JBLopen

Embedded Software Insight

BASEplatform User Manual

UM0001

November 20, 2020

Contents

Chapter

1 Introduction

1.1 About the User Manual

Welcome to the BASEplatform™ User Manual. This manual covers the structure, inner working and usage of the BASEplatform cross platform SDK along with examples code for many of the BASEplatform API. Note that additional documentation should be consulted along with this manual. Those documents are described in the roadmap section below. Additionally a copy of the BASEplatform API Reference Manual is included at the end of this manual, see Chapter 17, however a standalone API Reference Manual is also available from the documentation section of our website.

1.1.1 Audience

This user manual has been written with application designers who are using or are interested in using the BASEplatform in their project. Some background in embedded and especially C programming is assumed. Existing BASEplatform customers who want to follow along and experiment with the code can find a fully functional development project along with a getting started guide for their chosen platform and IDE within the BASEplatform release package. Readers who are interested in evaluating the BASEplatform should request an evaluation project are invited to contact us directly.

1.1.2 Additional Documentation

In addition to this user manual readers are advised to look-up the BASEplatform API Reference Manual available from the documentation section of our website. A copy of the API Reference Manual is also included at the end of this document, see Chapter 17. Users should also be aware of the following documents that should be part of their distribution package.

- Platform Reference Manual
- Getting Started Guide
- Hardware Errata Summary Report
- Readme and Changelog
- Test Report

1.2 Notation and Conventions

1.2.1 Size and Speed Units

Sizes and speeds are given using either binary or decimal units, whichever is best to describe the situation at hand. The most common size units, along with their corresponding values, are given in Table 1.

Bit unit symbol	Value in bits	Byte unit symbol	Value in bytes
kbit	1000 bits	kB	1000 bytes
Kibit	1024 bits	KiB	1024 bytes
Mbit	1000 kbit	MB	1000 kB
Mibit	1024 Kibit	MiB	1024 KiB
Gbit	1000 Mbit	GB	1000 MB
Gibit	1024 Mibit	GiB	1024 MiB
Tbit	1000 Gbit	TB	1000 GB
Tibit	1024 Gibit	TiB	1024 GiB

Table 1 – Most common size units and their corresponding values.

1.2.2 Text Formatting

`Monospace` is used throughout this user manual to indicate an element of code like a function name or a data type (e.g. `bp_time_sleep()`, `RTNC_SUCCESS`, `bp_uart_hndl_t`).

1.2.3 Abbreviations and Acronyms

Abbreviations and acronyms are used throughout this manual for the sake of conciseness. When a word or expression appears for the first time, it is written in full, along with the abbreviated form between parentheses. Subsequent occurrences of the same word or expression can then appear solely in the abbreviated form. A complete list of abbreviations and acronyms is given in Table 2.

Abbreviation	Meaning
API	Application Programming Interface
BSP	Board Support Package
CPU	Central Processing Unit
eMMC	Embedded Multi Media Card
GPIO	General Purpose Input Output
I/O	Input/Output
I2C	Inter-Integrated Circuit
ISR	Interrupt Service Routine
MCU	Microcontroller
RAM	Random Access Memory
RTOS	Real-Time Operating System
SD Card	Secure Digital Card
SoC	System on Chip
SPI	Serial Peripheral Interface
TSFS	TreeSpan File System
UART	Universal Asynchronous Receiver Transmitter

Table 2 – Abbreviations and Acronyms used in this manual.

Chapter

2

What is the BASEplatform

2.1 Introduction

The BASEplatform is a modular cross platform SDK written in C designed to offer application developers with all the necessary software modules, drivers and BSP for a successful embedded application. The BASEplatform can simplify the integration of the developer's choice of RTOS or function standalone in a bare-metal design. The BASEplatform can be used by itself, in a standalone configuration, to provide all the required support code for a selected platform or it can be integrated with an existing SDK.

The BASEplatform robust API provides all the required functionalities expected by a platform support SDK such as CPU, core and SoC initialization and control, interrupt management, low-speed serial communication through GPIO, UART, SPI or I2C and more. On top of that it can also offer mass storage capabilities, either by itself, or through an additional file system. The BASEplatform storage subsystem can interface with most types of embedded flash memory such as NOR, NAND and SD/MMC. Finally, Ethernet and USB connectivity are possible when coupled with a suitable protocol stack.

2.2 Key Features

Cross-platform RTOS agnostic SDK The BASEplatoform can support any MCU or SoC using a portable API. It also is not tied to a specific RTOS being able to run with both commercial and open source RTOS or simply in bare metal. The BASEplatform is also able to support a complete platform by itself without third party code or can be integrated into an existing SDK to extend the functionalities of the native SDK.

Modular and Adaptable Application developers can use as little or as many BASEplatform modules as required by the application. For example, the BASEplatform can be used solely to supply a storage media driver for a file system or the complete platform including low-speed serial communication and platform initialization and configuration.

SMP, UMP and AMP support The BASEplatform is compatible with uniprocessor, symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP) all with the same API.

Robust and Consistent API Since all the modules and drivers are designed and written from scratch the overall API is consistent from module to module. The error handling is also strictly implemented to offer the best possible reliability all the while reducing the development overhead. Finally, timeouts are optionally available on all blocking calls, enabling an easy defensive programming approach to handle unexpected deadlocks or peripheral failures.

External Board Component Support The BASEplatform can include drivers and modules to support external board components such as sensors, Ethernet PHY, non-volatile storage such as NOR, NAND and SD/MMC, power management ICs, I/O expanders and more.

2.3 Requirements

The BASEplatform is designed to support the widest range of platforms, toolchains and RTOSes without compromising features and performance. Platform support can range from small low power MCU to large heterogeneous multi-core System on Chips. Here's a short summary of the important system and toolchain requirements.

- 32 or 64-bit architecture is recommended for optimal performance.
- RAM and ROM requirements are dependent on the chosen RTOS. For estimation purposes, 3-5 KiB of ROM should be allocated per peripheral module and 64 bytes of RAM per peripheral instance when running in bare-metal.
- SMP (Symmetric MultiProcessing) platforms and AMP (Asymmetric MultiProcessing) applications on SMP processors (i.e. two RTOS instances on a dual core Cortex-A9) must be cache coherent.
- ISO/IEC 9899:1999 (a.k.a C99) compliant compiler.

2.3.1 Timebase Recommendations

The BASEplatform doesn't have any firm requirements with respect to the primary timebase. However for best results it is recommended to use a high-resolution (sub-millisecond) free running counter with compare-and-match capabilities. Using a high-resolution timer will allow the `time` and `timer` modules to perform and measure sub-millisecond delays. Also using a free-running compare-and-match counter instead of an auto-reload counter helps reduce jitter and prevent drift of the primary timebase. Additional information about the primary timebase can be found in Chapter 9.

2.4 Bare-Metal, RTOS and Third-Party Software

The BASEplatform can be used in a bare-metal or RTOS environment. When used with an RTOS the RTOS kernel port is usually provided by the BASEplatform as well, although it is possible to integrate the native RTOS port if desired.

The BASEplatform can also integrate other third-party software components. The range of third-party software that can be integrated is rather large but, for example

- Storage: Embedded Databases and File Systems
- Security: Signing and Encryption Libraries
- Acceleration: Image and Video processing, Crypto engines, FPGA Accelerators
- Communication: Protocol Stacks
- Tools: Debugging and Tracing

2.5 Native vs. Hosted

To maximize the versatility of the BASEplatform it can be used in two configurations, native and hosted. Which configuration is used depends on the application needs and current platform support code provided by the manufacturer.

2.5.1 Native Configuration

When the native configuration is used the BASEplatform is the provider of all the low-level support code, including the startup code, kernel port if any, interrupt management, etc. The native configuration is also known as the standalone configuration as in this configuration the BASEplatform is self-contained and can provide all the services required to function on a specific SoC. Note that the native configuration can be used in both bare-metal and with a third-party RTOS. It is also possible to integrate third-party components within a native configuration of the BASEplatform.

2.5.2 Hosted Configuration

In a hosted configuration, the BASEplatform is used on top of an existing SDK. This can be used to integrate the BASEplatform within a manufacturer's SDK or to use existing platform support code provided by the user. The degree of reliance of the BASEplatform on the native SDK or existing code can vary depending on the provided features of the existing code.

2.6 Conclusion

Whether you need a single driver or support for a complete platform the BASEplatform can help reduce risk and time to market by providing well designed, tested and documented peripheral driver and software integration.

Chapter

3

Modules Overview

3.1 Introduction

This chapter presents an overview of the major modules found within the BASEplatform. Apart from a few core modules, most of the modules are optional. An application could use a small number of the BASEplatform modules while another application could make use of all of them. Modules can be informally divided, for the sake of this overview, in three categories, top-level modules, hardware-specific modules and third party integration modules. The following list mostly describes the top-level modules but this chapter goes over a brief overview of what can be found in the two other categories.

3.2 Top-Level Modules

Informally, top-level modules are those modules that provides the primary cross platform API of the BASEplatform. Top-level modules, as the name suggest, are also usually at the top of a stack of modules. For example, the SPI module is a top-level module, with the same API across all platforms. An SPI driver for example, for the Xilinx Zynq SoC is not a top-level module. The distinction is somewhat informal, however, and is mostly used when convenient.

architecture The architecture module or `arch` for brevity provides both the CPU architecture and compiler/toolchain abstraction required by the other BASEplatform modules as well as the application. This includes CPU core control, global interrupt disable/enable, low-level core entry and initialization functions, memory barriers, etc. The current architecture port files are nearly always fully contains within the arch module. The arch module has two important dependencies, in the form of configuration header files, that must be provided by the application.

The first configuration header file which is included from the arch module contains preprocessor macro definitions that affects nearly all the modules of the BASEplatform. This configuration file is named `bp_cfg.h`. This master configuration file contains various important user-configurable parameters. See Chapter 8 for more details.

The second configuration header file is the architecture definition configuration file, `bp_arch_def_cfg.h`. This file should contain a single include directive to include the current architecture definition master header file. Those header files are provided with the BASEplatform but must be selected by this application configuration. This removes the necessity of adding an additional compiler include path to find the architecture specific header file. It also has the advantages of allowing those files to have a unique name. Again, see Chapter 8 for more details.

cache For platforms with CPU cache, especially CPU data cache, the cache module provides cache management function to enable, disable, configure and most importantly clean and invalidate the caches. The cache management API is most often used by the drivers but can be used by the application as well when performing DMA transfers from cacheable memory. See Chapter 12 for details on how to use the cache management API when performing DMA operation.

slock The spinlock and critical section module, abbreviated `slock`, provides synchronization primitives suitable for single core and multi-core platforms and are interoperable with both bare-metal or with a real-time kernel. Additionally, the spinlock API is compatible with bare-metal applications allowing API compatibility between UMP and SMP applications. For details on using this module for synchronization read Chapter 11.

time The time module provides the primary high-resolution timebase used by the BASEplatform as well as time measurement and delay functions. On a typical platform, the time module offers low-jitter sub-millisecond resulting time measurement and delays. Delays can be in the form of spin loops for short microsecond length delays or interrupt based for longer delays. See Chapter 9 for more information.

timer The BASEplatform includes high-resolution interrupt based software timers in the form of the timer module. The software timers are usually of a higher resolution and with a lower overhead compared to typical kernel software timers. Features of the timer module include the usual choice between one-shot and auto-reload timers as well as the option of changing the timer period or stopping the timer within the timer's callback function. Again, see Chapter 9 for more information.

soc SoC level definitions are used primarily to associate drivers with peripherals and provide those drivers with important parameters such as the base address and interrupt, reset and clock lines of a peripheral. Those definitions are provided by the SoC module. The role of SoC level definitions is explained in more detail in Chapter 4.

board The platform board definition, which includes the complete list of peripheral as well as their name is provided by the board module. A master header file must be sourced by the application to include the correct board header file for the current platform. The role of board-level definitions is explained in more detail in Chapter 4.

clock The `clock` module is primarily used by the drivers and the application to query the clock frequency of various peripheral clock lines as well as control the SoC's clock gates if relevant.

reset The `reset` module is used to control the peripheral reset lines if they are present within a SoC.

int Control and handling of the SoC interrupts is handled by the interrupt management module, `int` for short. The `int` module allows registering of interrupt service routines(ISR), interrupt enabling and disabling as well as interrupt priority configuration. For more details see Chapter 10.

util The utility module provides miscellaneous utility and convenience macro mostly used internally by the BASEplatform. The `util` module also contains the list of errors return code within the `bp_rtnc.h` header file.

gpio The General Purpose Input Output module or `gpio` provides control over an MCU GPIO lines including reading/writing and direction control. Chapter 16 covers the `gpio` module in detail.

uart With the Universal Asynchronous Receiver Transmitter module an application can perform serial communication using UART or UART-like peripherals. The `uart` module contains all the usual functionalities such as baud rate and protocol control, read and write as well as synchronous and asynchronous, interrupt-driven operation. The `uart` module is covered in details in Chapter 13.

i2c The Inter-Integrated Circuit module (I2C) module is responsible for serial communication with board components such as sensors, EEPROM, power management IC and more. The I2C module API has the usual configuration API with control over the protocol bit rate and addressing mode. Like other serial communication modules of the BASEplatform the I2C module offers both a synchronous and asynchronous interrupt driven API. See Chapter 15 for additional details on the `i2c` module.

spi The Serial Peripheral Interface module enables serial communication with external SPI devices. The `spi` module API offers full control over the bit and protocol mode (clock and polarity) of the SPI bus. Optional control of the chip select line is also available. As usual for the BASEplatform both a synchronous and asynchronous interrupt driven API is available for communication. See Chapter 14 for additional details on the `spi` module.

qspi The Quad-SPI module is used primarily by the media module and storage media subsystem in order to interface with external non-volatile QSPI NOR and NAND flash memories. The `qspi` module can also be used standalone by an application if direct control over the QSPI bus communication is desired.

kernel The `kernel` module provides an internal kernel API abstraction for the other BASEplatform module. This allows the BASEplatform modules to work seamlessly with various real time kernels or in a bare-metal environment.

board_comp The board component module, `board_comp` for short, is a collection of modules for all sorts of external components such as non-volatile memories, sensors, I/O expanders, power management ICs, Ethernet PHYs and more.

soc_comp The SoC component module, `soc_comp` for short, is a collection of modules to handle most peripherals found on a System on Chip.

integ The integration module, `integ` for brevity, is used for the integration of the BASEplatform with third-party code such as RTOS, SDK and existing customer platform support code.

test The BASEplatform `test` module contains the test framework and tests of the BASEplatform.

mem The memory management module, `mem`, provides the primary memory allocator used by the BASEplatform modules when creating new module instances. Through a selection of multiple memory allocator drivers it is possible to allow or deny freeing of allocated memory as needed by the application. It is also possible to configure the BASEplatform to use the C standard heap through `malloc` and `free` instead of its internal heap. For more details see Chapter 6.

media The `media` module is used to provide a byte-level abstraction to various non-volatile memories such as NOR, NAND and SD/MMC flash memory. The media module can be used directly by the application for raw access to a storage media or through a file system such as the TREEspan File System (TSFS) or a third party file system.

3.3 Hardware Specific Modules

Hardware specific modules as their name suggest are specific to a piece of hardware, whether a SoC's peripheral or an external board component. Most hardware-specific modules are located under the SoC component or Board component modules and include drivers, implementations as well as other single instance modules designed to handle specialized peripherals. An example of a hardware-specific module is an I2C driver for a specific MCU. Another example would be an interface module to an external power management IC.

3.4 Integration Modules

Finally, integration modules are any module that is designed to integrate a third party component with the BASEplatform. They can vary wildly in shape and form but include implementation of kernel abstractions, glue code for manufacturer SDK and compatibility layers.

3.5 Conclusion

This chapter presented a brief overview of the most common modules that comprise the BASEplatform. Readers are encouraged to read the module's specific chapters as well as the API reference of each module for additional details.

Chapter

4 Structure of the BASEplatform

4.1 Introduction

The BASEplatform follows a highly modular design which is covered in detail within this chapter. Intimate knowledge of the content of this chapter is certainly not necessary in order to use the BASEplatform. However, readers who are interested in the organization and nomenclature of the various modules may want to read on. This chapter will go over the type of modules that can be found within the BASEplatform as well as how those modules interact to provide a complete platform support API.

Let's start with some important definitions.

Module The basic components of the BASEplaform are modules. Everything be it a file, header or API is contained within a module. Modules often provide a public API also known as an interface to be used by the application, such as the UART module. Modules may also provide an interface required by another module, for example a UART peripheral driver module provides a UART driver interface to be used by the UART module. Modules can be included or excluded from a project as needed by the application. Modules can also have dependencies, for example a peripheral driver module will probably require the interrupt management module to register and configure interrupt handlers for the driver.

SoC, CPU and MCU Isolated CPU are a rare sight in embedded system nowadays. The BASEplatform is designed to support a CPU along with any peripherals used within the System on Chip (SoC). As such the term CPU, microcontroller(MCU) and SoC are used interchangeably within the documentation. Unless specified otherwise, when used, the terms CPU, MCU or SoC all refer to the processing cores as well as any on-chip peripherals. Note that an isolated CPU core is simply referred to as a CPU core or core for short.

Driver Drivers allow top-level modules to access a SoC's peripheral. For example, an I2C driver is used by the I2C module to interact with an I2C peripheral. Drivers can also be used to interacting with board-level components such as a I/O expander driver allows the GPIO module to access an external I/O expander. The API provided by a driver is always unique allowing more than one peripheral driver of one type to be included within an application. For example, a SoC might have two different SPI peripherals requiring two different drivers.

Drivers are not usually accessed directly by the application, however, drivers can be instantiated alone without any top-level module. This can be useful to reduce the overhead associated with accessing a specific peripheral. However, thread safety is usually implemented by the top-level module which means that accessing a driver directly must be done with care to prevent race conditions.

Port Port files or modules are used to provide the necessary integration between the BASEplatform and the underlying CPU architecture. Port files related to the CPU core are usually located within the architecture module and may be comprised of low-level C code and hand-coded assembly. Port files are also used to provide the core entry function which is executed first by a CPU core and perform any low level and C environment initialization required by the platform. Finally, architecture port files contain the necessary compiler abstractions used by the other BASEplatform modules and application.

In addition to the signification described in the previous paragraph, a port may also refer to a kernel port when the BASEplatform is used with an RTOS.

Implementation Modules of which there can be only a single instance but that require a platform specific implementation, such as the interrupt management module, do not use a driver to provide the platform-specific functions. Instead the module's API is provided directly by the platform specific implementation. Those modules, simply called implementation, always implement the same API usually declared in the top-level module's header file. For example the interrupt management module header file, `bp_int.h` is located within the `int` module. While, for example again, the ARM Generic Interrupt Controller implementation is located within the `arm_gic` module. As a consequence there can only be one implementation of each individual type within an application. Implementations are selected at compile time by including them within the BASEplatform library or build.

Integration Integration modules are used by the BASEplatform to interoperate with third-party components. This can include RTOS, manufacturer SDK and customer platform code and other third-party components. Integration modules are loosely defined and do not have strict API since they are tailored to the integrated component.

Architecture, Board and SoC Level The BASEplatform attempts to maximize code reuse and interoperability primarily by structuring the various modules, dependencies and configuration across three levels or area.

Architecure Level The architecture encompasses the platform's CPU architecture along with the chosen toolchain. This means that most modules that are specific to a CPU architecture or to a specific compiler are architecture-level modules.

SoC Level The System on Chip (SoC) level includes the CPU core (also known as the CPU implementation) along with all the on-chip peripherals. One can find at the SoC level most peripheral drivers, memory map definitions, interrupt, clock and reset routing among other things.

Board Level The board level includes more external component as well as everything not included in the other two levels. This includes external board component drivers and definitions as well as board support package files. By extension, any application specified files and configurations are considered to be board-level components. In this context from the BASEplatform point of view, a ”board” can be seen as a complete platform.

Platform Definitions Platform definitions are used by the BASEplatform primarily to describe a platform's components. Information contained within a platform definition include associating peripheral drivers to peripherals, describing the routing of interrupts, clock and reset lines as well as describing the interconnection of external components. Definitions are divided in two broad categories, SoC level definitions and board-level definitions.

SoC Level Definitions SoC level definitions describe the current System on Chip, this includes the list of peripheral drivers as well as interrupt, clock and reset configuration. SoC level definitions can also include the cache topology and other characteristics. A complete set of SoC definition files are usually delivered with the BASEplatform and do not require any user modifications.

Board Level Definitions Board level definitions includes all the external board components as well as any application defined parameters. This can include external peripheral routing, pin configuration, external constraints such as maximum clock rate, etc.

4.2 Configuration Files

The BASEplatform uses a minimal set of compile-time configurations located in header files provided by the application. These configuration files are used to set a variety of important preprocessor macro definitions. There are usually a minimum of three configuration header files. The first one contains various preprocessor definitions that are applications defined at compile time. The two others are used to select the master architecture header file as well as the master board definition file. More information on these configuration files can be found in Chapter 8.

4.3 Platform Structure

A complete platform is comprised of multiple modules to support the CPU architecture, compiler, CPU core, SoC peripherals and any external board components. Figure 1 shows a diagram of a typical platform. Note that the diagram below is for a platform supported entirely by the BASEplatform, some modules may be omitted if the BASEplatform is integrated into an existing SDK or with existing platform support code.

The modules are divided in five broad categories detailed below. For more details on each module see Chapter 3 as well as the individual module documentation.

Architecture components At the core of the BASEplatform we find the architecture and compiler support modules. These modules are nearly always present and provide other modules as well as the application with interfaces to control the CPU core, cache, MMU and more. The architecture components also provide the CPU core or cores entry function and low-level initialization of the C environment. Critical sections as well as spinlocks are also located within this section.

SoC-level components Most of the BASEplatform modules are used to interface with the SoC's peripherals. The list of supported peripherals can be very diverse depending on the application needs but usually includes some core SoC support modules as well as a selection of communication peripheral modules. The core SoC support modules, to name a few, include the time and timer module and implementation, interrupt management, peripheral clock, reset control and more. Other ancillary SoC peripherals may be included as needed. Finally, most applications require at least one external communication peripheral such as a UART, I2C or SPI. Additional peripherals can also be included as needed such as QSPI, Ethernet and more.

Board level components Board components can be very diverse including external storage, memory I/O expander, power management ICs, sensors etc. Some drivers may also require board-level functions and configurations which are called BSP. However this is relatively rare.

