAL` error unless it has a special meaning for the hardware.
- Specifying a clock speed outside of the peripheral's supported range will configure the closest supported rate.
- Specifying an unsupported mode will return `RTNC_NOT_SUPPORTED`.
- Drivers for peripherals with a fixed hardware configuration such as soft IPs for FPGAs, will usually ignore any configuration parameters and return successfully.

Prototype

```
int  bp_spi_cfg_set  ( bp_spi_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 module instance to configure.
	`p_cfg`	SPI configuration.
	`timeout_ms`	Timeout value in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_NOT_SUPPORTED`
`RTNC_FATAL`

Example

```
bp_spi_hndl_t spi_hndl;
bp_spi_cfg_t spi_cfg;

spi_cfg.clk_phase = 0u;
spi_cfg.clk_polarity = 1u;
spi_cfg.master = 1u;
spi_cfg.max_clk_speed = 0u;

bp_spi_cfg_set(spi_hndl, \&spi_cfg, TIMEOUT_INF);
```

Function

bp_spi_create()

<spi/bp_spi.h>

Creates an SPI module instance. The created SPI instance is associated with the SPI peripheral definition `p_def`. If successful, a handle to the newly created instance is returned through the `p_hndl` argument. After returning from a successful call to `bp_spi_create()` the newly created instance is in the created state and should subsequently be configured and enabled to be fully functional. See `bp_spi_cfg_set()` and `bp_spi_en()` for details.

The SPI definition structure `p_def` must be unique and can only be associated with a single UART instance. Once created, the SPI instance is assigned a name that can be used afterward to retrieve the interface handle by calling `bp_spi_hndl_get()`. The assigned name is set from the board definition structure `p_def` and must be unique.

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

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

When bp_spi_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_spi_create ( const bp_spi_board_def_t * p_def,
                    bp_spi_hndl_t *            p_hndl );
```

Attributes

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

Parameters

p_def	Definition of the SPI peripheral.
p_hndl	Pointer to the created SPI module instance.

Returned Errors

RTNC_SUCCESS
RTNC_ALREADY_EXIST
RTNC_NO_RESOURCE
RTNC_FATAL

Example

```
extern bp_spi_board_def_t g_spi0;
bp_spi_hndl_t spi_hndl;

bp_spi_create(\&g_spi0, \&spi_hndl);
```

Function

bp_spi_destroy()

<spi/bp_spi.h>

Destroys an SPI module instance. When supported, bp_spi_destroy() will free up all the resources allocated to the SPI module instance, including the peripheral driver and internal data structures. Depending on the memory allocation policy of the default memory allocator it may not be possible to free previously allocated memory, in that case RTNC_NOT_SUPPORTED is returned and the SPI module instance is left unaffected.

It is not necessary, but strongly recommended, to disable an SPI interface by calling bp_spi_dis() before attempting to destroy it. This helps ensure that no race condition exists between the instance destruction and ongoing transfers.

The result of using a UART module handle after its underlying instance is destroyed is undefined.

Prototype

```
int bp_spi_destroy ( bp_spi_hndl_t hndl,
                     uint32_t      timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the SPI module instance to destroy.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_NOT_SUPPORTED`
`RTNC_FATAL`

Function

bp_spi_dis()

<spi/bp_spi.h>

Disables an SPI interface. `bp_spi_dis()` will wait for the interface to be idle before disabling it.

The exact side effects of disabling an interface is driver dependent. In general the peripheral is disabled at the peripheral level, and, when possible, the module clock is gated.

The result of calling `bp_spi_dis()` or any other functions other than `bp_spi_en()` or `bp_spi_reset()` on an already disabled interface is undefined. With assertion checking enabled, some drivers will return `RTNC_FATAL` when attempting to access a disabled interface. The current enabled/disabled state of an interface can be queried using `bp_spi_is_en()`.

Prototype

```
int  bp_spi_dis  ( bp_spi_hndl_t  hndl,
                   uint32_t       timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the SPI module instance to disable.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Function

bp_spi_drv_hndl_get()

<spi/bp_spi.h>

Prototype

```
int  bp_spi_drv_hndl_get ( bp_spi_hndl_t       hndl,
                           bp_spi_drv_hndl_t *  p_drv_hndl );
```

Attributes

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

Parameters

hndl	Handle of the SPI module instance to query.
p_drv_hndl	Pointer to the SPI driver handle.

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Function

bp_spi_en()

<spi/bp_spi.h>

Enables an SPI interface. Enabling an SPI module instance in the disabled or configured state will, depending on the driver, enable the peripheral clock, de-assert reset, if asserted, and enable transmission and reception through the SPI peripheral.

Calling bp_spi_en() on an enabled SPI instance should be without side effect.

Prototype

```
int  bp_spi_en  ( bp_spi_hndl_t  hndl,
                  uint32_t       timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the SPI interface to enable.
timeout_ms	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_spi_flush()

<spi/bp_spi.h>

Flushes the transmit and receive paths. Flush the transmit and receive paths of an SPI interface. It is unspecified whether any data written but not yet transmitted is sent or dropped. Data held in the receive FIFO will be discarded.

Prototype

```
int bp_spi_flush ( bp_spi_hndl_t hndl,
                   uint32_t      timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the SPI module instance to flush.
timeout_ms	Timeout in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_spi_hndl_get()

<spi/bp_spi.h>

Retrieves a previously created SPI instance handle by name. If found, the result is returned through the p_hndl argument, otherwise RTNC_NOT_FOUND is returned and p_hndl is left as it was before the call to bp_uart_hndl_get().

The name of an instance is set in the bp_uart_board_def_t board definition passed to bp_spi_create().

Prototype

```
int bp_spi_hndl_get ( const char *     p_name,
                      bp_spi_hndl_t *  p_hndl );
```

Attributes

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

Parameters

p_name	Name of the SPI instance to retrieve.
p_hndl	Pointer to the SPI interface handle.