Utility components. The BASEplatform also includes some utility components that don't involve peripheral support. These components are primarily used by other BASEplatform modules but can also

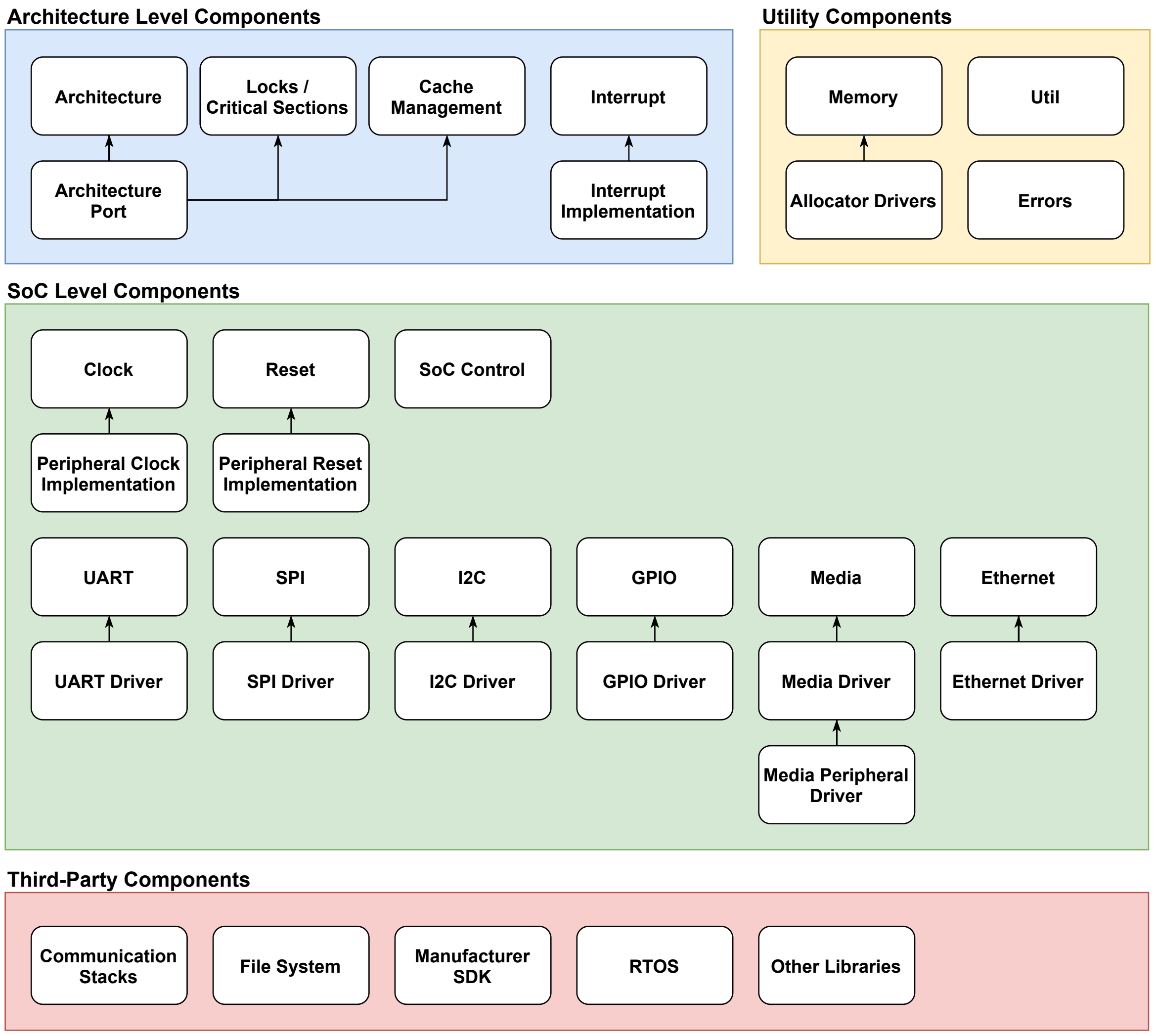

Figure 1 – Diagram of a typical platform.

be useful to the application. One of the most important of these utility component is the memory management module which provides the primary memory allocator. The primary memory allocator can be configured as per the application's requirement with a certain amount of memory and can allow or disallow the freeing of allocated memory. Other utility modules include buffer pools, utility macros, string and byte manipulation, as well as test and debug functions.

Third-Party Integration Finally, various third-party software components may be integrated with the BASEplatform using integration modules. The most common third party component is a real time kernel or RTOS, other third-party components can be very diverse such as file systems, Ethernet and USB communication stacks and many others. Existing SDK or platform support code can also be integrated if needed.

4.4 Simplified Platform Module Dependency Tree

A simplified platform dependency tree is shematised in Figure 2. Individual modules are abstracted but the three levels of architecture, SoC and Board components is kept as they are key to understanding the organization of the BASEplatform.

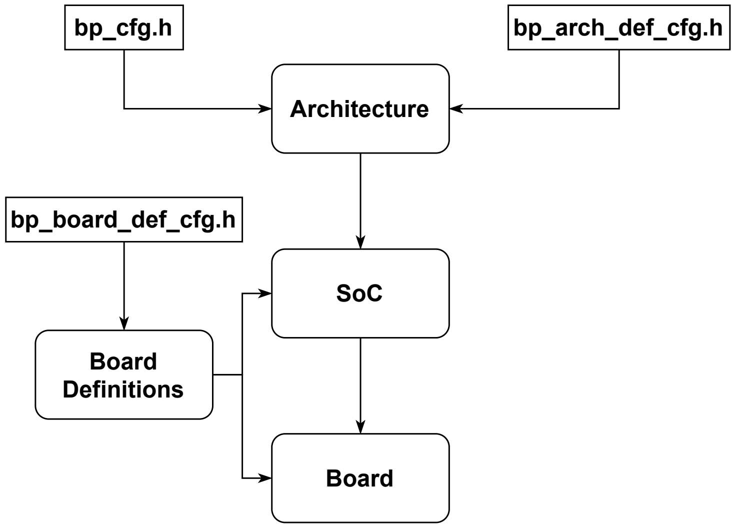

Figure 2 – Simplified platform dependency tree.

4.4.1 Architecture Level

At the lowest level of the dependency tree, we find the architecture-level components. The modules included thereof usually do not require any higher-level services from the SoC and board modules. However there can be exceptions to this general rule. For example, a portion of the cache hierarchy may be attached to the SoC instead of the CPU core. The most important of the architecture-level module is the architecture module itself, named ARCH for brevity. Most other modules included at the architecture level are sub-modules of the ARCH module.

As described earlier in this chapter, the BASEplatform has a global compile time configuration file, named `bp_cfg.h`. This configuration file is included from within the ARCH module and is considered a direct dependency of that module. Any other modules that require access to the definitions located within `bp_cfg.h` simply include the necessary header from the ARCH module instead of including the configuration file directly. This configuration file contains important user configurable parameters which must be available at compile time. See the Chapter 8 for details.

The ARCH module also requires an additional header file provided by the application named ”bp_arch_def_cfg.h”. This header file should contain a single include directive to the current architecture port master header file. This master header file allows the baseplatform to access important compiler and platform abstractions required at compile time. It also allows the BASEplatform to identify unequivocally the architecture and toolchain used by the application. This method of including the architecture port master header file allows each port header file to have a unique non-ambiguous name and remove the necessity of adding an include path entry to the architecture port.

4.4.2 SoC Level

The next level, the SoC level contains most of the peripheral modules. Modules at the SoC level usually have a primary dependency on the architecture-level components as well as various SoC-level modules. All of the SoC-level modules have a dependency on the SoC definition master header file which is included through the board definition master header file. The board master header file is included through an application supplied configuration header file `bp_board_def_cfg.h`. The SoC header file contains important definitions and declarations related to the current SoC. From the application point of view, the SoC header file is included indirectly by selecting the board header file. This is discussed in the next section. SoC-level modules, and especially most drivers, derive many important parameters such as a peripheral base address, interrupt clock and reset line from the SoC definitions. This information is passed to each driver when creating a new module instance at runtime. The SoC definition header file also contains important preprocessor macro definitions that must be available at compile time and are used by various modules, for example the total number of interest lines is often defined there.

4.4.3 Board Level

Finally, the board level completes the entire platform. The primary dependency of the board-level modules and definitions is the board definition master header file which is included through an application provided configuration header file `bp_board_def_cfg.h`. This header file should contain a single include directive to include the board definition file for the current platform.

Some drivers may require additional board-level functions which in this context form part of the board support package. Those BSP provides board specific functions and configurations that cannot be derived or configured from the generic SoC definitions and API. It is important to note that most drivers do not require any BSP component.

4.5 Typical Peripheral Stack Dependency Tree

Most peripherals are supported by a module stack containing a top-level module and a driver. An example for a UART peripheral is shown in figure Figure 3. The top-level module provides the lifecycle management functionalities, the driver agnostic top-level API as well as thread safety. Al the while, the driver is responsible for the interaction with the hardware. When creating a new peripheral module instance, I2C in this example, a board definition structure for the peripheral to initialize is passed to the module create function. The board definition contains the name to be given to the peripheral instance, any board-level configuration or BSP required as well as a link to the SoC definition for that peripheral. The top-level module uses the SoC definition to find the peripheral driver to be used when accessing this peripheral. The SoC definition also contains a set of driver specific definitions which are used by the driver to know the peripheral base address, clock, reset and interrupt lines as well as any other SoC level parameters required by the driver. Those definitions, especially the SoC definitions, are usually provided with the BASEplatform and do not have to be written from scratch by the application developer.

The example described in the previous paragraph forms the most basic and also the typical form of a complete stack of BASEplatform modules enabling the application to interact with a peripheral in a portable fashion. In general, peripheral module stacks are divided in a generic top-level module that provides all the necessary abstraction and a driver providing the hardware access functions. In addition, every driver can also expose additional driver specific API functions to access advanced and optional features of a hardware peripheral.

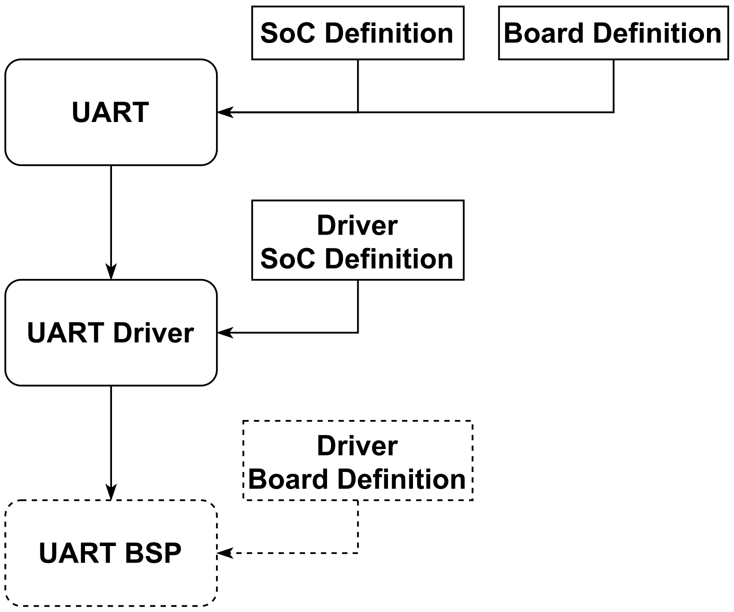

Figure 3 – Diagram of a typical peripheral stack.

4.6 Conclusion

As mentioned at the start of this chapter, it is not necessary to understand the structure of the BASEplatform to be able to use it. However a basic understanding of the organization can help. The three levels of organization, architecture, SoC and board forms the essence of how the BASEplatform is organized and how it enabled an unprecedented level of portability.

Chapter

5

Module Lifecycle

5.1 Introduction

Save for a few exceptions, most modules and drivers of the BASEplatform must be initialized in some way. This initialization usually happens in three steps in order to create, configure and finally enable a module. Trivial, low-level modules of which there can be only a single instance, such as an architecture utility module, may not require any initialization. While slightly more complex modules could require an initialization and configuration phase but when only a single instance of that module is possible, such as a SoC clock control module, it will have a single initialization function. This initialization function is usually suffixed `_init`, for example `bp_time_init()`, which must be called prior to using the `time` module. Finally, more complex, multi-instance modules, such as the UART module have a full suit of lifecycle management functions which control creation, destruction, enabling, configuration and much more. This chapter goes into details over the typical lifecycle of a BASEplatform module.

5.1.1 Multi-Instance Modules

Many modules of the BASEplatform allows the creation of multiple instances of themselves at runtime. An example of such a module is the UART module which requires one instance of the UART module to be created for each individual UART peripheral in a system. Multi-instance modules are created at runtime, for example using the `bp_uart_create()` function to create a new UART module instance. In addition, if the memory allocation policy allows freeing memory, multi-instance modules can be destroyed at runtime as well. For example `bp_uart_destroy()` can be called to reclaim the memory used by an existing UART module instance. See the Chapter 6 for more information about allocation policy.

5.1.2 Single-Instance Modules

For many modules it would not make sense to find more than one instance of those modules within an application. For example, the interrupt management module. These modules use a simpler lifecycle management API and only need to be initialized once at runtime, for example by calling `bp_int_init()` to initialize the interrupt management module. Usually, single instance modules cannot be destroyed at runtime even if the memory allocation policy allows the freeing of the module's memory. Also, single instance modules usually have a simpler API since it is not required to pass an instance handle to each API functions.

Additionally, some single instance modules may not have any internal state or need of any allocatable resources, in these cases it is not necessary to initialize those modules and the init function is omitted.

5.1.3 Exceptions

Some modules due to their complexity or by their design do not follow the generic lifecycle described in this chapter. Those modules usually require additional initialization steps tailored to the module in question. A good example of such an exception is the storage media module which can be used along with a file system and many different types of storage media. Other modules, especially third party RTOS integration modules, also require a tailor fitted initialization sequence and API which differ from the usual BASEplatform API. In all cases, readers should refer to the module's dedicated documentation for details.

5.2 Lifecycle Overview

Formally a module's lifecycle can be seen as a set of possible states along with various API to transition between those states. The range of possible states and transitions is schematized in Figure 4.

The possible states are as follows:

- *Created* - The module instance was just created or has been reset.
- *Configured* - The module instance was configured after being created but is not yet enabled.
- *Enabled* - The module instance is enabled and can be used to perform its intended function.
- *Disabled* - The module instance is disabled, it cannot be used unless it is enabled first.
- *Destroyed* - The module instance was destroyed and should no longer be used.
- *Error* - The module instance has encountered an unexpected condition.

Transition between those states is possible using a set of API functions which, for uniformity, usually share the same suffixes.

- `create()` - Creates a new module instance and place that instance in the created state.
- `cfg_set()` - Configures a module instance. If the module was in the created state, transition to the configured state otherwise, the configuration is simply updated.
- `en()` - Enables a module instance after which, if successful, the module instance is in the enabled state and can be used normally.
- `dis()` - Disables a module instance leaving it in a disabled state. Most of a module's API cannot be used on a disabled module other than `reset()` and `en()`.
- `reset()` - Resets a module instance from whichever state it's in back to the created state. This is the only call that should be performed on a module in the error state.
- `destroy()` - Destroys a module instance and, if permitted by the memory allocation policy, reclaims the memory and resources allocated to the instance.

To summarize the typical lifecycle, a module must be created, configured and then enabled using the module specific functions to perform those steps. Once enabled, the module can be used to perform its intended function such as sending and receiving bytes for a communication module. At any point in

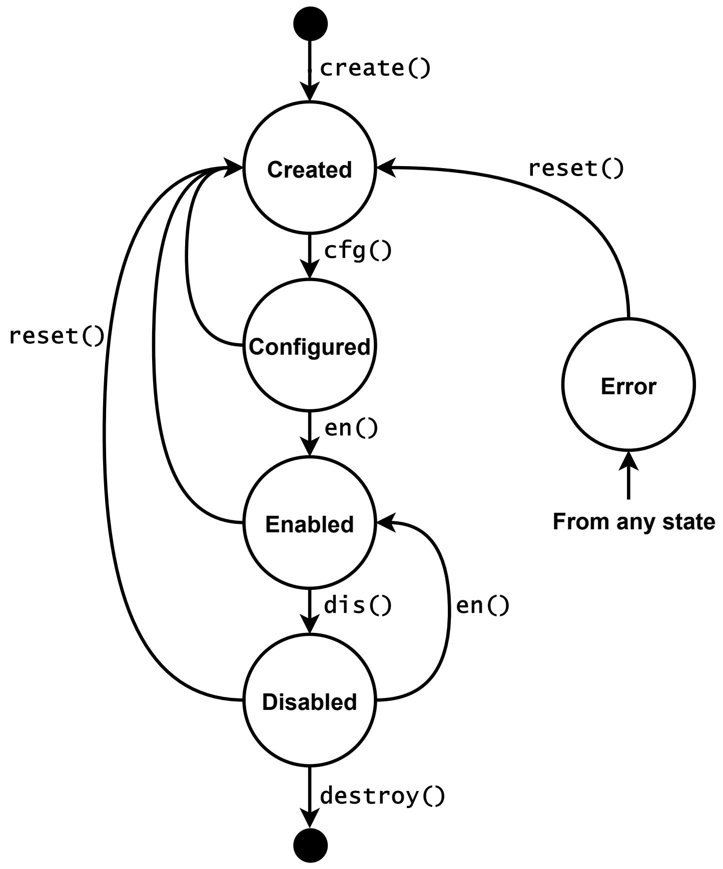

Figure 4 – Typical lifecycle of a BASEplatform module.

time, a variety of other lifecycle management functions can be called to disable, reset and destroy a module. Module instances within an application may transition multiple times between the various states as required by the application.

5.3 Lifecycle Details

5.3.1 Creating a Module

The first step in the life of a module instance is to create it. The function to do so for each module is usually suffixed by `_create`. For example `bp_uart_create()` is used to create a new UART module instance. Module creation functions typically take as an argument a pointer to a module handle through which the handle to the newly create module will be returned, if successful. Most create functions also take as an argument a pointer to a module's board definition structure, this structure contains all the necessary information required to associate the module instance with a physical peripheral. This information includes the peripheral name and driver as well as additional driver specific information like the peripheral base address, interrupt mapping and so on. Once a board definition is associated with a module instance, it should not be used again to create another module instance unless the previous module instance is destroyed first.

Unless stated otherwise, a module create function is the only function that can allocate resources such as memory and kernel objects. This means that once created successfully, a module should have all the resources required to operate for the lifetime of the application. If a module cannot acquire the resources needed upon creation, every attempt will be made to free any acquired resource to prevent a memory leak due to the creation failure. The ability to do so may depend on the memory allocation policy of the default memory allocator. As such, in an application where freeing memory is disallowed, all modules should be created early in the application lifetime and the memory made available to the BASEplatform should be sufficient for all the module instances required by the application. See Chapter 6 for more information on memory allocation policy.

5.3.2 Configuring a Module

Once created, a module should usually be configured. The exact configuration is module specific but can include, for example, the bit rate as well as protocol parameters. For example, the `bp_uart_cfg_set()` function of the UART module takes a structure as an argument to set the baud rate, parity and number of stop bits of a UART interface. Users should refer to a module's documentation for information on each module's configuration.

For easier portability and to keep the API concise, top-level modules such as the UART or SPI module have a set of standard configuration parameters such as the bit rate. The standard set of configuration is designed to accommodate most usage scenarios as well as most SoC peripherals. However, some peripheral drivers may not support all the possible configuration parameters. Moreover, simple peripherals with a fixed hardware configuration may work without being configured at all. However, for better portability it is recommended to always set a module's configuration before enabling it for the first time even if it's not required by the particular device driver used within an application.

As mentioned in the previous paragraph, the BASEplatform modules are designed to accommodate most usage scenarios and SoC peripherals. That being said the BASEplatform driver API enables the application to access driver specific configuration functions to access advanced, peripheral specific features. Readers interested in those features should read each driver's specific documentation for a list of driver-specific features.

5.3.3 Enabling a Module

After being created and configured a module should be enabled before being used. For example, using the UART module `bp_uart_en()` function. At this point the peripheral module is ready to be used. The exact effect of enabling a module instance is module and driver specific. In most cases this enables the module's clock and input/output pins. When relevant, reception through the peripheral is enabled a well, such as a UART receive path. Enabling an already enabled module should be without side effects.

5.3.4 Disabling a Module

A module instance in the enabled or configured state can be disabled. For example, using the UART module `bp_uart_dis()` function. No operations should be attempted on a disabled instance other than enabling or resetting the module instance. The exact side effects of disabling a module are driver specific and usually involve disabling the module clock and disabling the module's I/O pins to put the peripheral in a low power state. The peripheral configuration set prior to calling the module disable function will be retained or restored upon re-enabling the module instance. It is important to note that a module disable function as well as any other functions other than reset or enable should not be invoked on an already disabled instance. Doing so means the driver could potentially access a peripheral with an inactive clock which may cause either a bus fault or a bus hang. When assertion checks are enabled, a driver may return an `RTNC_FATAL` error code if an attempt is made to access a disabled peripheral.

5.3.5 Resetting a Module

A module in any state, other than destroyed, can be reset. Moreover, reset is the only action allowed on an instance that has returned an `RTNC_FATAL` error as described later in this chapter. A reset can be performed, for example, by calling An example is the UART module `bp_uart_reset()` function. Upon reset, a module will be returned to the created state and must subsequently be configured and enabled again to be functional. Note that resetting a module which is actively performing an operation should be avoided if possible as the result can be unpredictable. If a module reset is required as part of normal operation, it is recommended to first disable the module prior to performing the module reset.

In addition to resetting the internal state of the module and its driver, resetting a module will also reset the hardware peripheral if that operation is supported by the driver. The exact effects of resetting a peripheral is driver specific, but when available the soft reset feature of the peripheral is used by the driver. Additionally, if a centralized reset controller is available on the current SoC, the driver can toggle the reset line of the peripheral.

5.3.6 Destroying a Module

Depending on the allocation policy of the default allocator it may be possible to destroy a module. See for details on memory allocation. Again for example using the `uart` module, it is possible to destroy an instance by calling `bp_uart_destroy()`. Prior to destroying a module, it is recommended to disable it to make sure the destroy operation doesn't corrupt the peripheral's state. The instance's module handle becomes invalid upon successfully returning from the destroy function and should not be used again.

5.3.7 Transition and Recovery from an Error State

A module instance that encounters an unexpected condition, usually an assertion failure, will transition into the error state. If configured to do so, API functions that encounter a fatal condition will return the `RTNC_FATAL` status code to the application. As a general rule, a module where any of its API function returned `RTNC_FATAL` should not be used anymore as the internal state might be inconsistent. From the application point of view, it is recommended to handle a fatal error returned by any API function like any other unexpected failure. Usually either a platform reset or a transition into a safe state to prevent further safety degradation. In some cases, application designers may prefer to attempt resetting the module. This is particularly useful during testing where fault conditions might be injected or simulated.

5.4 Conclusion

The lifecyle of the various BASEplatform modules is central to the BASEplatform's versatility. An application may only require a few modules that are initialized and configured at boot time. All the while another more complex application may require a large variety of modules along with the need to dynamically create and destroy modules at runtime.

For the sake of clarity, the various BASEplatform modules attempt to follow the lifecycle described within this chapter. Namely the `create` - `enable` - `configure` - `disable` - `destroy` sequence of functions. More complex modules, however, may possess a unique initialization and configuration sequence which are documented within those modules' documentation.

Chapter

6 Memory Management

6.1 Introduction

This chapter gives an overview of the memory management module and associated topics related to memory management within the BASEplatform. Internally, the BASEplatform uses a memory allocator when creating new modules and objects. This allocator is provided by the memory management module, or `mem` for short. The application is also free to use the default allocator as well as create other instances of allocators for its own needs. Using the memory management module and a selection of different memory allocator drivers it is possible to change the behaviour of any allocator including the default one. It is, for example, possible to use a default allocator that prevents the freeing of memory while another application may prefer an allocation policy where the freeing of memory is allowed. It is also possible to map one allocator instance to the C library `malloc()` function to use the C heap instead of an internal heap. With the flexibility offered by the `mem` module, the application is able to select the features of the default memory allocator to suite the application's requirements and policies.

6.2 Overview

When creating new BASEplatform modules and objects the memory required for these objects is taken from a internal memory allocator. To increase the flexibility of the memory allocation within the BASEplatform that default memory allocator is provided by a module in itself, the memory management module or `mem` for brevity. The `mem` module can be used to instantiate one or more memory allocator and one of them must be set as the default allocator at the discretion of the application. When creating a new allocator, the application has a choice of different allocator drivers. Those different drivers allow the application to control the behaviour of any allocator it creates and consequently change the behaviour of the internal BASEplatform allocation. The application is also free to use the default allocator through a dedicated API or create new allocators for its own needs.

6.2.1 The Default Allocator

As said in the overview section, the BASEplatform needs an internal allocator when creating new modules or needs internal objects such as mutexes and semaphores. This default allocator should be created and set very early in the initialization sequence, otherwise any call that requires memory allocation will fail. When setting up the default allocator, the application designer should assign a sufficient amount of memory to it to serve the application's needs. Also a decision should be made if freeing allocated memory should be allowed or not. Once set, the default allocator cannot be changed. However it can be used by the application as well as the BASEplatform.

6.2.2 To Free or Not to Free

When selecting the default allocator, the most important choice is whether or not to allow the freeing of memory. This decision should be made according to the application's requirements. This choice affects the behaviour of the BASEplatform in a few ways. The first and most obvious is that destroying modules won't free the memory associated to them, effectively disabling the ability to destroy modules. Things are a little different with objects, however, such as semaphore, mutexes, timers and so on. With objects, when freeing is disallowed the objects are returned to a pool for those types of objects. For example, if a software timer is no longer needed and destroyed in that scenario it will be returned to a pool of timers. The next time a timer is allocated it will be taken from the pool instead of the heap.

6.2.3 Self-Contained Allocator

By default when creating an allocator the memory needed to hold the `mem` module metadata is taken from the default allocator. This is problematic when creating the default allocator itself as there is nothing to allocate memory from. That is why most allocator drivers have the option of being self-contained. When self-contained, the memory needed for the `mem` module instance is taken from the allocator being created instead of the default allocator.

6.2.4 Allocator Drivers

There are two important drivers available within the BASEplatform along with another more specialized driver.

Simple Heap The simplest of allocators, the simple heap or just heap allocator uses a contiguous region of memory that is progressively carved out each time a memory allocation is required. Freeing of memory is not possible when using the simple heap allocator. This results in the fastest allocation performance with practically no overhead to track the allocated and free memory. This allocator is useful for real-time and safety-critical applications that do not wish to perform any dynamic memory allocation. If used as the default allocator, it also means that once created, the BASEplatform modules cannot be destroyed.

C Heap Another simple allocator is one that uses the C heap. In this case both allocation and freeing of memory is possible and the memory used will come from whatever amount of memory is allocated to the C heap. Although not absolutely necessary, it is recommended to only create one allocator using the C heap. When using the C heap care should be taken when working in a multithreaded RTOS as the C malloc and free functions are not always thread-safe. The BASEplatform assumes that its internal allocator is thread-safe when functioning under an RTOS.

Stack The stack allocator is a specialized allocator that supports freeing but only in the reverse order in which the memory was allocated. Because of this feature it cannot be used as a general-purpose allocator nor can it be used for the default allocator. It does have the advantage of very fast allocation and deallocation with practically no overhead.

6.3 Usage

This section goes over the usage of the mem module including how to create an allocator, set the default allocator and how to allocate and free memory.

6.3.1 Create an Allocator

Whether it's for the application's own usage or to be used as the default allocator, the first step in the life of an allocator is to create it. Creating a new memory allocator instance follows a slightly different convention compared to most BASEplatform modules. In order not to burden the create API to make it generic enough to cater to the needs of all the types of allocator drivers, there is no generic create function in the mem module. Instead a driver create function should be used which will return an allocator handle. After having created an allocator the generic memory allocation API can be used with the newly created instance.

For example, to create a new allocator using the heap driver one would use the function as shown in Listing 2. When creating the allocator it is necessary to include both the mem module header file bp_mem.h as well as the desired allocator driver's header, bp_mem_alloc_heap_drv.h in this example. When simply using the created allocator the driver header file is not required.

```
#include <mem/bp_mem.h>
#include <mem/allocator/heap/bp_mem_alloc_heap_drv.h>

int rtn;

// Memory area to be managed by the allocator.
uint8_t memory[16u*1024u];

// Variable that will hold the newly created instance handle.
bp_mem_alloc_hndl_t alloc_hndl;

// Simple heap allocator definition structure.
bp_mem_alloc_heap_def_t alloc_def;

alloc_def.p_name = "main allocator";
alloc_def.p_base_addr = memory;
alloc_def.size = sizeof(memory);
alloc_def.self_contained = true;

rtn = bp_mem_alloc_heap_create(&alloc_def, &alloc_hndl);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 1 – Create a new heap allocator.

6.3.2 Set the Default Allocator

Setting the default allocator should be done very early in the initialization sequence using bp_mem_alloc_dflt_set(). Once set, the default allocator cannot be changed and the assigned allocator should not be destroyed either. Listing 2 shows an example of setting the default allocator from an existing allocator handle.

```c
#include <mem/bp_mem.h>

int rtn;

rtn = bp_mem_alloc_dflt_set(alloc_hndl);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 2 – Create a new heap allocator.

6.3.3 Allocate and Free Memory From the Default Allocator

Allocating and freeing of memory from the default allocator can be done using `bp_mem_alloc()` and `bp_mem_free()` as depicted in Listing 3. Due to the awkwardness of using a double void pointer `bp_mem_alloc()` and `bp_mem_alloc_from()` returns the status code in a variable passed by reference as one of the arguments. Thus the allocated memory can be returned through the function return value. The convention or returning NULL if memory allocation fails is followed here. The status code can be inspected to see the reason for the failure.

As an additional feature over the traditional C `malloc()` function, `bp_mem_alloc()` and `bp_mem_alloc_from()` take an additional argument to specify the minimum alignment for the allocated memory. Passing a value of 0 will use the platform's default alignment which is usually the largest possible natural alignment of any datatype.

```c
#include <mem/bp_mem.h>

int rtn;

// Variable to hold the allocated memory pointer.
void *p_mem;

// Allocate 16 bytes.
p_mem = bp_mem_alloc(16u, 0u, &rtn);
if(rtn != RTNC_SUCCESS) { /* Error management */ }

// Free the previously allocated 16 bytes.
rtn = bp_mem_free(p_mem);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 3 – Allocating 16 bytes from the default allocator.

6.3.4 Allocate and Free Memory From a Specific Allocator

Allocating from a specific allocator other than the default allocator is very similar to the previous example but using `bp_mem_alloc_from()` and `bp_mem_free_from()` as shown in Listing 4. The main difference is the necessity to pass the allocator's handle as argument. When using this method, it is important to return the memory to the same allocator it was allocated from. If assertion checks are enabled (See Chapter 7) some allocators can report that the memory pointer is out of range but that is not the case for all allocators.

```
#include <mem/bp_mem.h>

int rtn;

// Variable to hold the allocated memory pointer.
void *p_mem;

// Allocate 16 bytes from alloc_hndl.
p_mem = bp_mem_alloc_from(alloc_hndl, 16u, 0u, &rtn);
if(rtn != RTNC_SUCCESS) { /* Error management */ }

// Free 16 bytes to alloc_hndl.
rtn = bp_mem_free_from(alloc_hndl, p_mem);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 4 – Allocate 16 bytes from a specific allocator.

6.3.5 Query the Remaining Memory

It is possible, for some allocators, to query the remaining amount of memory by calling . If supported the remaining amount of memory will be returned. Otherwise `RTNC_NOT_SUPPORTED` is returned as the status code and the `p_rem_sz` argument is set to 0. Note that depending on the alignment requirement of future allocations and internal fragmentation of the free memory, it may not be possible to use all of the remaining memory.

```
#include <mem/bp_mem.h>

int rtn;

// Variable that will receive the remaining amount of memory.
size_t rem_sz;

p_mem = bp_mem_rem_sz_get(alloc_hndl, &rem_sz);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 5 – Query the remaining memory.

6.4 Conclusion

The memory management module is extensively used internally by the BASEplatform but is also available for the convenience of the application. The ability to select from multiple allocation drivers allows the application developer to fine-tune the behaviour of memory allocation and, critically, decide if freeing memory is allowed within the application. Readers interested in individual memory allocator drivers should consult the API Reference Manual for additional details.

Chapter

7 Error Handling

7.1 Introduction

Handling of statuses and errors returned by lower-level API functions is indispensable for a solid and reliable application. To simplify development and improve robustness the BASEplatform follows a consistent and simple error handling scheme. This scheme consists of two important aspects. The first being how errors are propagated through the call stack and reported to the application. The second is the assertion checks present throughout the code and the action taken upon triggering an assertion failure. To maximize the versatility of the BASEplatform error handling the application can control the presence of assertion check within the BASEplatform and customize the action taken when a failure is detected. The chapter also goes over recommended practices when it comes to handling errors returned by the BASEplatform API.

7.2 Overview

Within the BASEplatform the result of an API call is most often than not returned to the caller as the function's return value. This return value can inform the application of the status of the operation be it a success, failure or a specific status such as a value not found or a timeout. Those return values are called returned code. Internally, when an API receives a return code that cannot be handled at the current level the usual action is to pass it up the call stack up to the application. At that point the application can take appropriate action.

The BASEplatform uses a large quantity of configurable assertion checks to catch invalid arguments, data corruption and internal errors. Those assertion checks are designed to be usable in both debugging and production. They can, however, be disabled by the application developer to improve performance and reduce code size. When the checks are enabled, it is also possible to configure the action taken when encountering an assertion failure. By default the BASEplatform will return an `RTNC_FATAL` return code when reaching an assertion failure. Through a dedicated configuration, the application is able to override this behaviour and perform a different action such as breaking to the debugger when developing or triggering a CPU reset or a transition to a safe state when an assertion failure is detected.

The BASEplatform attempts to prevent hiding errors by returning them up the call stack as much as possible. At the same time, it tries to minimize error renaming or translation when necessary to prevent obscuring the initial cause of an error. Those last facts combined with the configurable asserts enable an application to control how errors are handled depending on the application's requirements. For example, one application may prefer to perform minimal error handling and management by disabling assertion checks to maximize performance and minimize code space. Another safety critical application may prefer to trigger a watchdog reset upon detection of a unrecoverable error to prevent further safety degradation.

7.3 Return Codes

Save for a few exceptions as well as functions that cannot fail, all of the BASEplatform API functions return a status code as the function's return value. This return code is under the form of a plain C `int`. A simple example of this is shown in Listing 6 where the return value of `bp_timer_init()` is checked to make sure that the initialization of the timer module was successful.

```
#include <util/bp_rtnc.h>
#include <timer/bp_timer.h>

// Variable to hold the return code.
int rtn;

rtn = bp_timer_init();
if (rtn != RTNC_SUCCESS) { /* Handle unexpected error here */ }
```

Listing 6 – Simple check of a function's return code.

All of the BASEplatform return codes are prefixed with `RTNC_` followed by a descriptive name for the status or error. The return codes are globally defined in the `util/bp_rtnc.h` header file. Listing 7 displays a portion of the `bp_rtnc.h` header file with some of the most common return code.

```
#define RTNC_SUCCESS             (0) /* Function completed successfully. */
#define RTNC_FATAL              (-1) /* Fatal error occurred. */
#define RTNC_NO_RESOURCE        (-2) /* Resource allocation failure. */
#define RTNC_IO_ERR             (-4) /* Transfer or peripheral operation failed. */
#define RTNC_TIMEOUT            (-5) /* Function timed out. */
#define RTNC_NOT_SUPPORTED      (-6) /* Operation not supported. */
#define RTNC_NOT_FOUND          (-7) /* Requested object not found. */
#define RTNC_ALREADY_EXIST      (-8) /* Object already created or allocated. */
#define RTNC_ABORT              (-9) /* Operation aborted by software. */
#define RTNC_INVALID_OP         (-10) /* Invalid operation. */
```

Listing 7 – Sample of error codes.

The application is advised not to rely on the exact numerical value of each error code as they may change in the future. In other words, the preprocessor definitions should always be used when comparing return codes. It is, however, guaranteed that `RTNC_SUCCESS` is 0 and that all other codes have negative value.

7.4 Fatal Errors

Errors can be classified either as fatal or nonfatal. An example of non-fatal error would be a UART interface timing out waiting for data to be received. In this case `RTNC_TIMEOUT` would be returned. While it may be a problem for the application, the timeout doesn't signify that anything wrong or unexpected happen within the BASEplatform or the underlying driver and peripheral. A fatal error, however, most often reported using the `RTNC_FATAL` return code means that an unrecoverable error was encountered. Most often than not the error is the result of a failed assertion check, which could mean an invalid argument, a corrupted internal structure or an unexpected condition. What to do when faced with a fatal error is application specific but should be treated like any other unrecoverable error. At a minimum when a fatal error is encountered the module or module instance that returned the error should not be accessed again. Some modules provide a reset functionality which will attempt to return the module into a known state.

In addition to the `RTNC_FATAL` return code, any unexpected return code should be treated as a fatal error. The list of possible return code of each API function is documented in the API reference manual.

7.5 Assertion Checks

Internally the BASEplatform uses a generous quantity of assertion checks to verify function arguments as well as internal state. Those checks can be enabled or disabled at compile time using configuration defines in `bp_cfg.h`. The application can set `BP_CFG_ARG_CHECK` to 1 to enable argument checking and `BP_CFG_ASSERT_CHECK` to 1 to enable other internal assertion checks. Setting either of them to 0 will disable that class of checks thus reducing code size and improving performance. The assertion checks are designed to be usable both during development and in production.

By default the result of encountering an assertion failure is the immediate return from the function that triggered the error with an `RTNC_FATAL` error. The application, however, can override this behaviour by defining `BP_ASSERT_ACTION()` in `bp_cfg.h`. A common configuration to use would be to call `BP_ARCH_PANIC` which usually disables interrupts and either go into an infinite loop or break to the debugger if possible. By breaking automatically when debugging, application developers can see quickly and painlessly the cause and location of an error, especially those caused by invalid arguments. Listing 7 shows an example of configuration useful when debugging.

```
// Enable argument checking.
#define BP_CFG_ARG_CHECK 1

// Enable internal error checking.
#define BP_CFG_ASSERT_CHECK 1

// Break to debugger or in a loop on failures.
#define BP_ASSERT_ACTION() BP_ARCH_PANIC()
```

Listing 8 – Example of assert configuration for debugging.

7.6 Recommended Method for Handling Return Codes

When checking and handling return codes, the recommended method is to first check for `RTNC_SUCCESS` then handle any expected return codes such as `RTNC_TIMEOUT`. Afterward all other return codes that are not expected or handled by the application should be considered fatal as shown in Listing 9. In the example the application calls the UART receive function `bp_uart_rx()` with a timeout value of 100 milliseconds. If the 8 bytes requested are received before the 100 millisecond timeout is reached then the function will return `RTNC_SUCCESS`. If, however, the 8 bytes are not received in time then the function will return `RTNC_TIMEOUT` which can be handled by the application. In a third scenario, let's say that the buf variable is NULL, then the function will return `RTNC_FATAL` to error.

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

// Variable to hold the return code.
int rtn;

rtn = bp_uart_rx(hndl, buf, 8u, &rx_len, 100u);
if(rtn != RTNC_SUCCESS) {
    if(rtn == RTNC_TIMEOUT) {
        /* Handle timeout condition. */
    } else {
        /* Handle fatal error. */
    }
}
```

Listing 9 – Example of handling return codes.

7.7 Conclusion

The BASEplatform error handling methodology is both designed to be simple and robust. The consistent way of returning status information to the application reduces programming errors and improve development speed. In addition using configurable asserts and assert action, it is possible for the application designer to tweak the behaviour of the BASEplatform in the face of unexpected and unrecoverable errors. Readers are encouraged to look the API Reference Manual for the API functions that are used within their application to see the list of return codes that can be returned by those functions as well as the condition that cause those codes to be returned.

Chapter

8

Configuration

8.1 Introduction

One of the design considerations of the BASEplatform is to limit the number of compile-time configurations. Most of the configurability of the BASEplatform comes from its modularity and flexibility. Modules and peripheral drivers primarily use definition structures that can be populated at runtime for setup instead of hard-coded compile time values. Additional information about those definition structures can be found in Chapter 4. However, it is hard to design a cross-platform SDK without some global compile time configurations. This chapter will go over those compile time configurations.

8.2 Overview

The BASEplatform uses, at a minimum, three configuration header files. The first one, `bp_cfg.h` contains various user-defined preprocessor macros, also known as configuration defines that can affect various aspects of the BASEplatform. The second file, `bp_arch_def_cfg.h`, should contain a single include directive to include the architecture port master header file specific to the current CPU architecture and compiler in use. The third file, `bp_board_def_cfg.h`, should also contain a single include directive for the master board definition file. The architecture port and board definition header files are used by the BASEplatform to know at compile-time various important aspects of the current platform.

8.2.1 Rationale

The design of the BASEplatform configuration files is aimed at improving performance and the simplicity of using the BASEplatform. They are also designed to increase the flexibility of the SDK as well as facilitating the usage of the SDK as a library. Finally, they are designed to ease the setup of a build project whether inside an IDE or built from a Makefile.

The aims described in the last paragraphs translate into three concrete design goals of the BASEplatform described below.

Improve performance. Using preprocessor definitions instead of run-time variables can help improve performance in some areas. For example, having a run-time configuration to enable or disable assertion checking would increase the code size considerably while adding overhead to each API calls.

Reduce the number of include paths. The BASEplatform file and directory structure is designed to reduce the number of include paths that need to be defined by the user when building their project. By including the board definition file as well as the architecture port file from the configuration files it becomes unnecessary to add their path to each IDE or Makefile project.

Prevent ambiguous file names and simplify package composition. Most RTOS or third-party libraries will use static names for port header files, which means that in a release package multiple files could end up having the same name. This becomes confusing to browse and also can confuse some build systems. Other than in rare exceptions, the BASEplatform attempts to never create files of the same name. This also means that multiple libraries for multiple targets can be built easily from a single cross-platform release package.

8.3 Configuration Files

At a minimum the BASEplatform requires three different configuration files to be sourced by the application. A suitable example of each one is supplied with the demonstration and development project distributed with the BASEplatform release package. Some platforms or third-party software integrated within the BASEplatform may require additional configuration files.

8.3.1 bp_cfg.h

`bp_cfg.h` is the primary configuration file and contains various user-defined preprocessor macros. Some of them, listed below, are mandatory for all platforms while other platform-specific configuration values may be needed or added by the developer.

BP_CFG_ARG_CHECK — Can be set to 1 to enable argument checking. If an invalid argument is detected by an API function `RTNC_FATAL` will be returned. See Chapter 7 for additional details on error handling.

BP_CFG_ASSERT_CHECK — Can be set to 1 to enable internal assertion checking. If possible `RTNC_FATAL` will be returned by an API function if an internal error is detected. See Chapter 7 for additional details on error handling.

BP_ASSERT_ACTION() — If set this macro is invoked on an assertion failure including invalid arguments and internal errors. See Chapter 7 for additional details on error handling.

BP_CFG_SMP — Must be set to 1 when running in an SMP configuration. This will change the behaviour of critical sections and spinlocks to be compatible with SMP platforms. This will also affect some peripheral drivers which may use additional synchronization constructs within their interrupt handler to prevent race conditions on SMP platforms.

BP_CFG_CORE_CNT — Number of cores when running in an SMP configuration otherwise must be set to 1. Usually this configuration would be set to a value higher than one when BP_CFG_SMP is set to 1. However for testing and development purpose it is possible to set the core count to 1 but with SMP enabled, this can be used to debug SMP-related issues while keeping the SMP compatible behaviour of locks and interrupts.

8.3.2 bp_arch_def_cfg.h

`bp_arch_def_cfg.h` is used to include the architecture port master header file. It should contain a single include directive for the architecture port file for the desired platform and compiler. Listing 10 shows an example of configuration file for the Cortex-M and GCC compiler. Usually the root of the BASEplatform source is part of the include path, as such it is recommended to include the architecture port file as shown in the example, starting from the `arch` module directory.

```
// bp_arch_def_cfg.h

#include <arch/port/arm-v7ar/gcc/bp_arm-v7ar_gcc_arch.h>
```

Listing 10 – Example of bp_arch_def_cfg.h for the Cortex-M GCC port.

8.3.3 bp_board_def_cfg.h

`bp_board_def_cfg.h` is used to include the board definition master header file. Like `bp_arch_def_cfg.h` it should contain a single include directive to for the current board definition header file. Listing 11 shows an example of board configuration file for the STM32F4 Discovery development board.

```
// bp_board_def_cfg.h

#include <board/stmicro/stm32f4discovery/bp_stm32f4discovery_board_def.h>
```

Listing 11 – Example of bp_board_def_cfg.h for the STM32F4 Discovery.

8.4 Conclusion

This is it for the compile-time configuration of the BASEplatform. As stated in the introduction, every attempt has been made to reduce the compile time configuration to a minimum. Readers should note that a complete and functional set of configuration files are provided with each release package. As such, developers should not have to play with the configuration too often with the exception maybe of the assertion checks configuration which are described in more details in Chapter 7.

Chapter

9

Time and Timer

9.1 Introduction

The `time` module handles the BASEplatform primary timebase to provide time measurement and time delay services. Its companion module, the `timer` module offers interrupt based software timers. Both modules, when paired with a high-resolution timebase, enables low-jitter sub-millisecond time measurements and delays. This chapter will go over the recommended specifications of the primary timebase, initialization of the `time` and `timer` modules as well as basic usage of both with code examples.

9.2 Overview

The `time` and `timer` modules are used both internally by the BASEplatform and the application. Internally, time delays and timers are used for the management of timeout and handle periodic events. The drivers also make use of sub-millisecond delays from time to time when managing peripherals with strict timing constraints.

9.2.1 The Time Module

As mentioned earlier in the introduction the `time` module handles the primary timebase. Using that timebase the `time` module can derive high-resolution time measurement and delays. The implementation of the `time` module is designed to minimize jitter and prevent drift when used with an appropriate hardware timer. In addition, the time module is based on a 64-bit time and as such can be considered to never wraparound. Both 32 and 64 bit versions of the time measurement and delay API are provided for convenience.

9.2.2 The Timer Module

The `timer` module derives its timebase from the `time` module to provide the application with interrupt based software timers. The resolution of the timers is the same as the primary timebase with the possibility of microsecond resolution timers if the selected timebase can handle it. Like the `time` module, the software timers are based on a 64-bit counter. Moreover the timer implementation on most platforms that supports it doesn't use a periodic tick which reduces CPU usage and improve power efficiency by minimizing the number of wake-up events in a low power design.

9.3 Primary Timebase

The quality of the services provided by the `time` and `timer` modules depends directly on the hardware timer used as the primary timebase. While there are no strict requirements, it is recommended to use a sub-millisecond resolution free-running timer with compare-and-match capabilities. Moreover, to improve performance a 64-bit timer is recommended if available. The use of a free-running timer with compare-and-match helps prevent drift as the timer won't need to be reset or reconfigured once stared. The use of a free-running timer with comparing and match can also improve performance slightly.

9.4 Initialization

Initialization of the `time` and `timer` modules should be performed early in the application lifetime. Also, it is necessary to initialize the `time` module before the `timer` module. Initialization can be performed by calling `bp_time_init()` and `bp_timer_init()` as shown in Listing 12. Neither function takes any arguments.

```
#include <time/bp_time.h>
#include <timer/bp_timer.h>

int rtn;

// Initializes the time module.
rtn = bp_time_init();
if (rtn != RTNC_SUCCESS) { /* Error handling */ }

// Initializes the timer module.
rtn = bp_timer_init();
if (rtn != RTNC_SUCCESS) { /* Error handling */ }
```

Listing 12 – Initialization of the Time and Timer modules.

9.5 Time Module Usage

The `time` module API can be divided in three categories. The first category deals with time measurement and allows querying the value of the primary timebase in a variety of units including retrieving the raw timer value. The second category includes all the time delay functions. The time delay functions are available in three flavours with the option of spin wait, interrupt based and hybrid delays. Finally, the last category contains functions to convert a time value between the raw timebase unit and various other units.