Returned Errors

RTNC_SUCCESS
RTNC_NOT_FOUND
RTNC_FATAL

Function

bp_spi_idle_wait()

<spi/bp_spi.h>

Waits for an SPI interface to be idle. bp_spi_idle_wait() will wait for the transfer logic to be idle in master mode and for any transfer operation to be complete in slave mode.

Prototype

```
int bp_spi_idle_wait ( bp_spi_hndl_t hndl,
                       uint32_t      timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the SPI module instance to wait on.
timeout_ms	Timeout in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_spi_is_en()

<spi/bp_spi.h>

Returns the enabled/disabled state of an SPI interface. If the call is successful, the state of the SPI interface is returned through the argument p_is_en.

The state of an interface is checked atomically in a non-blocking way. As such, bp_spi_is_en() can be called while another operation is in progress without blocking or from an interrupt service routine.

Prototype

```
int bp_spi_is_en ( bp_spi_hndl_t hndl,
                   bool *        p_is_en );
```

Attributes

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

Parameters

hndl	Handle of the SPI module instance to query.
p_is_en	Returned interface state, true if enabled false otherwise.

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Function

bp_spi_reset()

<spi/bp_spi.h>

Resets an SPI module instance. Upon a successful call to bp_spi_reset() the SPI interface is left in the created state, equivalent to the state a newly created instance. Before using the instance again, it must be reconfigured and enabled, see bp_spi_cfg_set() and bp_spi_en().

Any asynchronous transfers in progress will be aborted without calling their callback functions.

Prototype

```
int bp_spi_reset ( bp_spi_hndl_t hndl,
                   uint32_t      timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the SPI module instance to reset.
timeout_ms	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_spi_slave_desel()

<spi/bp_spi.h>

Deselects a selected SPI slave. Deselect any selected slave select line of SPI interface hndl and release exclusive control of the SPI interface. The SPI driver will always wait for the current transfer, if any, to be finished before deasserting the slave select line.

When hosted by an RTOS supporting mutexes, only the task that called bp_spi_slave_sel() is allowed to call bp_spi_slave_desel().

bp_spi_slave_desel() should always be called before selecting another slave to properly release the mutex.

Prototype

```
int bp_spi_slave_desel ( bp_spi_hndl_t hndl,
                         uint32_t      timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the SPI module instance to use.
timeout_ms	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_spi_slave_sel()

<spi/bp_spi.h>

Selects a specific SPI slave. Select slave interface `ss_id` of SPI interface `hndl` and take exclusive control of an SPI interface. When hosted on an RTOS, calling `bp_spi_slave_sel()` will acquire a mutex to ensure no other tasks can access the bus. `bp_spi_slave_desel()` must be called to release the bus.

Whether or not the slave select line is actually asserted after calling `bp_spi_slave_sel()` is driver specific. By default, the slave select line will be asserted by calling `bp_spi_slave_sel()` and will be kept asserted until `bp_spi_slave_desel()` is called. Some drivers may support additional modes of operation where the slave select behaves differently, see the driver documentation for details.

The exact mapping of slave select id is specific to the peripheral driver and may depend on driver specific configurations, see the driver documentation for details.

It is driver specified whether `RTNC_NOT_SUPPORTED` or `RTNC_FATAL` is returned when an out of range `ss_id` is specified for the current peripheral. For maximum flexibility, drivers for peripherals that do not support any slave select lines will ignore any selected slave select and return `RTNC_SUCCESS`.

Prototype

```
int  bp_spi_slave_sel  ( bp_spi_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 module instance to use.
`ss_id`	Numeric id of the slave select line to assert.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_NOT_SUPPORTED
RTNC_FATAL

Function

bp_spi_xfer()

<spi/bp_spi.h>

Performs an SPI operation. Transmit and/or receive through SPI interface using the transfer parameters p_tf.

The callback argument of p_tf, which is only used for asynchronous transfers, should be set to NULL.

In master mode, since the SPI protocol operates as a shift register the pointee of p_tf_len will always match the configured length unless an error happens. On error the value of p_tf_len is undefined.

In slave mode the number of bytes returned through p_tf_len will be the actual number of bytes transferred in case of a successful transfer or a receive timeout.

The timeout value is the amount of time to wait for the channel to be available. The time spent to perform the transfer is not counted to consider a timeout condition. Drivers that support querying the bit rate of the interface in master mode can return RTNC_FATAL in case the transfer operation is taking longer than expected.

Prototype

```
int bp_spi_xfer ( bp_spi_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 SPI module instance to use.
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

Example

```
bp_spi_tf_t tf;
size_t rx_len

tf.p_tx_buf = p_tx_buf;
tf.p_rx_buf = p_rx_buf;
tf.len = 100u;
tf.callback = NULL;

bp_spi_xfer(spi_hndl, \&tf, \&rx_len, timeout_ms);
```

Function

bp_spi_xfer_async()

<spi/bp_spi.h>

Transfers data asynchronously. Performs an asynchronous transfer operation according to the parameters of the p_tf argument, see the bp_spi_tf_t structure documentation for an explanation of the transfer parameters. Upon successfully starting a transfer, the function returns immediately. The callback specified in the p_tf structure will be called when the transfer is finished. If no callback is specified a fire and forget transfer will be performed, where the entire operation will be executed in the background. Care should be taken when using such transfers as it's not possible for the application to know if the transfer succeeded.

The timeout argument timeout_ms specifies the amount of time to wait for the channel to be available. The timeout value has no impact on the asynchronous transfer operation once started.

When bp_spi_xfer_async() returns with an RTNC_TIMEOUT error, the transfer is not started and the callback function specified in p_tf won't be called.

The structure referenced by p_tf must be valid for the entire asynchronous transfer operation and may be accessed by the SPI driver. Upon returning, the original state of the transfer will be preserved. p_tf will be passed verbatim to the callback and may be modified within the user callback to perform an additional transfer from the callback.

The p_ctxt member of the p_tf transfer descriptor can be used to pass user context information to the callback.

Prototype

```
int bp_spi_xfer_async ( bp_spi_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 SPI module instance to use for the asynchronous transfer.
p_tf	Transfer parameters.
timeout_ms	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_spi_xfer_async_abort()

<spi/bp_spi.h>

Aborts an asynchronous transfer. Aborts any running asynchronous transfer operation. The number of bytes already transmitted and received will be returned through p_tx_len and p_rx_len if they are not NULL.

In case of a successful abort the transfer callback function of the aborted operation won't be called. It is however, possible for the transfer to finish just before being aborted in which case bp_spi_xfer_async_abort() will return with RTNC_SUCCESS.

Aborting a transfer will clear the transmit and receive FIFOs if any, which can lead to data loss.

Prototype

```
int bp_spi_xfer_async_abort ( bp_spi_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 interface 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_action_t

<spi/bp_spi.h>

Asynchronous IO return action. These are the return value possible to an SPI asynchronous IO callback instructing the driver on the action to be performed. See bp_spi_async_cb_t and bp_spi_xfer_async() for details.

Values

BP_SPI_ACTION_FINISH	Finish normally.
BP_SPI_ACTION_RESTART	Restart a transfer with the data of the current transfer description structure.

Data Type

bp_spi_async_cb_t

<spi/bp_spi.h>

Asynchronous IO callback function pointer. Callback function pointer type to be used with non-blocking asynchronous transfers.

When an asynchronous transfer is finished, the callback will be called, if set. The `status` argument will be one of the following, indicating the result of the transfer:

- RTNC_SUCCESS The transfer is finished normally.
- RTNC_IO_ERR An I/O error occurred.
- RTNC_FATAL A fatal error was detected.

Two actions are possible when returning.

- BP_SPI_ACTION_FINISH Finish the transfer normally.
- BP_SPI_ACTION_RESTART Restart the transfer operation with the data in the `p_tf` transfer description structure.

The transfer descriptor structure is the same that was passed to the initial call to bp_spi_xfer_async(). It can be modified prior to returning BP_SPI_ACTION_RESTART to restart a transfer immediately from the callback using the updated transfer descriptor.

See bp_spi_xfer_async() for usage details.

Prototype

```
bp_spi_action_t  bp_spi_async_cb_t  ( int           status,
                                      size_t        tf_len,
                                      bp_spi_tf_t * p_tf );
```

Attributes

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

Parameters

status	Status of the asynchronous operation.
tf_len	Amount of bytes actually transferred in case of timeout or error.
p_tf	Pointer to the current transfer.

Returned Values

Return value of type bp_spi_action_t to signal the desired operation (terminate or restart).

Data Type

bp_spi_board_def_t

<spi/bp_spi.h>

SPI board-level hardware definition. Complete definition of an SPI interface, including the name, BSP as well as the SoC level definition structure of type `bp_spi_soc_def_t` providing the driver and driver specific parameters. The overall definition of a SPI interface should be unique, including the name, for each SPI module instance to prevent conflicts.

BSP definitions are driver specific and usually not required, when that is the case `p_bsp_def` should be set to NULL. See the driver's documentation for details.

See `bp_spi_create()` for usage details.

Members

`p_soc_def`	`const bp_spi_soc_def_t *`	SoC level definition.
`p_bsp_def`	`const void *`	Board and application specific definition.
`p_name`	`const char *`	SPI peripheral name.

Data Type

bp_spi_cfg_t

<spi/bp_spi.h>

SPI protocol configuration structure. Used to set or return the configuration of an SPI interface.

See `bp_spi_cfg_set()` and `bp_spi_cfg_get()` for usage details.

Members

`bit_rate`	`uint32_t`	Bit rate in Hertz.
`clk_phase`	`uint32_t`	Clock phase 1 or 0.
`clk_polarity`	`uint32_t`	Clock polarity 1 or 0.
`ss_id`	`uint32_t`	Slave select id to configure. Only used on controllers that supports multiple different SPI configuration in hardware.
`master`	`bool`	Set to `true` for master mode `false` for slave.

Data Type

bp_spi_drv_hndl_t

<spi/bp_spi.h>

SPI driver handle. Pointer to driver private data. The pointer contained in the handle is private and should not be accessed by calling code.

Members

p_hndl	void *	Pointer to the SPI driver internal data.

Data Type

bp_spi_hndl_t

<spi/bp_spi.h>

SPI handle. SPI handle returned by `bp_spi_create()`. The pointer contained in the handle is private and should not be accessed by calling code.

Members

p_hndl	bp_spi_inst_t *	Pointer to the SPI internal instance data.

Data Type

bp_spi_soc_def_t

<spi/bp_spi.h>

SPI hardware definition structure.

The SPI hardware definition structure is used to describe the peripheral at the SoC level. It specifies the driver to be used as well as the location, either as an index or more often a base address.

To be complete a SPI hardware instance also requires a board specific portion. Both this structure and the BSP structures are merged into a `bp_spi_board_def_t` structure to describe a complete SPI interface instance.

Members

p_drv	const bp_spi_drv_t *	Driver associated with this peripheral.
p_drv_def	const void *	Driver specific definition.

Data Type

bp_spi_tf_t

<spi/bp_spi.h>

SPI transfer setup structure. Used by the transfer API and the drivers to describe an SPI transfer.

See `bp_spi_xfer()` and `bp_spi_xfer_async()` for usage details.

Members

p_tx_buf	const void *	Pointer to the buffer to transmit.

`p_rx_buf`	`void *`	Memory location of the buffer that will contain the received data.
`len`	`size_t`	Length of the data to receive and/or transmit.
`callback`	`bp_spi_async_cb_t`	Async transfer callback. Should be set to NULL for non-async transfers.
`p_ctxt`	`void *`	Optional user context pointer passed to the asynchronous callback.

Macro

BP_SPI_HNDL_IS_NULL()

<spi/bp_spi.h>

Evaluates if an SPI module handle is NULL.

Prototype `BP_SPI_HNDL_IS_NULL ( hndl );`

Parameters `hndl` Handle to be checked.

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

Macro

BP_SPI_NULL_HNDL

<spi/bp_spi.h>

NULL SPI module handle.

Macro

BP_SPI_SS_NONE

<spi/bp_spi.h>

Special slave select value that represents no specific slave. See `bp_spi_slave_sel()` for usage details.

Chapter

14

UART

The UART module is used to interface with Universal Asynchronous Receiver-Transmitter and other similar serial interface peripherals. UART peripherals are usually comprised of two independent receive and transmit interfaces. To allow for maximum flexibility the UART module is designed to permit concurrent access to both the transmit and receive channel in a thread safe manner without blocking each other.

Some API functions that act on the entire UART peripheral state, such as `bp_uart_cfg_set()` and `bp_uart_dis()` and many others will need to lock both the transmit and receive paths to prevent any possible race conditions. The inner locking is designed to prevent deadlocks from occurring.

Considering the wide varieties of UART and UART-like peripherals, it would be impossible to design a high-level API that could leverage the unique features of many peripherals. To alleviate this, drivers are allowed to implement driver-specific functionalities to extend the features of the UART module. Details of these features can be found in each driver's documentation.

Function

bp_uart_acquire()

<uart/bp_uart.h>

Acquires exclusive access to a UART module instance. Upon a successful call, the UART instance will be accessible exclusively from the current thread.

`bp_uart_acquire()` has no effect in a bare-metal environment.

Prototype