9.5.1 Querying the Time

Time measurement can be performed by querying the current value of the primary timebase. For example `bp_time_get_ms()` can be called to get the current value in milliseconds. An example of such a call is depicted in Listing 13. Note that a large portion of the `time` module API doesn't return a status code and as such will use the return value for other purposes, such as reporting the current time in this example. It is assumed that functions that don't return an error cannot fail.

```
#include <time/bp_time.h>

// Variable to hold the queried time.
uint64_t time_ms;

// Query the current value of the primary timebase in milliseconds.
time_ms = bp_time_get_ms();
```

Listing 13 – Querying the current time in ms.

It is also quite possible to query the raw timebase value by using `bp_time_get()` instead of `bp_time_get_ms()`. This is useful to get the highest possible resolution when performing time measurement. An example is presented in Listing 14.

```
#include <time/bp_time.h>

// Variable to hold the queried time.
uint64_t time;

// Query the raw value of the primary timebase.
time = bp_time_get();
```

Listing 14 – Querying the current time in raw units.

Both API functions used in the previous two examples return a 64-bit value. If a 32-bit value is desired, instead one could use a cast to discard the upper 32 bits of the result. However all of the timer query API that returns a 64-bit value have a 32-bit counterpart for convenience, such as in Listing 15. On top of being convenient, using the 32-bit version may, on some platform, offers a slight performance advantage.

```
#include <time/bp_time.h>

// Variable to hold the queried time.
uint32_t time;

// Query the least significant 32 bits of the primary timebase current value.
time = bp_time_get32();
```

Listing 15 – Querying the current time as a 32-bit value.

9.5.2 Getting the Timebase Frequency

When using the raw timebase values as shown in the previous section, it can be useful to query the frequency of that timebase. It can be done easily using `bp_time_freq_get()` as shown in . Note that `bp_time_freq_get()` always return a 64-bit value.

```
#include <time/bp_time.h>

// Variable to hold the queried frequency.
uint64_t freq;

// Get the frequency of the primary timebase.
freq = bp_time_freq_get();
```

Listing 16 – Querying primary timebase frequency.

9.5.3 Delays

An application may sometimes wish to delay execution in order to control the timing of various operations. This can be achieved using the various delay API within the `time` module. Listing 17 shows the simplest way of performing a delay by calling `bp_time_sleep_ms()`. All of the delay functions of the BASEplatform are guaranteed to have waited for the entire length of time specified.

`bp_time_sleep()` and `bp_time_sleep_ms()` can perform a busy-wait or an interrupt based delay depending on the length of the delay specified. For short delays it is often more efficient to wait in a loop compared to the time it takes to setup a wake up from an interrupt and perform a full context switch. For longer delays, those functions will wait on an interrupt allowing other tasks to run in a multithreaded system.

```
#include <time/bp_time.h>

// Sleep for 100 milliseconds.
bp_time_sleep_ms(100u);
```

Listing 17 – Simple time delay in milliseconds.

Like the time measurement API, it is also possible to specify delays in the raw timebase unit, for example by calling `bp_time_sleep()` instead of `bp_time_sleep_ms()`. Application developers should be careful when using this feature as the code is less portable if the firmware is migrated to a different platform with a different timebase frequency. Listing 18 shows an example of specifying a delay in the raw timebase unit instead of milliseconds. Again `bp_time_freq_get()` can be called to get the timebse frequency.

```
#include <time/bp_time.h>

// Sleep for 1200 count of the raw timebase.
bp_time_sleep(1200u);
```

Listing 18 – Time delay in the raw timebase unit.

In a similar fashion as the time measurement API some time delay functions are available in both 32 and 64 bit variants. For example Listing 19 contains the 32-bit version of the previous example.

```
#include <time/bp_time.h>

// Sleep for 1200 count of the raw timebase.
bp_time_sleep32(1200u);
```

Listing 19 – 32-bit time delay.

9.5.4 Busy-Wait and Interrupt Based Delays

Sometimes it is preferable to use one kind of delay over another. For example, when performing a short delay within an interrupt to comply with a peripheral timing restrictions it is important to use a busy-wait loop. In that situation it is indicated to use `bp_time_sleep_busy()` or `bp_time_sleep_busy_ms()` over the plain version to ensure that a busy-wait loop will be used such as in Listing 20.

```
#include <time/bp_time.h>

// Sleep for 100 milliseconds in a loop.
bp_time_sleep_busy_ms(100u);
```

Listing 20 – Time delay using a busy-wait loop.

In contrast with the last example, one may prefer to force a context switch or at least an interrupt to happen when delaying, even for a short time. In a multitasking environment, this is similar to a yield. This can be achieved using `bp_time_sleep_yield()` or `bp_time_sleep_yield_ms()` like Listing 21. When running with a kernel, this is guaranteed to force a context switch, while in a bare-metal environment it will cause an interrupt to be generated.

```
#include <time/bp_time.h>

// Sleep for 100 milliseconds.
bp_time_sleep_yield_ms(100u);
```

Listing 21 – Time delay using a yielding or interrupt based delay.

9.6 Timer Module Usage

When an application desires to perform an action at some point in time in the future, delays are not always the best option. For one in a multithreaded environment, it monopolizes an entire thread and in a bare-metal environment it prevents further code execution until the delay has passed or requires tricky timing through the super loop. In those cases, software timers are a convenient low-overhead alternative.

In the BASEplatform, software timers are always interrupt based and usually tickless, meaning that timer processing only happens when a timer expires. On expiry, a user-specified callback is called by the timer processing interrupt. Within this callback the application can perform any desired action and also optionally restart the timer to create a periodic timer. To use a timer an application must first create it, then start it configuring the timer parameters. Later on, an application can either restart a timer, stop it or finally destroy it when it's no longer needed. When destroyed, the memory used by the timer is either freed, if that is supported by the default allocator (See Chapter 6), or returned to a pool of unused timers, ready to be reused the next time a timer is created.

9.6.1 Creating and Destroying a Timer

The first step in order to use a software timer is to create it, using `bp_timer_create()`. It can be destroyed later if no longer needed with `bp_timer_destroy()`. Listing 22 and Listing 23 show examples of creating and destroying a timer.

```c
#include <timer/bp_timer.h>

int rtn;

// Variable to receive the created timer's handle.
bp_timer_hndl_t hndl;

// Create a new timer storing the timer handle in 'hndl'.
rtn = bp_timer_create(&hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 22 – Creating a timer.

```c
#include <timer/bp_timer.h>

int rtn;

// Destroy a previously created timer.
rtn = bp_timer_destroy(hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 23 – Destroying a timer.

9.6.2 Timer Callback

Every timer requires a callback before starting it. Listing 24 contains a simple timer callback function. The timer callback is passed the timer handle and a user-provided pointer as arguments. The timer handle can be used to modify the timer from within the callback, for example by changing the period. The second argument is the same one passed to the timer start function for use by the application.

The return value of the callback function can be used to instruct the timer management interrupt of the action to perform on that timer. The possible actions are BP_TIMER_STOP to stop the timer and thus deactivate it, BP_TIMER_RESTART to restart the timer using new settings, or BP_TIMER_PERIODIC to restart the timer from the last time it expired without changing the period.

```
#include <timer/bp_timer.h>

bp_timer_action_t timer_callback(bp_timer_hndl_t hndl, void *p_arg) {

    // Perform callback actions here.

    // Return the stop instruction to the timer handling routine.
    return BP_TIMER_STOP;
}
```

Listing 24 – Simple timer callback.

9.6.3 Starting a Timer

Starting a timer can be done using bp_timer_start_ms() or one of its variants. As displayed in Listing 25, bp_timer_start_ms() takes as argument the timer handle, the timer period as well as the callback function and a user-specified pointer which will be passed unmodified to the callback when the timer expires. If the callback of Listing 24 is used, this will be a one-shot timer.

```
#include <timer/bp_timer.h>

int rtn;

// Start a timer with a period of 250 milliseconds.
rtn = bp_timer_start_ms(hndl, 250u, timer_callback, NULL);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 25 – Starting a timer.

9.6.4 Periodic Timer

To turn the previous example into a periodic timer one only needs to change the callback to return BP_TIMER_PERIODIC as in Listing 26. Starting the timer is done in the exact same way as shown in Listing 27. When returning BP_TIMER_PERIODIC from the callback the timer handling interrupt will reschedule the timer using the initial period specified. Moreover when rescheduling the timer the target expiry time will be calculated from the time the timer last expired not the current time. This way, the timer will not accumulate any drift due to the latency between the timer expiring and the time it's processed and put back into the active timer list.

```
#include <timer/bp_timer.h>

bp_timer_action_t perioric_timer_callback(bp_timer_hndl_t hndl, void *p_arg) {

    // Perform callback actions here.

    // Return the restart a periodic timer instruction to the timer handling routine.
    return BP_TIMER_PERIODIC;
}
```

Listing 26 – Periodic timer callback.

```
#include <timer/bp_timer.h>

int rtn;

// Start a timer with a period of 250 milliseconds.
rtn = bp_timer_start_ms(hndl, 250u, perioric_timer_callback, NULL);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 27 – Periodic timer start

9.6.5 Restarting a Timer

An expired or stopped timer can be restarted by calling `bp_timer_restart_ms()` or one of its variants. When restarting a timer, the new timer target is set from the last time the timer was set to expire, again preventing any drift. If instead the application wants to restart a timer from the current time `bp_timer_start()` can be used to restart an expired timer instead. Listing 28 shows an example of restarting a timer. When restarting a timer, the original callback will be used.

```
#include <timer/bp_timer.h>

int rtn;

// Restarts a timer with a period of 500 milliseconds.
rtn = bp_timer_restart_ms(hndl, 500u, NULL);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 28 – Restarting a timer

It is also possible to restart a timer from a timer's callback function as in Listing 29. In this case, `bp_timer_restart_ms()` is called from the callback to set the new period. When doing so, `BP_TIMER_RESTART` must be returned to inform the timer handling routine that the timer was restarted.

```
#include <timer/bp_timer.h>

bp_timer_action_t perioric_timer_callback(bp_timer_hndl_t hndl, void *p_arg) {
    int rtn;

    // Restart the timer while changing the period.
    rtn = bp_timer_restart_ms(hndl, 500u, NULL);
    if (rtn != RTNC_SUCCESS) { /* Error management */ }

    // Return the restart instruction to the timer handling routine.
    return BP_TIMER_RESTART;
}
```

Listing 29 – Restarting a timer from the callback function.

9.6.6 Stopping a Timer

The application can, at any time, stop a running timer using the API function `bp_timer_stop()`. An example of such a call is shown in Listing 30. If the timer is stopped successfully, the timer's callback won't be called. However if the timer has already expired then `bp_timer_stop()` will have no effect.

```
#include <timer/bp_timer.h>

int rtn;

// Stop the timer.
rtn = bp_timer_stop(hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 30 – Stopping a timer

9.7 Conclusion

The `time` and `timer` provides the application with an important set of services to measure and control the timing of an application's execution. Readers are encouraged to consult the API Reference Manual for both modules to get all the details on time and timer management. Also the reference manual contains additional convenience API to handle times in various units.

Chapter

10

Interrupt Management

10.1 Introduction

The interrupt management module or simply the `interrupt` module abstracts interrupt management and handling for the BASEplatform and the application. The services offered by the `interrupt` module includes registering interrupt service routines (ISR), enabling and disabling interrupt sources as well as controlling interrupt priority and type.

10.2 A Word on Platform Interrupt Support

This chapter documents the generic interrupt management API of the BASEplatform. Due to the wide variety of MCU and SoC many platforms will inevitably have some idiosyncrasies not covered by the generic API. Consequently, it is likely that the interrupt implementation on a specific platform deviates from the information contained in this section. While most deviations only affects the initialization, it is possible that some platforms require a slightly modified API or a special signature for interrupt service routines. Readers are advised to read the Platform Reference Manual for their SoC for further information.

10.3 Initialization

Very early in an application's lifetime the interrupt module should be initialized by calling `bp_int_init()`. An example of such a call is shown in Listing 31. Additional initialization and configuration steps may be required on certain platforms or when running with an RTOS. Those steps, if any, are documented in the Platform Reference Manual for each SoC or MCU. After initializing the `interrupt` module, the interrupts should be globally enabled by calling `bp_int_en()`. Note that the effect of `bp_int_en()` is platform specific and does more than just enabling the global interrupt line at the CPU level. Moreover, in order to disable and enable interrupts globally at runtime one should call `BP_ARCH_INT_DIS` and `BP_ARCH_INT_EN` instead of `bp_int_dis()`. `bp_int_dis()` is designed to disable the interrupt controller which may have unintended side effects and should be reserved for testing and debugging only.

```
#include <int/bp_int.h>

int rtn;

// Initialize the timer module.
rtn = bp_int_init().
if (rtn != RTNC_SUCCESS) { /* Error management */ }

// Enable interrupt processing.
rtn = bp_int_en().
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 31 – Initialization of the interrupt module.

10.4 Usage

The `interrupt` module is mostly used by drivers to register and control interrupts. Before an interrupt request can be serviced first, an ISR must be registered to that interrupt request then that it must be enabled. Optionally, the application may wish to alter the priority of the interrupt. Additionally on platforms that support it, it is possible to alter the type and polarity of an interrupt.

10.4.1 Registering an Interrupt Service Routine

An interrupt service routine within the BASEplatform looks like the function depicted in Listing 32. The interrupt handler will pass to the ISR three arguments, the first being a user-specified pointer, the second is the ID of the current interrupt. Lastly, the third argument is only used on SMP platforms and contains the source of software triggered interrupt for inter-core communication, in all other cases the value `source` is undefined.

```
#include <int/bp_int.h>

void isr_func(void *p_int_arg, int interrupt_id, uint32_t source)
{
    // ISR body.
}
```

Listing 32 – Example of interrupt service routine.

To register an ISR the driver or the application must call `bp_int_reg()` as shown in Listing 33. The first argument is the interrupt ID to register. Usually each SoC is provided a predefined list of interrupt names listed in a C enum variable. These are documented in the Platform Reference Manual for the SoC or MCU in question. It is usually recommended to use the enumeration constant instead of a hard-coded number. The second argument to `bp_int_reg()` is the ISR function to register, and the third argument is a user-defined pointer that will be passed to the ISR. It is possible to register an interrupt on an enabled interrupt line to replace a previously registered handler.

It's important to note that the method of mapping numerical interrupt id to physical interrupt lines is platform specific. This is another reason that the application should use the platform specific enumeration and must be careful with manipulating the interrupt id directly.

```
#include <int/bp_int.h>

int rtn

// Register ISR 'isr_func' to interrupt 10.
rtn = bp_int_reg(10u, isr_func, NULL);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 33 – Example of registering an ISR.

10.4.2 Enabling an Interrupt

After registering an interrupt that interrupt must be enabled before it can be used. Enabling an interrupt can be performed using `bp_int_src_en()`, passing the interrupt id to `bp_int_src_en()` like in Listing 34. Note that enabling an interrupt without a registered interrupt handler could potentially cause a CPU fault if that interrupt fires.

```
#include <int/bp_int.h>

int rtn

// Enabling interrupt 10.
rtn = bp_int_src_en(10u);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 34 – Example of enabling an ISR.

10.4.3 Disabling an Interrupt

An interrupt can also be disabled with `bp_int_src_dis()` as in Listing 35. Disabling an already disabled interrupt will have no effect.

```
#include <int/bp_int.h>

int rtn

// Disabling interrupt 10;
rtn = bp_int_src_dis(10u);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 35 – Example of disabling an ISR.

10.4.4 Configuring an Interrupt Priority and Type

On platforms that support multiple configurable interrupt priorities `bp_int_prio_get()` can be used to set the priority of an interrupt. Note that the numerical value of a priority is platform specific. Which means that the range of possible values as well as which end of the range corresponds to a higher priority can change from platform to platform.

```
#include <int/bp_int.h>

int rtn

// Set the priority of interrupt 10 to priority 1.
rtn = bp_int_prio_set(10u, 1u);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 36 – Example of configuring an interrupt priority.

The type of an interrupt can also be configured. The supported types may be fixed or limited but often an interrupt controller can allow changing an interrupt line from being edge sensitive to being level sensitive. Other controllers may also allow configuring the polarity of the interrupt trigger. If that is possible one can use the `bp_int_type_set()` to configure an interrupt, see Listing 37 for an example.

```
#include <int/bp_int.h>

int rtn

// Set interrupt 10 to trigger on a rising edge of the interrupt line..
rtn = bp_int_type_set(10u, BP_INT_TYPE_EDGE_RISING);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 37 – Example of configuring the sensitivity and polarity of an interrupt.

10.5 Conclusion

This chapter is only an introduction to the generic interrupt handling API of the BASEplatform. Readers are encouraged to read the BASEplatform API Reference Manual for the complete list of interrupt management functions and how they work in detail. In addition, it is strongly recommended to take a look at the Platform Reference Manual for important platform specific details of each individual SoC and MCU.

Chapter

11

Critical Section and Spin Lock

11.1 Introduction

The Critical Section and Spinlock module is abbreviated to `slock` in the documentation and the source code. The `slock` module provides uniprocessor and Symetric Multi Processing (SMP) compatible critical sections and spinlocks. The implementations and API are both compatible with bare-metal, uniprocessor RTOS and SMP RTOS. The `slock` module also features a unified spinlock API that can be used without a performance penalty across all types of systems, improving the portability of an application between uniprocessor and SMP systems.

11.2 What's a Critical Section?

Within the BASEplatform on a uni-processor system, a critical section is defined as a section of code where interrupts are disabled thus preventing an interrupt or a kernel context switch from happening. In effect the section is now guaranteed to be protected against concurrent execution since nothing is allowed to interrupt that section of code from executing. Critical sections are usually used to protect access to shared resources therefore preventing race conditions from happening.

Compared to mutexes and semaphores critical sections can be used from ISR and to protect against concurrent access from an ISR. This makes then used primarily when dealing with resources shared between the application and ISR. Critical sections also have a lower overhead than a mutex or a semaphore making them ideal for short sections of protected code. They are often used when simply updating or reading a shared variable or writing to a peripheral register.

Under an SMP system critical sections also protect against concurrent executions of other critical sections that may run on other CPU cores. Critical sections are not usually recommended in an SMP environment as they can have a considerable negative impact on performance. Spinlocks are recommended which are discussed in the next section.

11.3 What's a spinlock?

A spinlock in the BASEplatform is a form of mutex where a thread wanting to acquire a spinlock simply waits ("spin") for the lock to be free. The lack of a context switch means that spinlock can be used both to protect against concurrent access from multiple threads but also from interrupts. While the lock is held either by a thread or an interrupt the interrupts are disabled preventing the section of code protected by the spinlock to be interrupted.

Spinlocks are only useful under an SMP system. In the BASEplatform the spinlock implementation can be used on both SMP and uniprocessor systems. The implementation of the spinlock devolves to simple critical sections on a single-core application. This means that the spinlock API can be used to create applications compatible between uniprocessor and SMP kernels without a loss of performance due to the usage of spinlocks on the uniprocessor system.

11.4 Restrictions and Recommended Usage

In the case of both critical sections and spinlocks there are a few restrictions that must be observed. The first and most important is that a blocking function must never be used within the section of code protected by a critical section or a spinlock. With assertion checking enabled, the BASEplatform will attempt to guard against such errors. The second restriction is that the global interrupt flag should not be manipulated by the application while inside a protected section. Doing so could disable the afforded protection and desynchronize the state of the interrupt flag expected by the `slock` API.

When used with an RTOS the `slock` API can be used to supplement the kernel's native API. However care should be taken when mixing the use of the BASEplatform locks and critical sections and the kernel's critical sections.

The previous paragraphs detailed important restrictions that must always be observed. There are additional recommendations related to the usage of the `slock` API in addition to the restrictions above. The first recommendation is that the protected section of code should be as short as possible, otherwise excessively long critical sections could negatively affect interrupt latency and in some cases the application's performance as a whole. The second recommendation is to always use the spinlock API over the critical section API. This last recommendation helps with portability between uniprocessor and SMP systems. The spinlock API also offers various declination of the lock and unlock functions which can help improve performance further.

11.5 Critical Section Usage

Application developers are encouraged to use the spinlock API exclusively to improve portability between uniprocessor and SMP systems. However the critical section API remains available for developers who prefer simple critical sections. It's also important to note that critical sections are compatible with SMP systems but can have a negative impact on performance if they are overly used.

The critical section API is comprised of only two functions `bp_critical_section_enter()` and . When entering the critical section `bp_critical_section_enter()` returns a variable of type to hold the state of the global interrupt flag before entering the critical section. The same value must be passed to `bp_critical_section_exit()` to be restored when exiting the critical section. This allows the critical sections to be nested. Listing 40 shown an example of a simple critical section.

```c
#include <slock/bp_slock.h>

// Variable to hold the interrupt state before entering the critical section.
bp_irq_flag_t flag;

// Entering the critical section, taking care to save the interrupt flag.
flag = bp_critical_section_enter();

// Protected code goes here.

// Exiting the critical section passing the saved interrupt flag to be restored.
bp_critical_section_exit(flag);
```

Listing 38 – Simple critical section example.

11.6 Spinlock Usage

Spinlocks are used in a similar fashion to critical sections but require a spinlock object. This spinlock object can be used much like a mutex to protect shared resources. A simple usage of spinlock is shown in . The functions used `bp_slock_acquire_irq_save()` and `bp_slock_release_irq_restore()` are equivalent to the critical section API seen before but when running under an SMP rtos will only block other sections of code protected using the same lock object, allowing other protected section of code using a different lock to continue. When running under a uniprocessor system the example of is equivalent to a critical section.

```c
#include <slock/bp_slock.h>

// Variable to hold the interrupt state before entering the protected section.
bp_irq_flag_t flag;

// Spinlock object
slock_t lock;

// Acquiring the spinlock saving the interrupt state.
flag = bp_slock_acquire_irq_save();

// Protected code goes here.

// Releasing the spinlock while restoring the interrupt state.
bp_slock_release_irq_restore(flag);
```

Listing 39 – Simple critical section example.

11.6.1 Alternative Acquire and Release API

In addition to the acquire and release API shown in the last example, there are two other variants that can be used in special circumstances. The first one is `bp_slock_acquire_irq_dis()` and `bp_slock_release_irq_en()` as well as `bp_slock_acquire()` and .

bp_slock_acquire_irq_dis() and bp_slock_release_irq_en() can be used to unconditionally enable interrupts when exiting the protected section. This is only recommended when the use of spinlocks and critical sections are guaranteed not to be nested. bp_slock_acquire() and bp_slock_release() on the other hand do not manipulate the CPU interrupt disable flag at all. They should only be used when interrupts are guaranteed to be disabled prior entering the protected section. Both variants can offer a slight performance advantage over bp_slock_acquire_irq_save() and bp_slock_release_irq_restore(). Care should be taken to follow the restrictions associated with those alternative API functions. shows an example of all three types of spinlock API.

```
#include <slock/bp_slock.h>

// Variable to hold the interrupt state before entering the critical section.
bp_irq_flag_t flag;

// Spinlock object
slock_t lock;

// Default spinlock API. The IRQ flag variable must be passed between the acquire and
// release functions.
flag = bp_slock_acquire_irq_save();

// Protected code goes here.

bp_slock_release_irq_restore(flag);


// Unconditional IRQ enable API.
// Here the IRQ flag variable is unnecessary since the IRQ are always enabled when
   exiting
// the protected section. This kind of spinlock section should not be called if
// interrupts are disabled prior to entering the protected section. Also this kind of
// section cannot be nested within other protected sections.
bp_slock_acquire_irq_dis();

// Protected code goes here.

bp_slock_release_irq_en();


// Simple spinlock API to be used only when interrupts are disabled.
bp_slock_acquire();

// Protected code goes here.

bp_slock_release();
```

Listing 40 – Simple critical section example.

11.7 Conclusion

The slock module API is useful when writing codes that require short and fast sections of code to be protected against concurrent execution. Application developers are encouraged to take advantage of the spinlock features making them efficient and portable between SMP and uniprocessor systems. As usual readers are encouraged to look up the API Reference Manual of the BASEplatform for additional details on the `slock` module.