```
int  bp_uart_acquire  ( bp_uart_hndl_t  hndl,
                        uint32_t        timeout_ms );
```

Attributes

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

Parameters	`hndl`	Handle of the UART module instance to acquire.
	`timeout_ms`	Timeout in milliseconds.

Returned Errors
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_uart_cfg_get()

<uart/bp_uart.h>

Retrieves the current configuration of a UART interface. If successful, the UART configuration is returned through `p_cfg`. The configuration returned is derived from the hardware registers and reflects the actual configuration regardless of the last configuration set by `bp_uart_cfg_set()`.

The baud rate returned is the actual baud rate when known, otherwise the `baud_rate` member of `p_cfg` is set to 0.

When `bp_uart_cfg_get()` returns with an `RTNC_TIMEOUT` error, the destination of `p_cfg` is left unmodified.

Prototype

```
int  bp_uart_cfg_get  ( bp_uart_hndl_t   hndl,
                        bp_uart_cfg_t *  p_cfg,
                        uint32_t         timeout_ms );
```

Attributes

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

Parameters	`hndl`	Handle of the UART module instance to query.
	`p_cfg`	Pointer to the UART configuration.
	`timeout_ms`	Timeout value in milliseconds.

Returned Errors
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_uart_cfg_set()

<uart/bp_uart.h>

Configures a UART interface. The UART interface configuration is set from the `p_cfg` argument. If the interface was in the created state, it will transition to the configured state and must be enabled using `bp_uart_en()` before being used. Otherwise the interface configuration is updated.

The underlying driver will attempt to configure the closest baud rate to the specified baud rate. Calling `bp_uart_cfg_get()` will return the actual baud rate configured.

When `bp_uart_cfg_set()` returns with an `RTNC_NOT_SUPPORTED` or `RTNC_TIMEOUT` error, it is guaranteed that the current configuration is unaffected.

It is driver specific whether or not an `RTNC_NOT_SUPPORTED` error is returned on configurations not supported by the underlying peripheral. Unless specified differently by the driver documentation, the following holds true.

- A baud rate of 0 will return an `RTNC_FATAL` error unless it has a special meaning for the hardware.
- Specifying a baud rate outside of the peripheral's supported range will configure the closest supported rate.
- Specifying an unsupported parity will return `RTNC_NOT_SUPPORTED`.
- The configuration for one and a half and two stop bits can be used interchangeably by the driver if one of them is not supported by the hardware.
- In case both one and one and a half stop bits are unsupported, `RTNC_NOT_SUPPORTED` is returned if either one is specified.
- Drivers for peripherals with a fixed hardware configuration such as soft IPs for FPGAs, or virtual UART interfaces will usually ignore any configuration parameters and return successfully.

Prototype

```
int  bp_uart_cfg_set ( bp_uart_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 module instance to configure.
`p_cfg`	UART configuration to apply.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_NOT_SUPPORTED`
`RTNC_FATAL`

Example

```
bp_uart_hndl_t uart_hndl;
bp_uart_cfg_t uart_cfg;

bp_uart_cfg.baud_rate = 115200u;
bp_uart_cfg.parity = UART_PARITY_NONE;
bp_uart_cfg.stop_bits = UART_STOP_BITS_1;

bp_uart_cfg_set(uart_hndl, \&uart_cfg, TIMEOUT_INF);
```

Function

bp_uart_create()

<uart/bp_uart.h>

Creates a UART module instance. The created UART instance is associated with the UART peripheral definition p_def. If successful, a handle to the newly created instance is returned through the p_hndl argument. After returning from a successful call to bp_uart_create() the newly created instance is in the created state and should subsequently be configured and enabled to be fully functional. See bp_uart_cfg_set() and bp_uart_en() for details.

The UART definition structure p_def must be unique and can only be associated with a single UART instance. Once created, the UART instance is assigned a name that can be used afterward to retrieve the interface handle by calling bp_uart_hndl_get(). The assigned name is set from the board definition structure p_def and must be unique.

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

The board definition p_def passed to bp_uart_create() must be kept valid for the lifetime of the UART module instance.

When bp_uart_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_uart_create ( const bp_uart_board_def_t * p_def,
                     bp_uart_hndl_t *            p_hndl );
```

Attributes

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

Parameters

p_def	Definition of the UART peripheral.
p_hndl	Pointer to the created UART module instance.

Returned Errors

RTNC_SUCCESS
RTNC_ALREADY_EXIST
RTNC_NO_RESOURCE
RTNC_FATAL

Example

```
extern bp_uart_board_def_t g_uart0;
bp_uart_hndl_t uart_hndl;

bp_uart_create(\&g_uart0, \&uart_hndl);
```

Function

bp_uart_destroy()

<uart/bp_uart.h>

Destroys a UART module instance. When supported, bp_uart_destroy() will free up all the resources allocated to the UART module instance, including the peripheral driver and internal data structures. Depending on the memory allocation policy of the default memory allocator it may not be possible to free previously allocated memory, in that case RTNC_NOT_SUPPORTED is returned and the UART module instance is left unaffected.

It is not necessary, but strongly recommended, to disable a UART instance by calling bp_uart_dis() before attempting to destroy it. This helps ensure that no race condition exists between the instance destruction and ongoing transfers.

The result of using a UART module handle after its underlying instance is destroyed is undefined.

Prototype

```
int bp_uart_destroy ( bp_uart_hndl_t hndl,
                      uint32_t       timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the UART module instance to destroy.
timeout_ms	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_NOT_SUPPORTED
RTNC_FATAL

Function

bp_uart_dis()

<uart/bp_uart.h>

Disables a UART interface. bp_uart_dis() will wait for the interface to be idle before disabling it.

The exact side effects of disabling an interface is driver dependent. In general the peripheral is disabled at the peripheral level, and, when possible, the module clock is gated.

The result of calling bp_uart_dis() or any other functions other than bp_uart_en() or bp_uart_reset() on an already disabled interface is undefined. With assertion checking enabled, some drivers will return RTNC_FATAL when attempting to access a disabled interface. The current enabled/disabled state of an interface can be queried using bp_uart_is_en().

Prototype

```
int  bp_uart_dis  ( bp_uart_hndl_t  hndl,
                    uint32_t        timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the UART module instance to disable.
timeout_ms	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_uart_drv_hndl_get()

<uart/bp_uart.h>

Returns the driver handle associated with a UART module instance. The underlying driver handle will be returned through p_drv_hndl. The driver handle can be used to perform operations with the driver interface directly or to access driver specific features.

Prototype

```
int  bp_uart_drv_hndl_get  ( bp_uart_hndl_t        hndl,
                             bp_uart_drv_hndl_t *  p_drv_hndl );
```

Attributes

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

Parameters

hndl	Handle of the UART module instance to query.
p_drv_hndl	Pointer to the UART driver handle.

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Function

bp_uart_en()

<uart/bp_uart.h>

Enables a UART interface. Enabling a UART module instance in the disabled or configured state will, depending on the driver, enable the peripheral clock, de-assert reset, if asserted, and enable transmission and reception through the UART peripheral.

Calling bp_uart_en() on an enabled UART instance should be without side effect.

Prototype

```
int bp_uart_en ( bp_uart_hndl_t hndl,
                 uint32_t       timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the UART module instance to enable.
timeout_ms	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_uart_hndl_get()

<uart/bp_uart.h>

Retrieves a previously created UART instance handle by name. If found, the result is returned through the p_hndl argument, otherwise RTNC_NOT_FOUND is returned and p_hndl is left as it was before the call to bp_uart_hndl_get().

The name of an instance is set in the bp_uart_board_def_t board definition passed to bp_uart_create().

Prototype

```
int bp_uart_hndl_get (                  p_if_name,
                      bp_uart_hndl_t *  p_hndl );
```

Attributes

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

Parameters

p_if_name	Name of the UART instance to retrieve.
p_hndl	Pointer to the UART interface handle.

Returned Errors

RTNC_SUCCESS
RTNC_NOT_FOUND
RTNC_FATAL

Function

bp_uart_is_en()

<uart/bp_uart.h>

Returns the enabled/disabled state of a UART interface. If the call is successful the state of the UART interface is returned through the argument p_is_en.

The state of an interface is checked atomically in a non-blocking way. As such, bp_uart_is_en() can be called while another operation is in progress without blocking or from an interrupt service routine.

Prototype

```
int bp_uart_is_en ( bp_uart_hndl_t hndl,
                    bool *         p_is_en );
```

Attributes

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

Parameters

hndl	Handle of the UART module instance to query.
p_is_en	Interface state, true if enabled false otherwise.

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Function

bp_uart_release()

<uart/bp_uart.h>

Releases exclusive access to a UART interface.

bp_uart_release() has no effect in a bare-metal environment.

Prototype

```
int bp_uart_release ( bp_uart_hndl_t hndl );
```

Attributes

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

Parameters `hndl` Handle of the UART module instance to release.

Returned Errors
RTNC_SUCCESS
RTNC_FATAL

Function

bp_uart_reset()

<uart/bp_uart.h>

Resets a UART module instance. Upon a successful call to `bp_uart_reset()` the UART interface is left in the created state, equivalent to the state a newly created instance. Before using the instance again, it must be reconfigured and enabled, see `bp_uart_cfg_set()` and `bp_uart_en()`.

Any asynchronous transfer in progress will be aborted without calling their callback functions.