Chapter

12

Cache Management

12.1 Introduction

On platforms with CPU caches, the Cache Management Module or simply the `cache` module is used to perform cache maintenance operations. Those cache maintenance operations are primarily used by drivers performing DMA operations to and from cacheable main memory. When performing DMA operations from cacheable memory, one must use cache flush and invalidation to ensure cache coherency. Note that the `cache` module is concerned only with cache maintenance operations, primarily targeting the data cache. Initialization and configuration of the CPU caches are usually part of the startup sequence and would be provided by the appropriate architecture port delivered with the BASEplatform.

12.2 Overview

Cache maintenance is primarily divided into two operations, cache cleaning or flushing and cache invalidation. Cache flushing is useful to ensure that data written to the CPU cache but not yet written to main memory, also known as dirty cache lines, is flushed to memory. A cache flush operation is also known as a cache clean since it takes dirty data from the cache and writes it to main memory. This operation is useful when a buffer was written to by the application and must be read by a DMA engine. For example, when sending a data packet through an Ethernet interface. The cache flush operation is used to ensure that the Ethernet DMA engine is not reading the old data from main memory.

Cache invalidation removes data from the cache, allowing the CPU to fetch data directly from main memory. Cache invalidate operations are used to clear potentially stale data from the cache. This is useful when reading a buffer that was written by a DMA engine. For example, cache invalidation would be used before looking at a packet received from an Ethernet interface.

12.3 Usage

This section goes over the three main types of operations that can be performed with the `cache` module. Either invalidating the entire cache hierarchy, cleaning a range of memory from the cache or invalidating a range of memory from the cache.

12.3.1 Invalidate the Entire Cache

Invalidating the entire data or instruction cache is usually only used during initialization or when waking up from a low-power sleep. Listing 43 shows an example of calling `bp_cache_icache_inv_all()` and `bp_cache_dcache_inv_all()`.

```
#include <cache/bp_cache.h>

// Invalidate the instruction cache.
bp_cache_icache_inv_all();

// Invalidate the data cache.
bp_cache_dcache_inv_all();
```

Listing 41 – Invalidating the entire data and instruction caches.

12.3.2 Cache Clean

Once data has been written into a buffer that will be processed by DMA, it is usually necessary to clean the cache over the range of addresses covering the buffer. The buffer should be aligned to a cache line and should also be a multiple of a cache line otherwise the flush operation could affect memory outside the buffer. The cache can be cleaned using `bp_cache_dcache_range_clean()` as shown in Listing 42.

```
#include <cache/bp_cache.h>

// DMA buffer, should be aligned.
uint8_t buffer[256u];

// Buffer is written by the application here.
app_fill_buffer(buffer, 256u);

// Before sending the buffer to a DMA engine it must be cleaned to
// ensure that all the data is written to main memory.
bp_cache_dcache_range_clean(buffer, 256u);

// DMA operation can now be performed here.
dma_run(buffer);
```

Listing 42 – Cache clean example.

12.3.3 Cache Invalidate

Cache invalidation is used when a DMA engine has written to a buffer in main memory and the application wants to read from that buffer. The same constraint as cache flushing must be followed, the buffer must be aligned to a cache line and must be a multiple of a cache line size. Otherwise the cache invalidate operation could corrupt memory outside the buffer. Listing 43 shows how `bp_cache_dcache_range_inv()` can be used to handle a DMA buffer. Note that it is necessary on most platforms to invalidate twice. The first invalidate is to prevent any dirty data in the cache from being written to main memory by the cache controller and corrupting the data just written by the DMA engine. The second invalidate is to make certain that the cache doesn't contain stale data that would be returned to the CPU instead of the correct data residing in main memory.

```
#include <cache/bp_cache.h>

// DMA buffer, should be aligned.
uint8_t buffer[256u];

// Prior to starting the DMA operation the buffer must be invalidated
// to prevent dirty data from corrupting the data in main memory.
bp_cache_dcache_range_inv(buffer, 256u);

// DMA operation can now be performed here.
dma_run(buffer);

// After the DMA operation is completed, but before the application
// can read the data the buffer must be invalidated again. This prevents
// stale data from the cache being returned to the CPU instead of the data
// in main memory.
bp_cache_dcache_range_inv(buffer, 256u);

// Application can read the buffer now.
```

Listing 43 – Cache invalidate example.

12.3.4 Query the Data Cache Line Size

It can be useful for the application to know the data cache line size, for example to configure buffer alignment. The data cache line size is, however, not necessarily uniform across the CPU cache hierarchy and sometimes it is important to know either the smallest or the largest cache line size. The `cache` module contains two API to do just that `bp_cache_dcache_min_line_get()` and `bp_cache_dcache_max_line_get()`. Examples of their usage are shown in Listing 44.

```
#include <cache/bp_cache.h>

// Variable to hold the returned cache line size.
uint32_t line_size;

// Query the smallest cache line size in the data cache hierarchy.
line_size = bp_cache_dcache_min_line_get();

// Query the largest cache line size in the data cache hierarchy.
line_size = bp_cache_dcache_max_line_get();
```

Listing 44 – Querying the Data Cache Line Size.

12.4 Conclusion

This chapter has gone over an overview of cache management operations that can be performed with the `cache` module. While a small module the `cache` module is nonetheless essential on SoC and MCU using CPU data cache. Developers who must performance cache maintenance operations in their applications are encouraged to consult the BASEplatform API Reference Manual as well as the Platform Reference Manual for their platform for additional information on cache management.

Chapter

13 Universal Asynchronous Receiver Transmitter (UART)

13.1 Introduction

The BASEplatform `uart` module is used to interface with Universal Asynchronous Receiver-Transmitter (UART) and similar UART-like peripherals. A UART serial link is usually comprised of two independent transmit and receive paths. Each UART link is asynchronous most often use a simple implement a clocked version of the protocol or more advanced clock recovery schemes with baud rate detection. UART and UART-like peripherals are often used as the basis of higher-level protocols and communication standards such as RS-232, RS-485, IrDA and many others. The BASEplatform doesn't assume a specific protocol or target use and can usually work with most variations of the basic UART protocol.

Considering the wide variety of UART and UART-like peripherals, it would be impossible to design a high-level and portable API that could leverage the unique features of each and every peripheral. To alleviate this, drivers are allowed to implement driver-specific functionalities to extend the features of the core API of the `uart` module. Details on how to access those features is described later in this chapter.

13.2 Overview

The `uart` module features all of the necessary API to interact with a UART peripheral. This includes the usual lifecycle management functions of a BASEplatform I/O module. Next, the module allows control over the expected protocol parameters such as the baud rate, number of stop bits and parity. Finally, the API also contains a blocking transmit and receive API with optional timeout and polling mode option as well as an asynchronous I/O API.

The blocking API of the `uart` module allows concurrent access to both the transmit and receive path without blocking each other. This enables maximum performance and flexibility when transmitting and receiving at the same time. However, some portion of the API affects both the transmit and receive channels. An example of such a function is the UART configuration set function, `bp_uart_cfg_set()`. Those functions require access to both paths and thus using them will block, temporarily any other transmit or receive calls. Internally this is due to those API trying to lock both channels before performing any actions that may affect both transmit and receive.

13.3 Lifecycle

The `uart` module follows the standard lifecycle followed by many of the BASEplatform I/O modules. The various states and functions to move between those states are schematized in Figure 5.

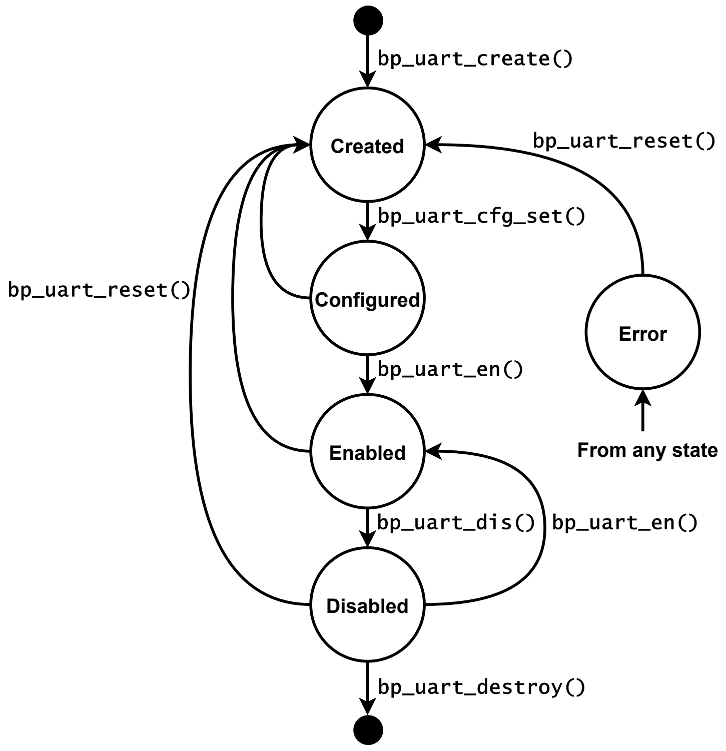

Figure 5 – State diagram of the `uart` module's lifecycle.

To be useful a `uart` module instance must first be created by calling `bp_uart_create()`. Once created the module instance must be configured with `bp_uart_cfg_set()` and then enabled using `bp_uart_en()`. At this point the `uart` module instance is ready to be used for transmission and reception. The module instance can then be disabled if desired by calling `bp_uart_dis()` and finally destroyed to reclaim any resource associated with the module instance with `bp_uart_destroy()`. It is also possible to reset a module instance using `bp_uart_reset()`.

The following sections look at each step in more details.

13.3.1 Create

The first step in the life of a `uart` module instance is to create it. This can be achieved using the `bp_uart_create()` function. After being created the new instance will be left in the created state after which it should usually be configured and then enabled prior to being used. An example of module creation is shown in Listing 45.

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

int rtn;

// Variable that will receive the newly created handle.
bp_uart_hndl_t hndl;


rtn = bp_uart_create(&g_uart_board_def, &hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 45 – Creating a new UART module instance.

bp_uart_create() takes as first argument a pointer to the board definition structure of the UART peripheral to be associated with the instance. These UART definition structures are part of the board definition and usually accessible by including the bp_board_def.h header file. See Chapter 4 for more information on board definitions. The second argument to bp_uart_create() is a pointer to the handle. If successful, it will be set to the newly created module handle.

Prior to returning, bp_uart_create() will perform all the necessary memory allocation required by the uart module and driver. This means that once created, a uart instance has all the necessary memory required for its entire lifetime. See Chapter 6 for additional details on memory allocation.

13.3.2 Configure

A newly create uart instance must be configured at least once prior to being enabled and used for communication by calling bp_uart_cfg_set(). Afterward it is possible to update the configuration at runtime by calling bp_uart_cfg_set() again. At any time the current configuration can be queried by calling bp_uart_cfg_get().

If the uart instance was just created, i.e. it's in the created state, it will transition to the configured state upon a successful call to bp_uart_cfg_set(). After being configured for the first time, the module can then be enabled. If the instance was already configured the configuration is simply updated and the instance stays in the same state as it was before unless an error occurs.

Listing 46 shows an example of module configuration where the UART peripheral is set to a baud rate of 115200 with no parity and one stop bit.

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

int rtn;

// UART configuration structure.
bp_uart_cfg_t cfg;

// Configuration to apply.
cfg.baud_rate = 115200u;
cfg.parity = BP_UART_PARITY_NONE;
cfg.stop_bits = BP_UART_STOP_BITS_1;
```

```
rtn = bp_uart_cfg_set(hndl, &cfg, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 46 – Example of UART configuration.

Apart from the module instance handle and the usual timeout value, `bp_uart_cfg_set()` takes as an argument a pointer to a configuration structure of type `bp_uart_cfg_t`. This configuration structure contains the baud rate, parity and stop bit configurations to be applied. See the API reference manual for additional details on the UART configuration structure and the side effects of `bp_uart_cfg_set()`.

13.3.3 Enable

A `uart` instance in the configured state must be enabled prior to being used, this can be achieved by calling `bp_uart_en()`. A disabled instance can also be enabled by the same call. A successful call to `bp_uart_en()` will place the module instance in the enabled state and it is then ready for transmission and reception. If the instance was already enabled, calling `bp_uart_en()` should have no effect.

The exact side effects of enabling a `uart` instance is specific to the driver and UART peripheral. In most cases it will enable the UART peripheral's clock and bring it out of reset if appropriate.

An example of enabling a `uart` instance is shown in Listing 47

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

int rtn;

rtn = bp_uart_en(hndl, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 47 – Enabling a UART module instance.

13.3.4 Disable

At runtime it is possible to disable an enabled `uart` instance with the `bp_uart_dis()` function. After being disabled, no other operations should be performed on the instance other than enabling it with `bp_uart_en()` or resetting the instance using `bp_uart_reset()`. Any other operation runs the risk of accessing a disabled peripheral which could cause a bus fault or hang. If assertion checks are enabled in the BASEplatform configuration (See Chapter 7) a fatal error may be returned when trying to access a disabled instance.

The exact side effects of disabling a `uart` instance is driver and hardware specific. If possible, the device is placed in a low power disabled state and the peripheral's clock is disabled. Also it is important to note that the created and configured state are not equivalent to the disabled state, especially from a power consumption point of view. Application developers who wish to configure a UART peripheral but leave it in a low power state should fully create and configure the instance and then call `bp_uart_dis()` to disable the module instance and peripheral.

Listing 48 presents an example of disabling a `uart` module instance.

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

int rtn;

rtn = bp_uart_dis(hndl, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 48 – Disabling a UART module instance.

13.3.5 Destroy

At any point if a `uart` instance is no longer needed, it can be destroyed using `bp_uart_destroy()`. Destroying an instance will reclaim any resource assigned to the instance but only if freeing of memory is permitted by the primary memory allocator. For more details on allocation policy see Chapter 6. Once destroyed the instance handle becomes invalid and should not be used again.

It is important to first disable a `uart` instance prior to destroying it otherwise the peripheral will be left active. Listing 49 contains an example of destroying a `uart` module instance.

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

int rtn;

// First disable the instance.
rtn = bp_uart_dis(hndl, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }

rtn = bp_uart_destroy(hndl, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 49 – Destroying a UART module instance.

13.3.6 Reset

A `uart` instance in any state, other than destroyed, can be reset with the `bp_uart_reset()` function. Resetting an instance will bring it back to the created state. If possible, the UART driver will try to perform a hardware soft reset on the peripheral if this is supported. This soft reset can either be at the peripheral level if there is a soft reset bit, or using a centralized peripheral reset controller if it is available on the current SoC.

Reset is the only operation that should be performed on a module instance that encountered an unexpected error, i.e., returned `RTNC_FATAL`. A reset could be attempted to return the internal state of the module and peripheral to a known state.

An example of reset is shown in Listing 50

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

int rtn;

rtn = bp_uart_reset(hndl, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 50 – Resetting a UART module instance.

13.4 Basic Usage

The most basic and most common usage of the `uart` module is to transmit and receive bytes through the UART peripheral interface using a blocking or polling API. This can be achieved using `bp_uart_tx()` for transmission and `bp_uart_rx()` for reception.

13.4.1 Transmission

An example of transmitting through a UART interface is shown in Listing 51. In the example, 8 bytes from the buffer `buf` is sent for transmission through the UART interface associated with the handle `hndl`. A successful return from `bp_uart_tx()` means that the bytes passed where either transmitted or have been queued in the UART peripheral internal FIFO. If there is not enough space in the FIFO to hold all the bytes to transmit, `bp_uart_tx()` will block until transmission is complete.

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

int rtn;

// Buffer to transmit.
uint8_t buf[8] = {0, 1, 2, 3, 4, 5, 6, 7};

rtn = bp_uart_tx(hndl, buf, 8u, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 51 – UART transmission example.

The timeout value passed to `bp_uart_tx()` specifies that amount of time to wait for the instance to be free, meaning that it is not accessed from another thread or performing an asynchronous operation. It is important to realize that if the interface becomes free within the timeout period, `bp_uart_tx()` will then block until the specific number of bytes to transfer is either sent or queued. This behaviour is slightly different on reception as described in the following section.

13.4.2 Reception

Reception is very similar to transmission as shown in Listing 52. When receiving, a buffer of a suitable size must be passed to `bp_uart_rx()` along with the maximum number of bytes to receive. Upon returning from `bp_uart_rx()` the actual number of bytes received will be stored in the variable passed as the `p_rx_len` argument.

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

int rtn;

// Buffer to store the received bytes.
uint8_t buf[8];

// Variable to receive the number of bytes actually received.
size_t rx_len;

rtn = bp_uart_rx(hndl, buf, 8u, &rx_len, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 52 – UART reception example.

When performing a UART receive operation, the timeout argument serves a dual purpose. It specifies the timeout for the entire reception operation, including the time waiting for the interface to be available, as well as the time it takes to receive the data. Passing a timeout value other than means that `bp_uart_rx()` will return with an `RTNC_TIMEOUT` return code if the specified amount of data wasn't received in time. `RTNC_TIMEOUT` will also be returned if the instance didn't become free within the timeout period. If a timeout occurs, the variable passed as the `p_rx_len` argument should be checked to see how many, if any, bytes were received.

Using a timeout value of 0 means that `bp_uart_rx()` will work in polling mode. In polling mode, `bp_uart_rx()` will return immediately with the data available from the UART FIFO buffer. With a timeout value of 0 the function will also return immediately if the interface is not free right away. In this last scenario, the number of bytes received is set to 0.

13.4.3 Flushing the UART FIFOs

UART peripherals often use a receive and transmit FIFOs to hold data temporarily thus improving performance and reducing CPU usage. This means that, especially on reception, it is possible to have stale data in the receive FIFO. To synchronize the flux of data the `uart` module possesses two flush functions, one for the transmit and the other for the receive path, `bp_uart_tx_flush()` and `bp_uart_rx_flush()`. Of the two, `bp_uart_rx_flush()` is often the most useful to clear any stale data in the receive FIFO as is shown in Listing 53.

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

int rtn;

rtn = bp_uart_rx_flush(hndl, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 53 – Flushing the UART receive FIFO.

13.5 Asynchronous I/O

Apart from the blocking API, the `uart` module also has a non-blocking asynchronous API. The asynchronous API allows one transmit and one reception operation to be done without blocking. Upon completion a user-specified callback will be called to report the result. From within the user supplied callback, it is possible to either finish the transfer or start another asynchronous transfer immediately. This last feature can be useful when having to continuously transmit or receive to and from a circular buffer.

Asynchronous reception and transmission are both performed in a similar fashion. The two operations require the setup of a descriptor structure detailing the transfer to be performed. The structure of type `bp_uart_tf_t` as shown in Listing 54 contains information about the buffer to use for reception or transmission, the length of the transfer to perform as well as a pointer to a user-specified callback function and context pointer.

```
struct bp_uart_tf {
    void *p_buf; // Memory buffer.
    size_t len; // Length of the data.
    bp_uart_async_cb_t callback; // Callback function.
    void *p_ctxt; // Optional user context pointer.
};
```

Listing 54 – UART asynchronous transfer structure definition.

To signal the application that a transfer is completed the UART driver will call a user-specified callback similar to the one in Listing 55. Through the callback arguments, the application is informed of the status of the completed transfer and the amount of data transferred. A pointer to the transfer descriptor structure is also passed to the callback. Within that description structure, the application code can access the original transfer parameters as well as the user context pointer. That user context pointer can be used to pass information from the application to the callback.

```
bp_uart_action_t uart_async_callback(int status, size_t tf_len,
    bp_uart_tf_t *p_tf) {

   // Action to be undertaken by the UART driver interrupt
   // handler upon return.
   return BP_UART_ACTION_FINISH;
}
```

Listing 55 – UART asynchronous transfer callback example.

When retuning from the callback, the application should use one of the two possible return values, namely `BP_UART_ACTION_FINISH` or `BP_UART_ACTION_RESTART`. If the callback returns with `BP_UART_ACTION_FINISH`, as shown in Listing 55, the transfer completes normally. If, however `BP_UART_ACTION_RESTART` is returned, the transfer will be restarted with the information from the transfer descriptor. The application is allowed to modify the transfer descriptor through the `p_tf` argument prior to restarting the transfer.

When performing an asynchronous transfer the transfer can be considered to have been started when `bp_uart_tx_async()` or `bp_uart_rx_async()` returns. It is important to keep in mind that, due to timing and context switch, the transfer may have already completed before returning.

13.5.1 Asynchronous Transmission

Listing 56 contains an example of asynchronous transmission. The callback function, as explained above isn't shown but should be defined somewhere. It's imperative to keep the buffer valid for the entire duration of the asynchronous transfer otherwise the transmitted data might be corrupted.

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

int rtn;

// Buffer to be sent.
uint8_t buf[8] = {0, 1, 2, 3, 4, 5, 6, 7};

// Transfer description structure.
bp_uart_tf_t tf;

// Transfer setup.
tf.p_buf = buf;
tf.len = 8u;
tf.callback = uart_async_callback;
tf.p_ctxt = NULL;

rtn = bp_uart_tx_async(uart_hndl, &tf, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 56 – UART asynchronous transmit example.

13.5.2 Asynchronous Reception

Asynchronous reception is very similar to the previous transmission example as shown in Listing 57. In fact, the function signatures are the same. Note that the number of bytes read is passed to the callback function.

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

int rtn;

// Buffer to hole the received data.
uint8_t buf[8];

// Transfer description structure.
bp_uart_tf_t tf;

// Transfer setup.
tf.p_buf = buf;
tf.len = 8u;
```

```
    tf.callback = uart_async_callback;
    tf.p_ctxt = NULL;

    rtn = bp_uart_rx_async(uart_hndl, &tf, TIMEOUT_INF);
    if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 57 – UART asynchronous reception example.

13.5.3 Aborting an Asynchronous Transfer

An asynchronous operation can be aborted with the two functions `bp_uart_tx_async_abort()` and `bp_uart_rx_async_abort()` as depicted in . Each abort functions has an optional argument, a pointer to a variable that will receive the number of bytes transmitted or received at the point of aborting. Note that it's possible for the transfer to complete while the abort function is called in which case the returned data length will be set to 0 and the abort won't have any effects.

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

int rtn;

// Variable to receive the number of bytes transferred prior to aborting.
size_t tf_len;

// Aborting a receive operation.
rtn = bp_uart_rx_async_abort(hndl, &tf_len, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }

// Aborting a transmit operation.
rtn = bp_uart_tx_async_abort(hndl, &tf_len, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 58 – Aborting an asynchronous transfer.

13.5.4 Waiting for a UART Interface to be Idle

If an application desires to wait for a UART interface to be idle, it can do so with the `bp_uart_tx_idle_wait()` and `bp_uart_rx_idle_wait()` API functions. Those two functions will wait for any blocking or asynchronous operations to be completed before returning. In addition `bp_uart_tx_idle_wait()` will wait for the transmit FIFO to be empty prior to returning. Note that this last feature doesn't guarantee that all the data in the FIFO is actually sent since most UART peripherals do not have a status bit to indicate that the transmit register is empty. En example usage of `bp_uart_tx_idle_wait()` is shown in Listing 59

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

int rtn;

rtn = bp_uart_tx_idle_wait(hndl, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 59 – Waiting for the transmit path to be idle.

13.6 Direct Access to the Driver

The `uart` module is designed to offer a strong set of features that covers the majority of use cases. At the same time, the set of features is also selected to ensure that it can be supported by a large majority of UART peripherals. As such, it is natural that some peripheral specific features cannot be supported by the top-level portable API of the `uart` module. To alleviate this, the UART drivers can implement driver specific functions available to the application to access advanced features of the underlying peripheral. To use them, however, the application needs to access the driver directly which is explained in this section.