Prototype

```
int  bp_uart_reset  ( bp_uart_hndl_t  hndl,
                      uint32_t        timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the UART interface to reset.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_uart_rx()

<uart/bp_uart.h>

Receives data. Receives up to `len` bytes from a UART interface into buffer `p_buf`. On completion, the actual number of bytes received is returned through `p_recv_len` if it's not NULL.

When a timeout value of 0 is specified, the UART driver will return any data, up to `len` bytes, that is available from the receive FIFO and return immediately. If `len` bytes were read from the FIFO, `RTNC_SUCCESS` is returned, otherwise `RTNC_TIMEOUT` is returned.

If supported by the UART driver and the underlying hardware, receive errors, such as parity, framing, and breaks will cause an immediate return with an `RTNC_IO_ERR` error. The number of bytes read up to that point is then returned through `p_recv_len`. It is driver dependent whether bytes with an error detected are written to the receive buffer or discarded. See the driver's documentation for details on how invalid bytes are handled.

When bp_uart_rx() returns with an RTNC_IO_ERR error, it is driver specific whether or not the invalid bytes are written to the receive buffer. When it is, the returned number of bytes read includes the invalid data. See the driver's documentation for details.

A NULL p_rx_len can be passed if the number of bytes read is of no interest to the caller.

Prototype

```
int bp_uart_rx ( bp_uart_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 UART modules instance 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_uart_rx_async()

<uart/bp_uart.h>

Receives data asynchronously. Performs an asynchronous receive operation according to the parameters of the p_tf argument, see the bp_uart_tf_t documentation for an explanation of the transfer parameters. Upon successfully starting a transfer, the function returns immediately. The callback specified in the p_tf structure will be called when the transfer is finished. If no callback is specified, a fire and forget transfer will be performed, where the entire operation will be executed in the background. Care should be taken when using such transfers as it's not possible for the application to know if the transfer succeeded.

The timeout argument timeout_ms specifies the amount of time to wait for the channel to be available. The timeout value has no impact on the asynchronous transfer operation once started.

When bp_uart_tx_async() returns with an RTNC_TIMEOUT error, the transfer is not started and the callback function specified in p_tf won't be called.

The structure referenced by p_tf must be valid for the entire asynchronous transfer operation and may be accessed by the UART driver. Upon returning, the original state of the transfer descriptor will be preserved. p_tf will be passed verbatim to the callback and may be modified within the user callback to perform an additional transfer from the callback.

The p_ctxt member of the p_tf transfer descriptor can be used to pass user context information to the callback.

Prototype

```
int bp_uart_rx_async ( bp_uart_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 UART module instance to use for reception.
p_tf	Transfer parameters.
timeout_ms	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_uart_rx_async_abort()

<uart/bp_uart.h>

Aborts an asynchronous reception. Aborts any running asynchronous reception operation. The number of bytes already received will be returned through p_rx_len if it's not NULL.

In case of a successful abort, the transfer callback function will be not be called. It is, however, possible for the transfer to finish just before being aborted in which case bp_uart_tx_async_abort() will return with RTNC_SUCCESS and the number of bytes received will be 0.

In case no asynchronous reception operation is in progress bp_uart_rx_async_abort() will return RTNC_SUCCESS and the number of bytes received returned will be 0.

Prototype

```
int bp_uart_rx_async_abort ( bp_uart_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 UART module instance 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_uart_rx_flush()

<uart/bp_uart.h>

Flushes the receive path. The receive FIFO of the UART interface is cleared, any data pending in the UART FIFO is discarded.

Prototype

```
int bp_uart_rx_flush ( bp_uart_hndl_t hndl,
                       uint32_t       timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the UART module to flush.
timeout_ms	Timeout in milliseconds.

Returned Errors RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_uart_rx_idle_wait()

<uart/bp_uart.h>

Waits for a UART interface receive path to be idle.

Prototype

```
int bp_uart_rx_idle_wait ( bp_uart_hndl_t hndl,
                           uint32_t       timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the UART module instance to wait on.
timeout_ms	Timeout in milliseconds.

Returned Errors RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_uart_tx()

<uart/bp_uart.h>

Transmits data. Transmits len bytes from buffer p_buf through a UART interface.

The timeout value specifies the amount of time to wait for the channel to be available. The time spent to perform the transfer is not counted to consider a timeout condition.

UART peripherals do not usually have a way to detect transmission issues. However, for those peripherals that can, and when the error is not due to a software or internal hardware issue, RTNC_IO_ERR can be returned by the driver, see the driver's documentation for details.

Drivers are allowed to use an internal timeout, independent of the timeout_ms argument, to detect a stuck peripheral when a transmit operation is taking longer than expected. An RTNC_FATAL error is returned in those cases, see the driver's documentation for details.

It is unspecified how many data, if any, was actually transmitted from a failed transfer.

Prototype

```
int bp_uart_tx ( bp_uart_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 UART module instance 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_uart_tx_async()

<uart/bp_uart.h>

Transmits data asynchronously. Performs an asynchronous transmit operation according to the parameters of the p_tf argument, see the bp_uart_tf_t documentation for an explanation of the transfer parameters. Upon successfully starting a transfer the function returns immediately. The callback specified in the p_tf structure will be called when the transfer is finished. If no callback is specified, a fire and forget transfer will be performed, where the entire operation will be executed in the background. Care should be taken when using such transfers as it's not possible for the application to know if the transfer succeeded.

The timeout argument timeout_ms specifies the amount of time to wait for the channel to be available. The timeout value has no impact on the asynchronous transfer operation once started.

When bp_uart_tx_async() returns with an RTNC_TIMEOUT error, the transfer is not started and the callback function specified in p_tf won't be called.

The structure referenced by p_tf must be valid for the entire asynchronous transfer operation and may be accessed by the UART driver. Upon returning, the original state of the transfer descriptor will be preserved. p_tf will be passed verbatim to the callback and may be modified within the user callback to perform an additional transfer from the callback.

The p_ctxt member of the p_tf transfer descriptor can be used to pass user context information to the callback.

Prototype

```
int  bp_uart_tx_async  ( bp_uart_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 UART module instance to use for transmission.
p_tf	Transfer parameters.
timeout_ms	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_uart_tx_async_abort()

<uart/bp_uart.h>

Aborts an asynchronous transmission. Aborts any running asynchronous transmission operation. The number of bytes already transmitted will be returned through p_tx_len if it's not NULL.