Apart from accessing driver specific features, using the driver interface directly offers a slight improvement in performance since the call overhead is reduced. However it also means that access to the driver is not inherently thread-safe since the top-level `uart` module is responsible for ensuring thread safety. Finally, for applications that want to reduce the RAM usage to a minimum, it is possible to instantiate a UART driver by itself without having an instance of the top-level `uart` module. Information on the driver API can be found in the API reference manual.

13.6.1 Retrieving the Driver Handle

The first step in accessing a driver is to retrieve its handle. This can be performed using `bp_uart_drv_hndl_get()` as in Listing 60. In the example, if successful, the driver handle will be set in `drv_hndl`. That handle can then be used to access the driver API.

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

int rtn;

// Variable that will receive the driver handle.
bp_uart_drv_hndl_t drv_hndl;

rtn = bp_uart_drv_hndl_get(hndl, &drv_hndl);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 60 – Retrieving the driver handle from a UART module instance.

13.6.2 Locking the UART Module Instance

Most BASEplatform drivers are not inherently thread-safe. As such, when accessing a driver directly, care must be taken in a multitasking environment to prevent concurrent access to the same driver instance. This can be achieved in multiple ways. At the application level, one of the simplest way is to ensure that the driver and `uart` module instance are only accessed from a single thread only. Another option is to implement a locking mechanism at the application level using a mutex or semaphore. This last method requires an extra kernel object, however.

To simplify concurrent access to the driver and module instance from multiple threads, the `uart` module API includes two functions, `bp_uart_acquire()` and `bp_uart_release()` to acquire and release the module instance mutexes. Acquiring the instance prevents any concurrent access from any thread trying to use the top level `uart` module API. To work properly, however, this requires all the threads accessing the driver to use the acquire and release calls. It is also possible to use the top-level `uart` module API while locked as the locking is recursive.

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

int rtn;

rtn = bp_uart_acquire(hndl, TIMEOUT_INF)
if(rtn != RTNC_SUCCESS) { /* Error management */ }

// Perform driver and uart module operations here.

rtn = bp_uart_release(hndl, TIMEOUT_INF)
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 61 – Acquire and Release of a UART Instance.

13.6.3 Calling a Driver API

Listing 62 shows a complete example of calling a driver API including fetching the driver handler and locking the `uart` module. For the purpose of the example, it is assumed that the STM32 UART driver is being used, but it would work in a similar fashion with any other driver. Note that to access the driver API it's necessary to include the driver's header file, `bp_stm32_uart_drv.h` for this example.

```
#include <uart/bp_uart.h>
#include <soc_comp/stmicro/stm32_uart/bp_stm32_uart_drv.h>

int rtn;

// Variable that will receive the driver handle.
bp_uart_drv_hndl_t drv_hndl;

rtn = bp_uart_drv_hndl_get(hndl, &drv_hndl);
if(rtn != RTNC_SUCCESS) { /* Error management */ }

rtn = bp_uart_acquire(hndl, TIMEOUT_INF)
if(rtn != RTNC_SUCCESS) { /* Error management */ }

// Call the driver function. Note the usage of the driver handle.
rtn = bp_stm32_uart_en(drv_hndl, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

```
rtn = bp_uart_release(hndl, TIMEOUT_INF)
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 62 – Direct driver access example.

13.6.4 Alternative Driver Calling Method

In the previous section's example, a driver's function was called directly using its function name. For driver specific functions, which are non-standard this is the only way possible. However for the standard portion of the driver API, such as the lifecycle management functions and transfer functions, it is also possible to use the driver API structure. This structure is declared in the driver header file, for example to keep the same example of the STM32 UART driver, the driver structure is called `g_bp_stm32_uart_drv`. Using it would look like the example in Listing 63.

```
#include <uart/bp_uart.h>
#include <soc_comp/stmicro/stm32_uart/bp_stm32_uart_drv.h>

int rtn;

rtn = g_bp_stm32_uart_drv.en(drv_hndl, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 63 – Direct driver access alternative example.

13.7 Conclusion

This concludes the chapter on the `uart` module. The chapter covered the basics of using the `uart` module such as the lifecycle and basic transmission and reception. The chapter also went over the asynchronous API and how to access the UART drivers directly. Readers are encouraged to read the `uart` module API reference for additional details on its usage and other advanced features of the API. BASEplatform users should also take a look at the Platform Reference Manual for their selected platform for addition information about their platform.

Chapter

14 Serial Peripheral Interface (SPI)

14.1 Introduction

The `spi` module is used to interface with external components using a Serial Peripheral Interface (SPI) peripheral and link. An SPI interface is comprised of two synchronous serial links for input and output, a clock line as well as one or more optional chip select lines. Communication is controlled entirely by the master who performs simultaneous reading and writing by toggling the clock line.

Considering the wide variety of SPI and SPI- compatible peripherals it would be impossible to design a high-level and portable API that could leverage the unique features of each and every peripheral. To alleviate this, drivers are allowed to implement driver-specific functionalities to extend the features of the core API of the `spi` module. Details on how to access those features are described later in this chapter.

14.2 Overview

The `spi` module offers all the usual API functions to interact with an SPI peripheral. This includes the usual lifecycle management functions as well as transmission, reception and control of the SPI chip select lines if present. The module also enables the application to configure the clock phase and polarity (a.k.a the SPI mode) as well as the serial clock frequency. The data transfer API features the usual blocking, polling and asynchronous I/O variants for transmission and reception.

The communication API can be used to perform transmission and reception at the same time either using two different buffers or by swapping the transmitted and received data in a single buffer. While this last feature is rarely used when interacting with external peripherals, it can be useful when performing full duplex communication with another MCU.

14.3 Lifecycle

The `spi` module follows the standard lifecycle used by many of the BASEplatform I/O modules. The various states and functions to move between those states are schematized in Figure 6.

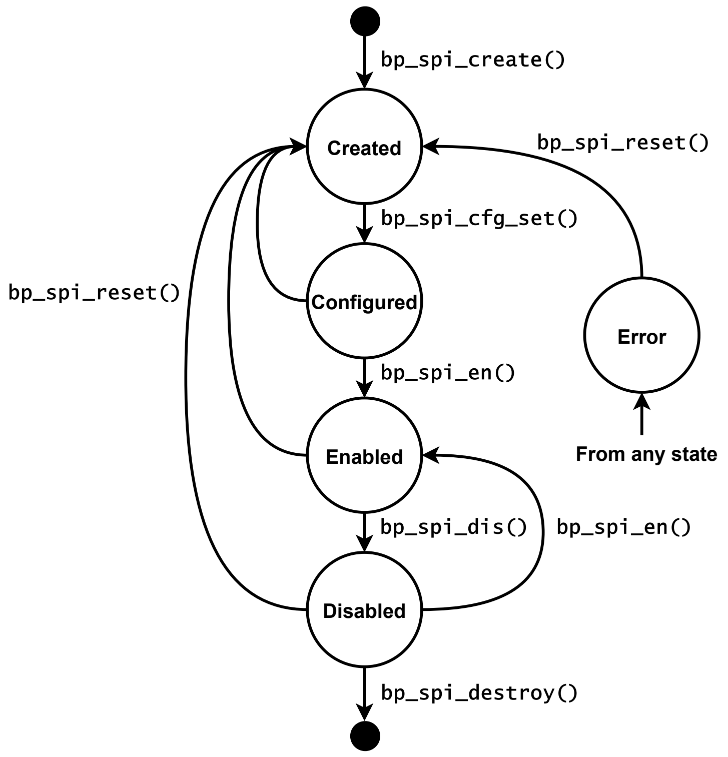

Figure 6 – State diagram of the `spi` module's lifecycle.

To be useful a `spi` module instance must first be created by calling `bp_spi_create()`. Once created the module instance must be configured with `bp_spi_cfg_set()` and then enabled using `bp_spi_en()`. At this point the `spi` module instance is ready to be used for transmission and reception. The module instance can then be disabled if desired by calling `bp_spi_dis()` and finally destroyed to reclaim any resource associated with the instance by calling `bp_spi_destroy()`. It is also possible to reset a module instance using `bp_spi_reset()`.

The following sections look at each step in more details.

14.3.1 Create

Like most I/O modules of the BASEplatform the first step in order to use the `spi` module is to create a new `spi` module instance. This can be done using the `bp_spi_create()` function. After being created successfully, the newly created instance will be left in the created state after which it should usually be configured and enabled prior to being used. Listing 64

```
#include <spi/bp_spi.h>
#include <board/bp_board_def.h>

int rtn;

// Variable that will receive the newly created handle.
bp_spi_hndl_t hndl;
```

```
rtn = bp_spi_create(&g_spi_board_def, &hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 64 – Creating a new SPI module instance.

The first argument to `bp_spi_create()` is a pointer to the board definition structure of the SPI peripheral to associate with the new instance. These SPI definition structures are part of the board definition and usually accessible by including the `bp_board_def.h` header file. See Chapter 4 for more information on board definitions. The second argument to `bp_spi_create()` is a pointer to the handle. If successful, it will be set to the newly created `spi` module handle.

`bp_spi_create()` performs all the necessary resource allocation required by the `spi` module and associated SPI driver prior to returning. As such, once created, an `spi` module instance has all the necessary memory and kernel objects required for its entire lifetime. See Chapter 6 for additional details on memory allocation.

14.3.2 Configure

After being created an `spi` instance must be configured at least once before being enabled and used for communication. Configuration is performed using `bp_spi_cfg_set()`. After being configured for the first time, it is possible to update the configuration at runtime by calling `bp_spi_cfg_set()` again. It is also possible to query the current configuration with `bp_spi_cfg_get()`.

If the `spi` instance was in the created state prior to the call to `bp_spi_cfg_set()`, it will transition to the configured state if the configuration is successful. Afterward it can then be enabled and used. If the instance was already configured then the configuration is updated and the `spi` instance's state remains unaffected.

Listing 65 contains an example of configuration. In the example a configuration structure, `cfg`, of type `bp_spi_cfg_t` is populated to set the configuration of an SPI peripheral. The configuration used in the example is for a master running at a 1 MHz clock rate with polarity and phase set to 0. The populated configuration structure is then passed to `bp_spi_cfg_set()` to perform the configuration.

```
#include <spi/bp_spi.h>

int rtn;

// SPI configuration structure.
bp_spi_cfg_t cfg;

// SPI configuration.
cfg.bit_rate = 1000000u;
cfg.clk_phase = 0u;
cfg.clk_polarity = 0u;
cfg.master = true;

rtn = bp_spi_cfg_set(hndl, &cfg, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 65 – Configuring an SPI module instance.

14.3.3 Enable

Once configured an `spi` instance must be enabled. This is performed using `bp_spi_en()`. A disabled instance can also be enabled using the same call. A successful call to `bp_spi_en()` will place the instance in the enabled state and it will now be ready to be used for transmission and reception. If the instance was already in the enabled state then `bp_spi_en()` should have no effect.

The exact side effect of enabling a `spi` module instance is driver and platform specific. In most cases it will enable the peripheral itself, take it out of reset and activate the peripheral's clock.

An example of enabling an `spi` module instance is shown in Listing 66.

```c
#include <spi/bp_spi.h>

int rtn;

rtn = bp_spi_en(hndl, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 66 – Enabling an SPI module instance.

14.3.4 Disable

At runtime it is possible to temporarily disable a previously enabled `spi` instance by calling `bp_spi_dis()`. After being disabled no other operations should be performed on the disabled instance other than enabling it with `bp_spi_en()` or resetting the instance using `bp_spi_reset()`. Any other operation runs the risk of accessing a disabled peripheral which could cause a bus fault or hang. If assertion checks are enabled in the BASEplatform configuration (See Chapter 7), a fatal error may be returned when trying to access a disabled instance.

The exact side effects of disabling an `spi` instance are driver and hardware specific. If possible, the device is placed in a low power disabled state and the peripheral's clock is disabled. Also it is important to note that the created and configured state are not equivalent to the disabled state, especially from a power consumption point of view. Application developers who wish to configure a SPI peripheral but leave it in a low power state should fully create and configure the instance and then call `bp_spi_dis()` to disable the module instance and peripheral.

Listing 66 presents an example of disabling an `spi` module instance.

```c
#include <spi/bp_spi.h>

int rtn;

rtn = bp_spi_dis(hndl, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 67 – Disabling an SPI module instance.

14.3.5 Destroy

When an `spi` module instance is no longer needed it can be destroyed using `bp_spi_destroy()`. Doing so will reclaim any resource allocated by the instance assuming that the freeing of memory is allowed by the default allocator. For more details on allocation policy see Chapter 6. Once destroyed the instance handle becomes invalid and should not be used again.

It is important to disable an `spi` instance prior to destroying it, otherwise the peripheral might be left active. An example of destroying an `spi` instance is shown in Listing 68.

```
#include <spi/bp_spi.h>

int rtn;

// First disable the instance.
rtn = bp_spi_dis(hndl, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }

rtn = bp_spi_destroy(hndl, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 68 – Destroying an SPI module instance.

14.3.6 Reset

An `spi` instance in any state, other than destroyed, can be reset with the `bp_spi_reset()` function. Resetting an instance will bring it back to the created state. If possible, the SPI driver will try to perform a hardware soft reset on the peripheral if this is supported. This soft reset can either be at the peripheral level if there is a soft reset bit, or using a centralized peripheral reset controller if it is available on the current SoC.

Reset is the only operation that should be performed on a module instance that encountered an unexpected error, i.e. returned `RTNC_FATAL`. A reset could be attempted to return the internal state of the module and peripheral to a known state.

An example of reset is shown in Listing 69

```
#include <spi/bp_spi.h>

int rtn;

rtn = bp_spi_reset(hndl, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 69 – Resetting an SPI module instance.

14.4 Basic Usage

This section goes over the basic usage of the SPI module. This includes transmission and reception of data and manipulation of the chip select lines.

14.4.1 Chip Select and De-Select

Before performing a master SPI transfer, it is often required to assert the chip select of the targeted slave. Doing so can be handled in two ways depending on the SPI peripheral in use and how the chip select are wired. The first method is to use the dedicated chip select API of the `spi` module. This method is used when the current SPI driver and SoC have dedicated chip select lines which are manipulated through the SPI peripheral. An example of asserting and deasserting such a chip select line is shown in Listing 70. Performing a chip select in this way will also take exclusive control the `spi` module instance to prevent other threads from attempting to do a transfer of their own to the wrong slave.

```
#include <spi/bp_spi.h>

int rtn;

 rtn = bp_spi_slave_sel(hndl, 0u, TIMEOUT_INF);
 if (rtn != RTNC_SUCCESS) { /* Error management */ }

 // Perform transfer here.

 rtn = bp_spi_slave_desel(hndl, TIMEOUT_INF);
 if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 70 – Controlling the chip select lines.

If a SoC or MCU doesn’t have any dedicated chip select lines then the chip select must be wired to a GPIO. In that case the `gpio` module should be used to assert and de-assert the chip select line. Details of the `gpio` modules can be found in Chapter 16.

14.4.2 Master SPI Transfer

At the logic level, an SPI transfer is performed by shifting data out of a shift register from the master to the slave while shifting in data from the slave to the master. As such an SPI peripheral is always transmitting and receiving at the same time. Since transmission and reception are the same operation to the SPI peripheral the `spi` module only has one transfer function `bp_spi_xfer()`, which can be used for both transmission and reception.

The application specifies the transfer to perform using a transfer description structure of type `bp_spi_tf_t`, the definition of which is reproduced in Listing 71. The structure that must be populated by the application contains various fields, two of which, `callback` and `p_ctxt` are only used by asynchronous transfers explained later in this chapter. They can be set to NULL or ignored for blocking transfers.

The transfer description structure contains two buffer pointers, one for transmission and one for reception. The application can set one of them to NULL if only transmission or reception is desired. If the transmit buffer is NULL, then zeroes will be sent out of the shift register to receive data. If the reception pointer is NULL then the received data is discarded. Finally, the desired length of the transfer can be specified.

```
typedef struct bp_spi_tf {
    const void *p_tx_buf; // Transmit buffer.
    void *p_rx_buf; // Receive buffer.
    size_t len; // Length of the data to receive and/or transmit.
    bp_spi_async_cb_t callback; // Async transfer callback.
    void *p_ctxt; // Optional user context pointer.
} bp_spi_tf_t;
```

Listing 71 – SPI definition structure member details.

To perform a transfer the description structure must be passed to `bp_spi_xfer()` as shown in Listing 72. In a master SPI transfer the `p_tf_len` argument of `bp_spi_xfer()` can be set to NULL as it is not needed. For a master transfer the timeout value only means the time to wait for the interface to be available, it does not affect the transfer once started.

```
#include <spi/bp_spi.h>

int rtn;

// Transmit and receive buffers.
uint8_t tx_buf[8];
uint8_t rx_buf[8];

// Transfer description structure.
bp_spi_tf_t tf;

// Setup the transfer description structure.
// If only the received or transmitted data is important
// one of the buffer pointer can be set to NULL.
tf.p_tx_buf = tx_buf;
tf.p_rx_buf = rx_buf;
tf.len = 8u;

rtn = bp_spi_xfer(hndl, &tf, NULL, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 72 – Master SPI transfer example.

14.5 Asynchronous I/O

Apart from the blocking API the `spi` module also features a non-blocking asynchronous API. With the asynchronous API an application can start an SPI transfer without having to wait for it to complete afterward. Upon completion of the SPI asynchronous transfer, a user-specified callback is called from the SPI interrupt handler to signal the application that the transfer is finished and to report the result.

From within this callback that application can choose to either finish the transfer or start another asynchronous transfer immediately. This last feature can be very useful when transferring to and from a circular buffer.

Asynchronous reception and transmission is performed in a similar fashion to the blocking API described in the previous sections. The transfer description structure to be used is the same, however, when starting an asynchronous transfer it is important to specify a callback. The application can also set a user-defined pointer that will be passed unmodified to the callback when invoked.

To signal the application that a transfer is completed the SPI driver will call a user-specified callback similar to the one in Listing 73. Through the callback arguments, the application is informed of the status of the completed transfer and the amount of data transferred. A pointer to the transfer descriptor structure is also passed to the callback. With that description structure, the application code can access the original transfer parameters as well as the user context pointer. That user context pointer can be used to pass information from the application to the callback.

```
bp_spi_action_t spi_async_callback(int status, size_t tf_len, bp_spi_tf_t *p_tf)
{

    // Action to be undertaken by the SPI driver interrupt
    // handler upon return.
    return BP_SPI_ACTION_FINISH;
}
```

Listing 73 – SPI asynchronous transfer callback example.

When retuning from the callback, the application should use one of the two possible return values, namely `BP_SPI_ACTION_FINISH` or `BP_SPI_ACTION_RESTART`. If the callback returns with `BP_SPI_ACTION_FINISH`, as shown in Listing 73, the transfer is completed normally. If, however `BP_SPI_ACTION_RESTART` is returned, the transfer will be restarted with the information from the transfer descriptor. The application is allowed to modify the transfer descriptor through the `p_tf` argument prior to restarting the transfer.

When performing an asynchronous transfer the transfer can be considered to have been started when `bp_spi_xfer_async()` returns. It is important to keep in mind that, due to timing and context switch, the transfer may have already completed before returning.

14.5.1 Asynchronous Transfer

Listing 74 contains an example of asynchronous transfer. The callback function, as explained above isn't shown but should be defined somewhere. It's imperative to keep the buffer valid for the entire duration of the asynchronous transfer otherwise the transmitted data might be corrupted.

```
#include <spi/bp_spi.h>

int rtn;

// Transmit and receive buffers.
uint8_t tx_buf[8];
```

```
uint8_t rx_buf[8];

// Transfer description structure.
bp_spi_tf_t tf;

// Setup the transfer description structure.
tf.p_tx_buf = tx_buf;
tf.p_rx_buf = rx_buf;
tf.len = 8u;
tf.callback = spi_async_callback;
tf.p_ctxt = NULL;

rtn = bp_spi_xfer_async(hndl, &tf, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 74 – Master SPI asynchronous transfer example.

14.5.2 Aborting an Asynchronous Transfer

A running asynchronous transfer can be aborted by calling `bp_spi_xfer_async_abort()`. The abort function has two optional arguments which, if passed to the function, will return the number of bytes transferred and received at the moment the transfer was aborted. Note that it's possible for the transfer to complete while the abort function is called in which case the returned transfer length will be 0 and the abort function won't have any effect. Listing 75 contains an example of aborting an asynchronous SPI transfer.

```
#include <spi/bp_spi.h>

int rtn;

size_t rx_len;
size_t tx_len;

rtn = bp_spi_xfer_async_abort(hndl, &tx_len, &rx_len, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 75 – Master SPI asynchronous transfer example.

14.5.3 Waiting for an SPI Interface to be Idle

If an application wishes to wait for an SPI interface to be idle, this can be done by calling `bp_spi_idle_wait()`. Calling this function will wait for any blocking or asynchronous transfer to be completed before returning. In addition it will also wait for the physical transfer to be completed if possible. An example of calling `bp_spi_idle_wait()` is shown in Listing 76.

```
#include <spi/bp_spi.h>

int rtn;

rtn = bp_spi_idle_wait(hndl, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 76 – Waiting for an SPI interface to be idle.

14.6 Direct Access to the Driver

The spi module is designed to offer a strong set of features that covers the majority of use cases. At the same time, the set of features is also selected to ensure that it can be supported by a large majority of SPI peripherals. As such, it is natural that some peripheral specific features cannot be supported by the top-level portable API of the spi module. To alleviate this, the SPI drivers can implement driver specific functions available to the application to access advanced features of the underlying peripheral. To use them, however, the application needs to access the driver directly which is explained in this section.

Apart from accessing driver specific features, using the driver interface directly offers a slight improvement in performance since the call overhead is reduced. However it also means that access to the driver is not inherently thread safe as the top-level spi module is responsible for ensuring thread safety. Finally, for applications that want to reduce the RAM usage to a minimum, it is possible to instantiate an SPI driver by itself without having an instance of the top-level spi module. Further Information on the driver API can be found in the API Reference Manual.

14.6.1 Retrieving the Driver Handle

The first step in accessing a driver is to retrieve its handle. This can be performed using bp_spi_drv_hndl_get() as in Listing 77. In the example, if successful, the driver handle will be set in drv_hndl. That handle can then be used to access the driver API.

```
#include <spi/bp_spi.h>

int rtn;

// Variable that will receive the driver handle.
bp_spi_drv_hndl_t drv_hndl;

rtn = bp_spi_drv_hndl_get(hndl, &drv_hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 77 – Retrieving the driver handle from an SPI module instance.

14.6.2 Locking an SPI Module Instance

Most BASEplatform drivers are not inherently thread-safe. As such, when accessing a driver directly, care must be taken in a multitasking environment to prevent concurrent access to the same driver instance. This can be achieved in multiple ways. At the application level, one of the simplest ways is to ensure that the driver and `spi` module instance are only accessed from a single thread only. Another option is to implement a locking mechanism at the application level using a mutex or semaphore. This last method requires an extra kernel object, however.

With the `spi` module locking an instance is rather simple, it is sufficient to call `bp_spi_slave_sel()` to lock the instance and `bp_spi_slave_desel()` to release it. This can be used by the application to prevent other threads from accessing the module API while performing direct driver operations.

14.6.3 Calling a Driver API

Listing 78 shows a complete example of calling a driver specific API. For the purpose of the example the driver specific function `bp_imx_spi_loop_mode_en()` is called. This function is specific to the i.MX SPI driver and enables the peripheral loopback mode. When calling a driver directly, it is necessary to include that driver's header file, `bp_imx_spi_drv.h` in this case.