In case of a successful abort, the transfer callback function will not be called. It is, however, possible for a transfer to finish just before being aborted in which case bp_uart_tx_async_abort() will return with RTNC_SUCCESS and the number of bytes transmitted returned will be 0.

In case no asynchronous transfer operation is in progress bp_uart_tx_async_abort() will return RTNC_SUCCESS and the number of bytes transmitted will be 0.

Prototype

```
int bp_uart_tx_async_abort ( bp_uart_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 UART module instance 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_uart_tx_flush()

<uart/bp_uart.h>

Flushes the transmit path. Empty the transmit FIFO of the UART interface. It is unspecified whether any data written but not yet transmitted is sent or dropped.

Prototype

```
int bp_uart_tx_flush ( bp_uart_hndl_t hndl,
                       uint32_t       timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the UART module instance to flush.
`timeout_ms`	Timeout in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_uart_tx_idle_wait()

<uart/bp_uart.h>

Waits for a UART interface transmit path to be idle.

Prototype

```
int bp_uart_tx_idle_wait ( bp_uart_hndl_t hndl,
                           uint32_t       timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the UART module instance to wait on.
`timeout_ms`	Timeout in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type

bp_uart_action_t

<uart/bp_uart.h>

Asynchronous IO return action. These are the return values possible to a UART asynchronous IO callback, instructing the driver on the action to be performed. See `bp_uart_tx_async()`, `bp_uart_rx_async()` and `bp_uart_async_cb_t` for usage details.

Values

`BP_UART_ACTION_FINISH`	Finish normally.
`BP_UART_ACTION_RESTART`	Restart a transfer with the data of the current transfer description structure.

Data Type

bp_uart_parity_t

<uart/bp_uart.h>

UART parity type. For use to specify the UART parity setting within the `bp_uart_cfg_t` configuration structure.

See `bp_uart_cfg_t`, `bp_uart_cfg_set()` and `bp_uart_cfg_get()` for usage details.

Values

`BP_UART_PARITY_NONE`	No parity.
`BP_UART_PARITY_ODD`	Odd parity.
`BP_UART_PARITY_EVEN`	Even parity.
`BP_UART_PARITY_MARK`	Mark parity.
`BP_UART_PARITY_SPACE`	Space parity.

BP_UART_PARITY_NULL	Special invalid value.

Data Type

bp_uart_stop_bits_t

<uart/bp_uart.h>

UART stop bits configuration. Number of stop bits for use with the bp_uart_cfg_t configuration structure. Some of these values may be interpreted slightly differently by some drivers, such as 1.5 stop bits may be interpreted as 2 stop bits if the UART peripheral doesn't support one and a half stop bits.

See bp_uart_cfg_t, bp_uart_cfg_set() and bp_uart_cfg_get() for usage details.

Values

BP_UART_STOP_BITS_1	One stop bit.
BP_UART_STOP_BITS_1_5	On and a half stop bits.
BP_UART_STOP_BITS_2	Two stop bits.
BP_UART_STOP_BITS_NULL	Special invalid value.

Data Type

bp_uart_async_cb_t

<uart/bp_uart.h>

Asynchronous IO callback function pointer. Callback function pointer type to be used with non-blocking asynchronous transfers.

When an asynchronous transfer is finished, the callback will be called if set. The status argument will be one of the following, indicating the result of the transfer:

- RTNC_SUCCESS The transfer finished normally.
- RTNC_IO_ERR An I/O error occurred.
- RTNC_FATAL A fatal error was detected.

Two actions are possible when returning.

- BP_UART_ACTION_FINISH Finish the transfer normally.
- BP_UART_ACTION_RESTART Restart the transfer operation from the updated p_tf transfer description structure.

The transfer descriptor structure is the same that was passed to the initial call to bp_uart_tx_async() or bp_uart_rx_async(). It can be modified prior to returning BP_UART_ACTION_RESTART to restart a transfer immediately from the callback using the updated transfer descriptor.

See bp_uart_tx_async() and bp_uart_rx_async() for usage details.

Prototype