```
#include <spi/bp_spi.h>
#include <soc_comp/imx/bp_imx_spi_drv.hh>

int rtn;

// Variable that will receive the driver handle.
bp_spi_drv_hndl_t drv_hndl;

rtn = bp_spi_drv_hndl_get(hndl, &drv_hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }

rtn = bp_imx_spi_loop_mode_en(drv_hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 78 – Direct driver access example.

14.6.4 Alternative Driver Calling Method

In the previous section's example, a driver's function was called directly using its function name. For driver specific functions, which are non-standard this is the only way possible. However for the standard portion of the driver API, such as the lifecycle management functions and transfer functions, it is also possible to use the driver API structure. This structure is declared in the driver header file, for example to keep the same example of the i.MX SPI driver, the driver structure is called `g_bp_imx_spi_drv`. Using it would look like the example in Listing 79.

```
#include <spi/bp_spi.h>
#include <soc_comp/imx/bp_imx_spi_drv.hh>

int rtn;

rtn = g_bp_imx_spi_drv.en(drv_hndl, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 79 – Alternative direct driver access example.

14.7 Conclusion

This concludes the chapter on the `spi` module. The chapter covered the basics of using the `spi` module such as the lifecycle and basic transmission and reception. The chapter also went over the asynchronous API and how to access the SPI drivers directly. Readers are encouraged to read the `spi` module API reference for additional details on its usage and other advanced features of the API. BASEplatform users should also take a look at the Platform Reference Manual for their selected platform for additional information about their platform.

Chapter

15

Inter-Integrated Circuit (I2C)

15.1 Introduction

The `i2c` module is used to interface with other board components using the I2C bus. The I2C bus is a low speed two-wire half-duplex bus where each device on the bus uses open-drain I/O with pull-up resistors. Using this open-drain design the I2C bus can be used in a multi-master configuration although that is rare. Contrary to UART and SPI the I2C protocol supports multiple slaves using an addressing scheme comprising a 7 or 10-bit address. The I2C bus also allows the slave devices to wait before sending data, a technique known as clock stretching. This removes the strict timing constraint often seen when the master controls the data clock such as in SPI where a slave device has a very small window of time to prepare the data to send.

Considering the wide variety of I2C and I2C compatible peripherals, it would be impossible to design a high-level and portable API that could leverage the unique features of each and every peripheral. To alleviate this, drivers are allowed to implement driver-specific functionalities to extend the features of the core API of the `i2c` module. Details on how to access those features is described later in this chapter.

15.2 Overview

The `i2c` module features the usual I/O API of the BASEplatform including the standard lifecycle management API as well as transmission and reception. The module also offers the expected control over the I2C clock rate and choice of either master or slave configuration. The I/O API like the `uart` and `spi` module gives the application developer the choice between a blocking, polling and non-blocking asynchronous I/O API.

I2C drivers from the BASEplatform usually attempt to minimize context switches and unnecessary interrupts. While I2C communication is relatively low speed, every effort is made to reduce the CPU overhead of performing I2C communication.

15.3 Lifecycle

The I2C module follows the usual lifecycle of BASEplatform I/O modules. The various states and functions to move between those states are schematized in Figure 7.

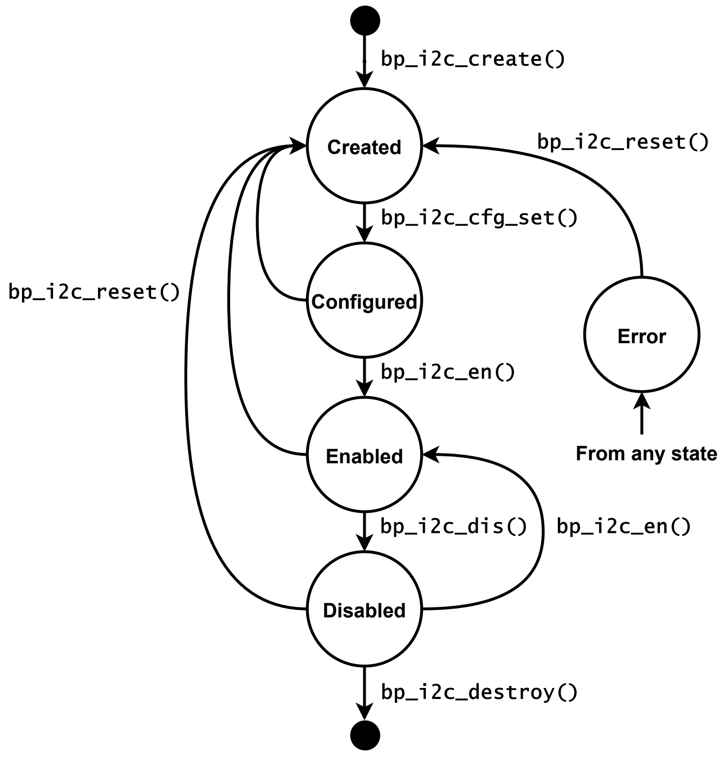

Figure 7 – State diagram of the `i2c` module's lifecycle.

To be useful a `i2c` module instance must first be created by calling `bp_i2c_create()`. Once created the module instance must be configured with `bp_i2c_cfg_set()` and then enabled using `bp_i2c_en()`. At this point the `i2c` module instance is ready to be used for transmission and reception. The module instance can then be disabled if desired by calling `bp_i2c_dis()` and finally destroyed to reclaim any resource associated with the module instance with `bp_i2c_destroy()`. It is also possible to reset a module instance using `bp_i2c_reset()`.

The following sections look at each step in more details.

15.3.1 Create

A new instance of the `i2c` module can be created using `bp_i2c_create()`. After being created successfully, the new instance will be left in the created state after which it should be configured and enabled before being used. An example of creating an `i2c` module instance is shown in Listing 80.

```
#include <i2c/bp_i2c.h>
#include <board/bp_board_def.h>

int rtn;

// Variable that will receive the newly created handle.
bp_i2c_hndl_t hndl;


rtn = bp_i2c_create(&g_i2c_board_def, &hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 80 – Creating a new I2C module instance.

The first argument to `bp_i2c_create()` is a pointer to the board definition structure of the I2C peripheral to associate with the new instance. These I2C definition structures are part of the board definition and usually accessible by including the `bp_board_def.h` header file. See Chapter 4 for more information on board definitions. The second argument to `bp_i2c_create()` is a pointer to the handle. If successful, it will be set to the newly created `i2c` module handle.

`bp_i2c_create()` performs all the necessary resource allocation required by the `i2c` module and associated I2C driver prior to returning. As such, once create an `i2c` module instance has all the necessary memory and kernel objects required for its entire lifetime. See Chapter 6 for additional details on memory allocation.

15.3.2 Configure

After being created an `i2c` instance should be configured after which it can finally be enabled. Configuration of an I2C peripheral is done by calling `bp_i2c_cfg_set()`. At runtime it is possible to call `bp_i2c_create()` again to update the confifugration if desired. At any time it is possible to query the current configuration with `bp_i2c_cfg_get()`.

If the `i2c` instance was just created, it will transition into the configured state if the call to `bp_i2c_cfg_set()` is successful. If the instance was already configured then the configuration is updated but the state of the instance is left unchanged unless a fatal error occurs.

An example of configuration is shown in Listing 81. In the example the instance is configured to run at 100 kHz in a master configuration. When running as a master the I2C address member of the configuration `slave_addr` is ignored and can be set to 0.

```
#include <i2c/bp_i2c.h>

int rtn;

// I2C configuration structure.
bp_i2c_cfg_t cfg;

// Populating the I2C configuration.
cfg.bit_rate = 100000u;
cfg.master = 1;
cfg.slave_addr = 0u; // Slave address ignored for master.

rtn = bp_i2c_cfg_set(hndl, &cfg, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 81 – Configuring an I2C module instance.

15.3.3 Enable

After having configured an `i2c` instance, it must then be enabled. This can be achieved with the `bp_i2c_en()` API function. It is also possible to enable an instance that was disabled previously. A successful call to `bp_i2c_en()` will place the module in the enabled state and it can then be used for transmission and reception. If the instance was already enabled then `bp_i2c_en()` should have no effect.

The exact side effects of enabling an `i2c` instance are specific to the driver and I2C peripheral. In most cases it will enable the I2C peripheral's clock and bring it out of reset if appropriate.

Listing 82 shows and example of enabling an `i2c` instance.

```
#include <i2c/bp_i2c.h>

int rtn;

rtn = bp_i2c_en(hndl, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 82 – Enabling an I2C module instance.

15.3.4 Disable

At runtime it is possible to disable an enabled `i2c` instance with the `bp_i2c_dis()` function. After being disabled, no other operations should be performed on the instance other than enabling it with `bp_i2c_en()` or resetting the instance using `bp_i2c_reset()`. Any other operation runs the risk of accessing a disabled peripheral which could cause a bus fault or hang. If assertion checks are enabled in the BASEplatform configuration (See Chapter 7) a fatal error may be returned when trying to access a disabled instance.

The exact side effects of disabling a `i2c` instance is driver and hardware specific. If possible, the device is placed in a low power disabled state and the peripheral's clock is disabled. Also it is important to note that the created and configured state are not equivalent to the disabled state, especially from a power consumption point of view. Application developers who wish to configure an I2C peripheral but leave it in a low power state should fully create and configure the instance and then call `bp_i2c_dis()` to disable the module instance and peripheral.

Listing 83 presents an example of disabling an `i2c` module instance.

```
#include <i2c/bp_i2c.h>

int rtn;

rtn = bp_i2c_dis(hndl, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 83 – Disabling an I2C module instance.

15.3.5 Destroy

At any point if a `i2c` instance is no longer needed, it can be destroyed using `bp_i2c_destroy()`. Destroying an instance will reclaim any resource assigned to the instance but only if freeing of memory is permitted by the primary memory allocator. For more details on allocation policy see Chapter 6. Once destroyed the instance handle becomes invalid and should not be used again.

It is important to first disable an `i2c` instance prior to destroying it otherwise the peripheral will be left active. Listing 84 contains an example of destroying an `i2c` module instance.

```
#include <i2c/bp_i2c.h>

int rtn;

// First disable the instance.
rtn = bp_i2c_dis(hndl, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }

rtn = bp_i2c_destroy(hndl, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 84 – Destroying an I2C module instance.

15.3.6 Reset

A `i2c` instance in any state, other than destroyed, can be reset with the `bp_i2c_reset()` function. Resetting an instance will bring it back to the created state. If possible, the I2C driver will try to perform a hardware soft reset on the peripheral if this is supported. This soft reset can either be at the peripheral level if there is a soft reset bit, or using a centralized peripheral reset controller if it is available on the current SoC.

Reset is the only operation that should be performed on a module instance that encountered an unexpected error, i.e. returned `RTNC_FATAL`. A reset could be attempted to return the internal state of the module and peripheral to a known state.

An example of reset is shown in Listing 85.

```
#include <i2c/bp_i2c.h>

int rtn;

rtn = bp_i2c_reset(hndl, TIMEOUT_INF);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 85 – Resetting an I2C module instance.

15.4 Basic Usage

This section will go over some example usage of the `i2c` module for master read and write. Since reading and writing are basically the same operation as far as the I2C bus is concerned the `i2c` module only has one transfer function `bp_i2c_xfer()` that can be used for reading and writing.

15.4.1 Master Transfer

Transferring data using the `i2c` module API is done through the use of a transfer description structure. The content of that structure is reproduced in Listing 86. Within the `bp_i2c_tf_t` structure two pointers, the `callback` and `p_ctxt` pointers are only useful when using the asynchronous I/O described later in this chapter. For a transfer using the blocking API they can be set to NULL or ignored.

When performing a transfer, the application must set the pointer to the buffer that contains or will receive the transferred data as well as the number of bytes to transfer. Then the transfer direction must be set to either `BP_I2C_DIR_TX` or `BP_I2C_DIR_RX`. Finally, the slave address should be set for a master transfer. Attempting to access an inexistent slave will return `RTNC_IO_ERR`.

As an additional feature the `hold_nack` member of the transfer description structure can be used in master mode to hold the bus after a transfer has completed. This prevents any other master from using the bus until it is released. This is useful when multiple transfers must be done in an atomic fashion without interference from other bus masters.

```
typedef struct bp_i2c_tf {
    bp_i2c_dir_t dir; // Transfer direction.
    bool hold_nack; // Hold or Nack after transfer.
    void *p_buf; // Data buffer to transmit or receive.
    uint16_t slave_addr; // Slave address.
    uint32_t buf_len; // Length of data to transmit or receive in bytes.
    bp_i2c_async_cb_t callback; // Async transfer callback.
    void *p_ctxt; // Optional user context pointer.
} bp_i2c_tf_t;
```

Listing 86 – I2C transfer description structure.

Once populated the structure should be passed to `bp_i2c_xfer()` to start the transfer. An example of this is depicted in Listing 87. When performing a master transfer the `p_tf_len` argument is unused and should be set to NULL. Also for a master transfer the timeout argument only means the length of time to wait for the interface to be free but doesn't dictate how much time it may take to perform the actual transfer.

```
#include <i2c/bp_i2c.h>

int rtn;

// Buffer to transmit.
uint8_t buf[8];

// Transfer description structure.
bp_i2c_tf_t tf;

// Transfer setup.
tf.p_buf = buf;
tf.buf_len = 8u;
tf.dir = BP_I2C_DIR_TX;
tf.slave_addr = 10u;
tf.hold_nack = false;

rtn = bp_i2c_xfer(hndl,  &tf, NULL, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 87 – Master I2C transfer example.

15.5 Asynchronous I/O

In addition to the blocking API described in the previous sections the `i2c` module also has an asynchronous I/O API that allows the application to start a transfer without blocking. Upon completion of the transfer, a user speficied callback is invoked from the I2C interrupt handler. The result of the operation is reported as an argument to the callback. From within the callback, the application can decide to either end the transfer or restart the transfer using different transfer settings. This last feature allows the application to transmit or receive continuously without intervention of a thread or the background task.

Launching an asynchronous transfer is very similar to using the blocking API. The main difference is that the application must pass a suitable callback to be called upon completion. This callback should be similar in form to the one in Listing 88. Through the callback arguments, the application is informed of the status of the completed transfer and the amount of data transferred. A pointer to the transfer descriptor structure is also passed to the callback. Within that description structure, the application code can access the original transfer parameters as well as the user context pointer. That user context pointer can be used to pass information from the application to the callback.

```
bp_i2c_action_t i2c_async_callback(int status, size_t tf_len, bp_i2c_tf_t *p_tf)
{

    // Action to be undertaken by the I2C driver interrupt
    // handler upon return.
    return BP_I2C_ACTION_FINISH;
}
```

Listing 88 – I2C asynchronous transfer callback.

When retuning from the callback, the application should use one of the two possible return values, namely `BP_I2C_ACTION_FINISH` or `BP_I2C_ACTION_RESTART`. If the callback returns with `BP_I2C_ACTION_FINISH`, as shown in Listing 88, the transfer is completed normally. If, however, `BP_I2C_ACTION_RESTART` is returned, the transfer will be restarted with the information from the transfer descriptor. The application is allowed to modify the transfer descriptor through the `p_tf` argument prior to restarting the transfer.

When performing an asynchronous transfer the transfer can be considered to have been started when `bp_i2c_xfer_async()` returns. It is important to keep in mind that, due to timing and context switch, the transfer may have already completed before returning.

15.5.1 Master Asyncrhronous Transfer

Listing 89 contains an example of asynchronous transfer. The callback function, as explained above isn't shown but should be defined somewhere. It's imperative to keep the buffer valid for the entire duration of the asynchronous transfer otherwise the transmitted data might be corrupted.

```c
#include <i2c/bp_i2c.h>

int rtn;

// Buffer to transmit.
uint8_t buf[8];

// Transfer description structure.
bp_i2c_tf_t tf;

// Transfer setup.
tf.p_buf = buf;
tf.buf_len = 8u;
tf.dir = BP_I2C_DIR_TX;
tf.slave_addr = 10u;
tf.hold_nack = false;
tf.callback = i2c_async_callback;
tf.p_ctxt = NULL;

rtn = bp_i2c_xfer_async(hndl,  &tf, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 89 – Master I2C asynchronous transfer example.

15.5.2 Aborting an Asynchronous Transfer

A running asynchronous transfer can be aborted by calling `bp_i2c_xfer_async_abort()`. The abort function has an optional argument which, if passed to the function, will return the number of bytes transferred at the moment the transfer was aborted. Note that it's possible for the transfer to complete while the abort function is called in which case the returned transfer length will be 0 and the abort function won't have any effect. Listing 90 contains an example of aborting an asynchronous I2C transfer.

```
#include <i2c/bp_i2c.h>

int rtn;

size_t tf_len;

rtn = bp_i2c_xfer_async_abort(hndl, &tf_len, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 90 – Master I2C asynchronous transfer abort example.

15.5.3 Waiting for an I2C Interface to be Idle

If an application wish to wait for an I2C interface to be idle, this can be done by calling `bp_i2c_idle_wait()`. Calling this function will wait for any blocking or asynchronous transfer to be completed before returning. In addition it will also wait for the physical transfer to be completed if possible. An example of calling `bp_i2c_idle_wait()` is shown in Listing 91.

```
#include <i2c/bp_i2c.h>

int rtn;

rtn = bp_i2c_idle_wait(hndl, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 91 – Waiting for an I2C interface to be idle.

15.6 Direct Access to the Driver

The `i2c` module is designed to offer a strong set of features that covers the majority of use cases. At the same time, the set of features is also selected to ensure that it can be supported by a large majority of SPI peripherals. As such, it is natural that some peripheral specific features cannot be supported by the top-level portable API of the `i2c` module. To alleviate this, the I2C drivers can implement driver specific functions made available to the application to access advanced features of the underlying peripheral. To use them, however, the application needs to access the driver directly which is explained in this section.

Apart from accessing driver specific features, using the driver interface directly offers a slight improvement in performance since the call overhead is reduced. However it also means that access to the driver is not inherently thread safe as the top-level `i2c` module is responsible for ensuring thread safety. Finally, for applications that want to reduce the RAM usage to a minimum, it is possible to instantiate an I2C driver by itself without having an instance of the top-level `i2c` module. Information on the driver API can be found in the API reference manual.

15.6.1 Retrieving the Driver Handle

The first step in accessing a driver is to retrieve its handle. This can be performed using `bp_i2c_drv_hndl_get()` as in Listing 92. In the example, if successful, the driver handle will be set in `drv_hndl`. That handle can then be used to access the driver API.

```
#include <i2c/bp_i2c.h>

int rtn;

// Variable that will receive the driver handle.
bp_i2c_drv_hndl_t drv_hndl;

rtn = bp_i2c_drv_hndl_get(hndl, &drv_hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 92 – Retrieving the driver handle from an I2C module instance.

15.6.2 Locking the I2C Module Instance

Most BASEplatform drivers are not inherently thread-safe. As such, when accessing a driver directly, care must be taken in a multitasking environment to prevent concurrent access to the same driver instance. This can be achieved in multiple ways. At the application level, one of the simplest ways is to ensure that the driver and `i2c` module instance are only accessed from a single thread only. Another option is to implement a locking mechanism at the application level using a mutex or semaphore. This last method requires an extra kernel object, however.

To simplify concurrent access to the driver and module instance from multiple threads, the `i2c` module API includes two functions, `bp_i2c_acquire()` and `bp_i2c_release()` to acquire and release the module instance mutexes. Acquiring the instance prevents any concurrent access from any thread trying to use the top level `i2c` module API. To work properly, however, this requires all the threads accessing the driver to use the acquire and release calls. It is also possible to use the top-level `i2c` module API while locked as the locking is recursive.

```
#include <i2c/bp_i2c.h>

int rtn;

rtn = bp_i2c_acquire(hndl, TIMEOUT_INF)
if(rtn != RTNC_SUCCESS) { /* Error management */ }

// Perform driver and i2c module operations here.

rtn = bp_i2c_release(hndl, TIMEOUT_INF)
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 93 – Acquire and Release of an I2C Instance.

15.6.3 Calling a Driver API

Listing 94 shows a complete example of calling a driver API including fetching the driver handler and locking the `i2c` module. For the purpose of the example, it is assumed that the Zynq I2C driver is being used, but it would work in a similar fashion with any other drivers. Note that to access the driver API it's necessary to include the driver's header file, `bp_zynq_i2c_drv.h` for this example.

```
#include <i2c/bp_i2c.h>
#include <soc_comp/zynq/zynq_i2c/bp_zynq_i2c_drv.h>

int rtn;

// Variable that will receive the driver handle.
bp_i2c_drv_hndl_t drv_hndl;

rtn = bp_i2c_drv_hndl_get(hndl, &drv_hndl);
if(rtn != RTNC_SUCCESS) { /* Error management */ }

rtn = bp_i2c_acquire(hndl, TIMEOUT_INF)
if(rtn != RTNC_SUCCESS) { /* Error management */ }

// Call the driver function directly. Note the usage of the driver handle.
rtn = bp_zynq_i2c_timeout_set(drv_hndl, 32u);
if(rtn != RTNC_SUCCESS) { /* Error management */ }

rtn = bp_i2c_release(hndl, TIMEOUT_INF)
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 94 – Direct driver access example.