```
bp_uart_action_t  bp_uart_async_cb_t  ( int            status,
                                         size_t         tf_len,
                                         bp_uart_tf_t * p_tf );
```

Attributes

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

Parameters

`status`	Status of the asynchronous operation.
`tf_len`	Number of bytes actually read or written.
`p_tf`	Pointer to the current transfer.

Returned Values

Return value of type `bp_uart_action_t` to signal the desired operation (Terminate or restart).

Data Type

bp_uart_board_def_t

<uart/bp_uart.h>

UART board level hardware definition. Complete definition of a UART interface, including the name, BSP as well as the SoC level definition structure of type `bp_uart_soc_def_t` providing the driver and driver specific parameters. The overall definition of a UART interface should be unique, including the name, for each UART module instance to prevent conflicts.

BSP definitions are driver specific and usually not required, when that is the case `p_bsp_def` should be set to NULL. See the driver's documentation for details.

See `bp_uart_create()` for usage details.

Members

`p_soc_def`	`const bp_uart_soc_def_t *`	SoC level hardware definition.
`p_bsp_def`	`const void *`	Board and application specific definition.
`p_name`	`const char *`	UART peripheral name.

Data Type

bp_uart_cfg_t

<uart/bp_uart.h>

UART protocol configuration structure. Used to set or return the configuration of a UART interface.

See `bp_uart_cfg_set()` and `bp_uart_cfg_get()` for usage details.

Members

baud_rate	uint32_t	Baud rate.
parity	bp_uart_parity_t	Parity.
stop_bits	bp_uart_stop_bits_t	Number of stop bits.

Data Type

bp_uart_drv_hndl_t

<uart/bp_uart.h>

UART driver handle. The pointer contained in the handle is private and should not be accessed by calling code. Used by the application to access the driver directly.

See bp_uart_driver_create_t and the driver documentation for details.

Members

p_hndl	void *	Pointer to the internal UART driver data.

Data Type

bp_uart_hndl_t

<uart/bp_uart.h>

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

Members

p_hndl	bp_uart_inst_t *	Pointer to the UART module internal instance data.

Data Type

bp_uart_soc_def_t

<uart/bp_uart.h>

UART module SoC level hardware definition structure.

The UART hardware definition structure is used to describe the peripheral at the SoC level. The structure specifies the driver to be used as well as a driver specific definition structure usually specifying the location, clock, interrupt and various other parameters required by each UART drivers.

To be complete, a UART hardware instance also requires a board specific portion. Both this structure and the BSP structures are referenced by a bp_uart_board_def_t structure to describe a form a complete UART interface definition.

Members

p_drv	const bp_uart_drv_t *	Driver associated with this interface.
p_drv_def	const void *	Driver specific definition structure.

Data Type

bp_uart_tf_t

<uart/bp_uart.h>

UART transfer setup structure. Used for asynchronous transfers and internally by some drivers.

Members

p_buf	void *	Memory buffer to transmit from or receive to.
len	size_t	Length of the data to transmit or receive in bytes.
callback	bp_uart_async_cb_t	Asynchronous transfer callback function.
p_ctxt	void *	Optional user context pointer passed to the asynchronous callback.

Macro

BP_UART_HNDL_IS_NULL()

<uart/bp_uart.h>

Evaluates if a UART module handle is NULL.

Prototype `BP_UART_HNDL_IS_NULL ( hndl );`

Parameters `hndl` Handle to be checked.

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

Macro

BP_UART_NULL_HNDL

<uart/bp_uart.h>

NULL UART handle.

Macro

BP_UART_PARITY_IS_VALID()

<uart/bp_uart.h>

Checks if UART parity value is valid.

Expansion `true` if the parity value is valid. `false` otherwise.

Macro

BP_UART_STOP_BITS_IS_VALID()

<uart/bp_uart.h>

Checks if UART stop bits value is valid.

Expansion `true` if the stop bits value is valid. `false` otherwise.

Chapter

15

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

16

Architecture Definitions

Definitions used by the architecture module to set the CPU architecture, compiler and endianness.

Macro

BP_ARCH_CPU_ARM_V5

<arch/bp_arch_def.h>

ARM v5, for example the ARM9.

Macro

BP_ARCH_CPU_ARM_V6

<arch/bp_arch_def.h>

ARM v6, for example the ARM11.

Macro

BP_ARCH_CPU_ARM_V6M

<arch/bp_arch_def.h>

ARM v6m, for example the Cortex-M0.

Macro

BP_ARCH_CPU_ARM_V7AR

<arch/bp_arch_def.h>

ARM v7ar, for example the Cortex-A9 or Cortex-R5.

Macro

BP_ARCH_CPU_ARM_V7M

<arch/bp_arch_def.h>

ARM v7m, for example the Cortex-M4.

Macro

BP_ARCH_CPU_ARM_V8A

<arch/bp_arch_def.h>

ARM v8a, for example the Cortex-A53.

Macro

BP_ARCH_CPU_ARM_V8M

<arch/bp_arch_def.h>

ARM v8a, for example the Cortex-M23.

Macro

BP_ARCH_CPU_ARM_V8R

<arch/bp_arch_def.h>

ARM v8r, for example the Cortex-R52.

Macro

BP_ARCH_CPU_LINUX

<arch/bp_arch_def.h>

Linux, any architecture.

Macro

BP_ARCH_CPU_MICROBLAZE

<arch/bp_arch_def.h>

Xilinx Microblaze soft processor.

Macro

BP_ARCH_CPU_NONE

<arch/bp_arch_def.h>

CPU architectures definitions. The macro `BP_ARCH_CPU` will be defined to one of the following by the architecture port.No or invalid architecture.

Macro

BP_ARCH_CPU_SPARCV8

<arch/bp_arch_def.h>

SPARC v8.

Macro

BP_ARCH_CPU_SPARCV9

<arch/bp_arch_def.h>

SPARC v9.

Chapter

17 GPIO Driver

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

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

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

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

Timer Implementation

The declarations found in this module serves as the basis of the timer module implementations. User application should not usually call these functions directly and should instead use the timer module API.

Function

bp_timer_impl_halt()

<timer/bp_timer_impl.h>

Halts the timer processing.

This is an internal function and should not be called from application code.

Prototype

```
int bp_timer_impl_halt ( );
```

Attributes

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

Returned Errors

RTNC_SUCCESS
RTNC_FATAL

Function

bp_timer_impl_init()

<timer/bp_timer_impl.h>

Timer implementation init function.

This is an internal function and should not be called from application code.

Prototype `int bp_timer_impl_init ( );`

Attributes

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

Returned Errors
RTNC_SUCCESS
RTNC_FATAL

Function

bp_timer_impl_next_update()

<timer/bp_timer_impl.h>

Updates the next expiration target.

This is an internal function and should not be called from application code.

Prototype `int bp_timer_impl_next_update (uint64_t target);`

Attributes

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

Parameters
`target` Updated target.

Returned Errors
RTNC_SUCCESS
RTNC_FATAL

Function

bp_timer_impl_resume()

<timer/bp_timer_impl.h>

Resumes the timer processing.

This is an internal function and should not be called from application code.

Prototype `int bp_timer_impl_resume ( );`

Attributes

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

Returned Errors	RTNC_SUCCESS RTNC_FATAL

Chapter 22

Document Revision History

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