15.6.4 Alternative Driver Calling Method

In the previous section's example, a driver's function was called directly using its function name. For driver specific functions, which are non-standard this is the only way possible. However for the standard portion of the driver API, such as the lifecycle management functions and transfer functions, it is also possible to use the driver API structure. This structure is declared in the driver header file, for example to keep the same example of the Zynq I2C driver, the driver structure is called g_bp_zynq_i2c_drv. Using it would look like the example in Listing 95.

```
#include <i2c/bp_i2c.h>
#include <soc_comp/zynq/bp_zynq_spi_drv.hh>

int rtn;

rtn = g_bp_zynq_i2c_drv.en(drv_hndl, TIMEOUT_INF);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 95 – Alternative direct driver access example.

15.7 Conclusion

This chapter went over the i2c module functionalities and basic usage examples. Including creating and configuring a new i2c module instance and various other lifecycle operations. The chapter also went over I2C transfer using both the blocking API and the non-blocking asynchronous API. Readers are encouraged to read the i2c module API reference for additional details on its usage and other advanced features of the API. BASEplatform users should also take a look at the Platform Reference Manual for their selected platform for addition information about their platform.

Chapter

16

General Purpose I/O (GPIO)

16.1 Introduction

The `gpio` module is used to interact with an MCU's General Purpose Input/Output (GPIO) peripheral. It can also be used to control external I/O expanders when coupled with a suitable driver. The module allows control and reading of the pins state and direction. The `gpio` module follows the same design as other low speed I/O modules such as the `uart` and `spi` modules. As such it contains most of the same lifecycle management functions as well as a dedicated API to control the GPIO pins.

Readers should note that the `gpio` module is only concerned with control over the pin state and direction. The module does not, however, have any control over the pin multiplexing configuration (a.k.a pin mux) nor does it have control over the pad configuration such as pull-up and drive strength configuration. Instead, a separate platform specific module is provided for each MCU or SoC thus giving the maximum degree of control over the mux configuration. These platform specific modules are documented in the Platform Reference Mannual for the device in question.

16.2 Overview

The `gpio` module is similar to other BASEplatform I/O modules but has a few idiosyncrasies specific to itself. The module's lifecycle management functions are similar to the others, however, it lacks the configuration step. The I/O API is thead-safe, as the other I/O modules, but in addition it is usually non-blocking when interacting with the GPIO pins belonging to the MCU. However there are some exceptions to that last statement, specifically when the GPIO module is used to access an external I/O expander. This situation is described later in this chapter. It is also important to remember that the initialization sequence of pin muxing, pad configuration and finally GPIO configurations are very platform specific. Care should be taken to study the documentation as well as the example provided with the BASEplatform for each platform.

To improve performance and reduce the overhead of accessing the GPIO pins, the BASEplatform GPIO drivers are designed to be non-blocking while still being thread-safe. This means that the API functions of the `gpio` modules do not take a timeout value as argument as is common with other BASEplatform modules. Another reason to make the GPIO API non-blocking is to permit them to be used from within critical sections and interrupt service routines.

16.2.1 Effect of Disabling or Resetting the GPIO Module

Unlike other communication peripherals the GPIO peripheral can be central to the operation of a SoC. Resetting or disabling it could affect the state of external pins and on some platform even disable pins configured to alternate peripheral functions other than GPIO. As such it is not recommended to disable or reset the GPIO module at runtime unless the side-effects are properly understood. Even then, in many cases the GPIO driver may not have the ability to disable or reset the GPIO peripheral if that involves too many subsystems.

16.2.2 Using the GPIO Module to Interface With I/O Expanders

The GPIO module and most GPIO drivers are non-blocking making them more efficient since the time it takes to set and configure GPIO pins is quite short. Due to that last statement, the overhead of using a mutex appears superfluous. However, when using the GPIO module with an I/O expander driver care should be taken as the driver may be blocking. This is definitely the case when accessing an SPI or I2C expander which requires the use of a blocking communication API to interact with the expander. Even simple expander drivers based on a shift register may prefer to use a semaphore or mutex in order not to disable interrupts for too long. Consequently, while the GPIO API can be used from within critical sections and interrupt handler this should not be attempted when interfacing with an I/O expander.

16.2.3 Pin and Bank Numbering

The `gpio` API is designed to be cross-platform, however, not every platform follows the same scheme when it comes to numbering their GPIO. For example, some will use bank numbers while others will not. The BASEplatform attempts to follow the manufacturer's numbering when possible. This means that the signification of bank and pin numbers passed to the `gpio` module API is platform specific. Additional details can be found within each platform's Reference Manual.

16.3 Lifecycle

The `gpio` module lifecycle is similar to many other BASEplatform modules albeit being a little simpler due to the lack of configuration step. The various states and functions to move between those states are schematized in Figure 8.

To be used a `gpio` module instance must first be created by calling `bp_gpio_create()`. Once created the module instance must be enabled with `bp_gpio_en()`. At this point the `gpio` module instance is ready to be used. The module instance can then be disabled if desired by calling `bp_gpio_dis()` and finally destroyed to reclaim any resource associated with the module instance with `bp_gpio_destroy()`. It is also possible to reset an instance using `bp_gpio_reset()`.

The following sections look at each step in more details.

16.3.1 Create

Like all I/O modules, the `gpio` module must first be created. This can be achieved by calling `bp_gpio_create()`. After being created the new instance will be in the created state and must then be enabled prior to being used. As mentioned multiple time in this chapter the `gpio` module doesn't have a configuration step. An example of creating a `gpio` module instance is shown in Listing 96.

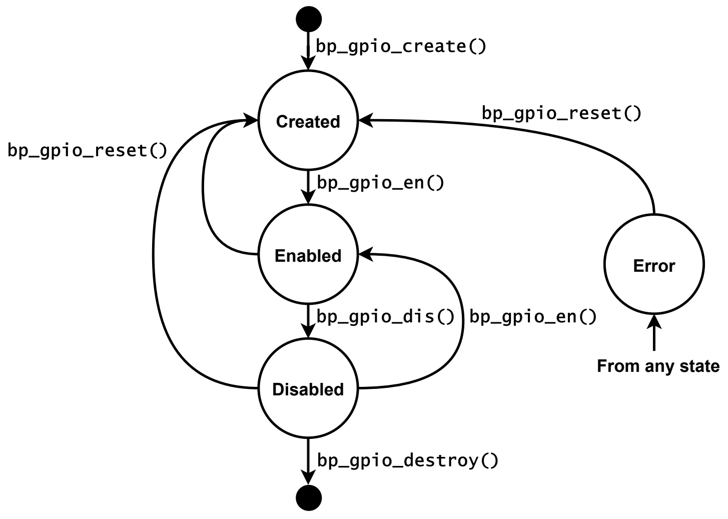

Figure 8 – State diagram of the `gpio` module's lifecycle.

```c
#include <gpio/bp_gpio.h>
#include <board/bp_board_def.h>

int rtn;

// Variable that will receive the newly created handle.
bp_gpio_hndl_t hndl;


rtn = bp_gpio_create(&g_gpio_board_def, &hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 96 – Creating a new GPIO module instance.

The first argument to `bp_gpio_create()` is a pointer to the board definition structure of the GPIO peripheral to associate with the new instance. These GPIO definition structures are part of the board definition and usually accessible by including the `bp_board_def.h` header file. See Chapter 4 for more information on board definitions. The second argument to `bp_gpio_create()` is a pointer to the handle. If successful, it will be set to the newly created `gpio` module handle.

`bp_gpio_create()` performs all the necessary resource allocation required by the `gpio` module and associated GPIO driver prior to returning. As such, once create an `gpio` module instance has all the necessary memory and kernel objects required for its entire lifetime. See Chapter 6 for additional details on memory allocation.

16.3.2 Enable

After having created a `gpio` instance it must then be enabled. This can be done using the `bp_gpio_en()` API function. It is also possible to enable an instance that was disabled previously. A successful call to bp_gpio_en() will place the module in the enabled state and it can then be used normally. If the instance was already enabled then bp_gpio_en() should have no effect.

The exact side effects of enabling a gpio instance is specific to the driver and GPIO peripheral. In most cases it will enable the peripheral's clock and bring it out of reset if appropriate. Listing 97 shows and example of enabling a gpio instance.

```
#include <gpio/bp_gpio.h>

int rtn;

rtn = bp_gpio_en(hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 97 – Enabling an GPIO module instance.

16.3.3 Disable

At runtime it is possible to disable an enabled gpio instance with the bp_gpio_dis() function. Although as mentioned earlier in this chapter it may not be advisable to disable a GPIO peripheral. After being disabled, no other operations should be performed on the instance other than enabling it with bp_gpio_en() or resetting the instance using bp_gpio_reset(). Any other operation runs the risk of accessing a disabled peripheral which could cause a bus fault or hang. If assertion checks are enabled in the BASEplatform configuration (See Chapter 7) a fatal error may be returned when trying to access a disabled instance.

The exact side effects of disabling a gpio instance is driver and hardware specific. If possible, the device is placed in a low power disabled state and the peripheral's clock is disabled. Also it is important to note that the created state is not equivalent to the disabled state, especially from a power consumption point of view. Application developers who wish to create a GPIO peripheral but leave it in a low-power state should fully create the instance and then call bp_gpio_dis() to disable the module instance and peripheral.

Listing 98 presents an example of disabling a gpio module instance.

```
#include <gpio/bp_gpio.h>

int rtn;

rtn = bp_gpio_dis(hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 98 – Disabling an GPIO module instance.

16.3.4 Destroy

At any point if a gpio instance is no longer needed, it can be destroyed using bp_gpio_destroy(). Destroying an instance will reclaim any resource assigned to the instance but only if freeing of memory is permitted by the primary memory allocator. For more details on allocation policy see Chapter 6. Once destroyed the instance handle becomes invalid and should not be used again.

It is not recommended to destroy the primary `gpio` module instance that controls the MCU's pins.

```
#include <gpio/bp_gpio.h>

int rtn;

rtn = bp_gpio_destroy(hndl);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 99 – Destroying a GPIO module instance.

16.3.5 Reset

A `gpio` instance in any state, other than destroyed, can be reset with the `bp_gpio_reset()` However it is not recommended to reset the MCU's GPIO peripheral to prevent corruption of the pin state.

An example of reset is shown in Listing 100.

```
#include <gpio/bp_gpio.h>

int rtn;

rtn = bp_gpio_reset(hndl);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 100 – Resetting a GPIO module instance.

16.4 Usage

The `gpio` module can be used to perform one of three things, set the direction of a GPIO pin, read a GPIO pin or set the state of a GPIO pin. This section will cover basic examples of each operation.

16.4.1 Setting the Direction of a GPIO Pin

Setting the direction of a GPIO pin can be done with `bp_gpio_dir_set()` while it is possible to query the direction of a pin with `bp_gpio_dir_get()`. The direction must be set to either to `BP_GPIO_DIR_IN` or `BP_GPIO_DIR_OUT`. Listing 101 show an example of setting the direction of pint 5 of bank 0 as an output pin.

```c
#include <gpio/bp_gpio.h>

int rtn;

rtn = bp_gpio_dir_set(hndl, 0u, 5u, BP_GPIO_DIR_OUT);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 101 – Setting the direction of a GPIO pin.

16.4.2 Setting The State of a GPIO Pin

The state of a GPIO pin can be set using the `bp_gpio_data_set()` function. In Listing 102 an example of setting a GPIO output to high, or 1 is shown. Note that the function will succeed and attempt to set the state of the pin even when the GPIO pin is configured as an input pin. What happens in this case depends on the GPIO peripheral. In many cases the state will be saved and the GPIO pin will take that value if the GPIO direction is changed to be an output pin.

```c
#include <gpio/bp_gpio.h>

int rtn;

// Set pin 5 of bank 0 to high.
rtn = bp_gpio_data_set(hndl, 0u, 5u, 1u);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 102 – Setting the state of a GPIO pin.

It's also possible to toggle the state of a gpio pin as in Listing 103. This will flip the state from 0 to 1 or 1 to 0.

```c
#include <gpio/bp_gpio.h>

int rtn;

rtn = bp_gpio_data_tog(hndl, 0u, 5u);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 103 – Toggling the state of a GPIO pin.

16.4.3 Reading The State of a GPIO Pin

Reading a GPIO pin is achieved by calling `bp_gpio_data_get()`. The GPIO driver will return the actual pin state if supported. On some platforms it means that the state of the pin may be read even if the pin is configured as an alternate peripheral other than a GPIO.

```
#include <gpio/bp_gpio.h>

int rtn;

// Variable that will receive the GPIO pin state.
uint32_t data;

rtn = bp_gpio_data_read(hndl, 0u, 5u, &data);
if(rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 104 – Reading the state of a GPIO pin.

16.5 Direct Access to the Driver

Like other I/O modules, it is possible to call a GPIO driver directly. This allows the application to access driver specific features as well as slightly reduce the call overhead. In the case of other I/O drivers such as UART and I2C, accessing the driver API directly meant that the call wasn't thread-safe as the top-level module provided the necessary mutex or semaphore. The GPIO drivers are slightly different in that they are usually thread safe when accessing an MCU's GPIO peripheral. An exception to this rule is when accessing an external I/O expander in that case care should be taken.

An additional difference between the GPIO drivers and other I/O drivers is that GPIO drivers for the MCU or SoC GPIOs can be called with a NULL handle. Again this is offered as an option to slightly reduce the overhead of controlling GPIO pins and also because there is usually only one instance of the platform's GPIO driver. Note that again if accessing a GPIO driver for an external I/O expander then it is necessary to use the instance's driver handle.

16.5.1 Retrieving the Driver Handle

Retrieving the driver handle of a `gpio` instance can be performed using `bp_gpio_drv_hndl_get()` as in Listing 105. In the example, if successful, the driver handle will be set in `drv_hndl`. That handle can then be used to access the driver API.

```
#include <gpio/bp_gpio.h>

int rtn;

// Variable that will receive the driver handle.
bp_gpio_drv_hndl_t drv_hndl;

rtn = bp_gpio_drv_hndl_get(hndl, &drv_hndl);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 105 – Retrieving the driver handle from an GPIO module instance.

16.5.2 Calling a Driver API

Calling a driver API can be done using the driver handle and by including the driver header file. Listing 106 shows an example of calling a driver API directly. For the purpose of the example, the intel FPGA GPIO driver data toggle function is used, the call would be similar with other drivers.

```
#include <gpio/bp_gpio.h>
#include <soc_comp/intel_fpga/hps_gpio/bp_hps_gpio_drv.h>

int rtn;

rtn = bp_hps_gpio_drv_data_tog(drv_hndl, 0u, 5u);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 106 – Calling a GPIO driver directly.

In the previous example of Listing 106 the instance handle was used but since the driver in question is the SoC's GPIO peripheral driver it is also possible to use a NULL handle instead as in Listing 107. A NULL handle is provided in the `BP_GPIO_NULL_HNDL` preprocessor macro.

```
#include <gpio/bp_gpio.h>
#include <soc_comp/intel_fpga/hps_gpio/bp_hps_gpio_drv.h>

int rtn;

rtn = bp_hps_gpio_drv_data_tog(BP_GPIO_NULL_HNDL, 0u, 5u);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 107 – Calling a GPIO driver directly with a NULL handle.

16.5.3 Alternative Driver Calling Method

In the previous section's example, a driver function was called directly using its function name. For driver specific functions, which are non-standard this is the only way possible. However for the standard portion of the driver API, such as the lifecycle management functions and transfer functions, it is also possible to use the driver API structure. This structure is declared in the driver header file, for example to keep the same example as the last one, the driver structure is called `g_bp_hps_gpio_drv`. Using it would look like the example in .

```
#include <gpio/bp_gpio.h>
#include <soc_comp/intel_fpga/hps_gpio/bp_hps_gpio_drv.h>

int rtn;

rtn = g_bp_hps_gpio_drv.data_tog(BP_GPIO_NULL_HNDL);
if (rtn != RTNC_SUCCESS) { /* Error management */ }
```

Listing 108 – Direct driver access example.

16.6 Conclusion

This concludes the chapter on the `gpio` module. The chapter went over creating and managing a `gpio` module instance as well as reading, configuring and setting the state of GPIO pins. It was also highlighted that the `gpio` module has a few unique features compared to other I/O modules. Namely that the primary interface is non-blocking and can be used from interrupt service routines and critical sections. Readers are encouraged to read the `gpio` module API reference for additional details on its usage and other advanced features of the API. BASEplatform users should also take a look at the Platform Reference Manual for their selected platform for addition information about their platform.

Chapter 17

API Reference Manual

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`

<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_ADDR_SZ`

<bp_arch.h>

Defined by the architecture port to the size of the addresses in bytes. Usually set to 4 on a 32-bit platform and 8 on a 64-bit platform.

Macro

`BP_ARCH_ALIGN_MAX`

<bp_arch.h>

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

Macro

BP_ARCH_COMPILER

<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()

<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

<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()

<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

<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()

<bp_arch.h>

core's interrupts are disabled. The result can be assigned to a variable of type 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()

<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()`

<bp_arch.h>

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

Macro

`BP_ARCH_IS_INT()`

<bp_arch.h>

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

Macro

`BP_ARCH_IS_INT_OR_CRIT()`

<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()`

<bp_arch.h>

Memory barrier.

Macro

`BP_ARCH_PANIC()`

<bp_arch.h>

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

Macro

`BP_ARCH_RMB()`

<bp_arch.h>

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

Macro

`BP_ARCH_SEV()`

<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()`

<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()`

<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()`

<bp_arch.h>

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

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()`

<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()

<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()

<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()

<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()

<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()`

<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()`

<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
✗	✓	✓	✓

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()`

<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()`

<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()

<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()

<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()`

<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()`

<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()`

<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()

<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

<bp_slock.h>

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

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()`

<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()`

<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()`

<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()

<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()

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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()

<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()

<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()`

<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()`

<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()`

<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()`

<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()

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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`

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()`

<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()

<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()

<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()

<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()

<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()`

<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()`

<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()

<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()

<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()

<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()`

<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()`

<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()

<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

<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

<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

<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.

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_core_freq_get()`

<bp_clock.h>

Returns the current clock frequency of the CPU core if known. If the core frequency is unknown or cannot be determined 0 is returned.

Prototype `uint32_t bp_clock_core_freq_get ( );`

Attributes

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

Returned Values Core clock frequency if known, 0 otherwise.

Function

`bp_clock_dis()`

<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()`

<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()

<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()

<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()

<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()

<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`

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()`

<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()`

<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()

<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()

<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`

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()

<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()

<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()

<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()

<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()

<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()

<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()

<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()

<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()

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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`

<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.
`BP_INT_TYPE_NULL`	Special invalid value.

Data Type

`bp_int_handler_t`

<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

<bp_int.h>

Special invalid interrupt value.

Macro

BP_INT_TYPE_IS_VALID()

<bp_int.h>

Checks if an interrupt type value is valid.

Expansion `true` if the interrupt type value is valid. `false` otherwise.

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()

<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()

<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()

<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`

MEM

The memory module provides the application and other BASEplatform modules a variety of memory management functions. The operations are centered around memory allocators which can be defined at runtime with various memory regions and attributes. The different choice of allocators can be used to decide the memory management policy for the entire application. For example the default heap allocator does not support freeing memory for use with safety critical application or to prevent fragmentation issues. In contrast to the heap allocator, the c_malloc allocator will use the underlying C library malloc and free function which can also be set as the default allocator so that the entire application use the C malloc heap.

Unless otherwise specified all allocators are non-blocking and thread safe, even when running under an SMP RTOS. Although most of the allocators are atomic and non-blocking memory allocation and freeing should not be performed from an interrupt service routine.

One allocator should be defined and configured as the default allocator with `bp_mem_alloc_dflt_set()`. This should be done very early in the system startup before any BASEplatform objects are created.

Function

`bp_mem_alloc()`

<bp_mem.h>

Allocates memory from the default allocator.

Prototype

```
void *  bp_mem_alloc  ( size_t  size,
                        size_t  align,
                                p_mem );
```

Attributes

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

Parameters

`size`	Amount of memory to allocate in bytes.
`align`	Alignment of the allocated memory block.
`p_mem`	Pointer to the allocated memory.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

`bp_mem_alloc_create()`

<bp_mem.h>

Creates a new memory allocator from the definition `p_def`. If successful a handle to the newly created allocator will be returned through `p_hndl`. See `bp_mem_alloc_def` for details of the available configurations.

Prototype

```
int  bp_mem_alloc_create  ( const bp_mem_alloc_def_t *  p_def,
                                  bp_mem_alloc_hndl_t *      p_hndl );
```

Attributes

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

Parameters

`p_def`	Allocator's definition.
`p_hndl`	Pointer to the created handle.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

bp_mem_alloc_destroy()

<bp_mem.h>

Destroys a memory allocator. Not all allocators support being destroyed, if it's not supported `RTNC_NOT_SUPPORTED` is returned and the allocator is left unaffected.

Destroying an allocator is usually only useful when an application needs to temporarily partition a region of memory. Destroying the default allocators is not recommended.

Prototype

```
int  bp_mem_alloc_destroy  ( bp_mem_alloc_hndl_t  hndl );
```

Attributes

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

Parameters

`hndl`	Handle of the allocator to destroy.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_SUPPORTED`
`RTNC_FATAL`

Function

bp_mem_alloc_dflt_get()

<bp_mem.h>

Retrieves the default memory allocator if one is set, otherwise a null handle is retururned through `p_hndl` and `RTNC_NOT_FOUND` is returned as the function return code.

Prototype

```
int  bp_mem_alloc_dflt_get  (  hndl );
```

Attributes

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

Parameters `hndl` Pointer to the returned allocator handle.

Returned Errors
`RTNC_SUCCESS`
`RTNC_NOT_FOUND`
`RTNC_FATAL`

Function

`bp_mem_alloc_dflt_set()`

<bp_mem.h>

Sets the default memory allocator to be used when calling `bp_mem_alloc()` and `bp_mem_free()`. Once a default allocator is set it can't be set again, calling `bp_mem_alloc_dflt_set()` with an allocator set will return `RTNC_FATAL`.

Prototype

```
int  bp_mem_alloc_dflt_set  ( bp_mem_alloc_hndl_t  hndl );
```

Attributes

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

Parameters `hndl` Handle of the allocator to use by default.

Returned Errors
`RTNC_SUCCESS`
`RTNC_FATAL`

Function

`bp_mem_alloc_from()`

<bp_mem.h>

Allocates memory from the a specific allocator.

Prototype

```
void *  bp_mem_alloc_from  ( bp_mem_alloc_hndl_t  hndl,
                             size_t               size,
                             size_t               align,
                                                  p_mem );
```

Attributes

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

Parameters	`hndl`	Handle of the allocator to use.
	`size`	Amount of memory to allocate in bytes.
	`align`	Alignment of the allocated memory block.
	`p_mem`	Pointer to the allocated memory.

Returned Errors
`RTNC_SUCCESS`
`RTNC_NO_RESOURCE`
`RTNC_FATAL`

Function

bp_mem_free()

<bp_mem.h>

Frees a previously allocated block of memory to the default allocator. The memory will be returned to the allocator if supported, otherwise `RTNC_NOT_SUPPORTED` is returned and the previously allocated memory is unaffected.

Prototype

```
int  bp_mem_free  ( void *  p_mem );
```

Attributes

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

Parameters	`p_mem`	Pointer to the memory to free.

Returned Errors
`RTNC_SUCCESS`
`RTNC_NOT_SUPPORTED`
`RTNC_FATAL`

Function

bp_mem_free_from()

<bp_mem.h>

Frees a previously allocated block of memory to a specific allocator if supported. The memory will be returned to the allocator if supported, otherwise `RTNC_NOT_SUPPORTED` is returned and the previously allocated memory is unaffected.

Prototype

```
int  bp_mem_free_from  ( bp_mem_alloc_hndl_t  hndl,
                         void *               p_mem );
```

Attributes

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

Parameters	`hndl`	Handle of the allocator to use.
	`p_mem`	Pointer to the memory to free.

Returned Errors	`RTNC_SUCCESS`
	`RTNC_NOT_SUPPORTED`
	`RTNC_FATAL`

Function

`bp_mem_lock_acquire()`

<bp_mem.h>

Acquires an exclusive lock to the memory allocator.

This function is for internal use only.

Prototype

```
int bp_mem_lock_acquire ( bp_mem_alloc_hndl_t hndl,
                                              p_mem );
```

Attributes

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

Parameters

`hndl`	Handle of the allocator to use.
`p_mem`	Pointer to the memory to free.

Returned Errors	`RTNC_SUCCESS`
	`RTNC_NOT_SUPPORTED`
	`RTNC_FATAL`

Function

`bp_mem_lock_release()`

<bp_mem.h>

Releases an exclusive lock to the memory allocator.

This function is for internal use only.

Prototype

```
int bp_mem_lock_release ( bp_mem_alloc_hndl_t hndl,
                                              p_mem );
```

Attributes

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

Parameters

`hndl`	Handle of the allocator to use.
`p_mem`	Pointer to the memory to free.

Returned Errors	`RTNC_SUCCESS`
	`RTNC_NOT_SUPPORTED`
	`RTNC_FATAL`

Data Type

bp_mem_alloc_def_t

<bp_mem.h>

Allocator definition structure. Used to describe the type and parameters of a memory allocator with bp_mem_alloc_create(). Some of the parameters may be interpreted differently or ignored by certain types of memory allocator.

As an exception compared to other BASEplatform modules, the definition structure itself is not needed after creation. As such, the memory occupied by the definition structure can be reused after the creation of a memory allocator.

Members

p_name	const char *	Name of the memory allocator.
p_drv	bp_mem_alloc_drv_t *	Pointer to the memory allocator driver.
p_base_addr	void *	Base address.
size	size_t	Size of the memory region starting from the base address.
self_contained	bool	Set to true if the allocator's internal data should be located within the allocator's own memory region.
p_ext_def	void *	Pointer to additional, allocator specific, definitions.

Data Type

bp_mem_alloc_drv_t

<bp_mem.h>

Data Type

bp_mem_alloc_hndl_t

<bp_mem.h>

Allocator handle. Memory allocator handle returned by bp_mem_alloc_create(). The pointer contained in the handle is private and should not be accessed by calling code.

Members

p_hndl	bp_mem_alloc_inst_t *	Pointer to the internal memory allocator's data.

Data Type

bp_mem_alloc_inst_t

<bp_mem.h>

Internal memory allocator instance data. This structure is exposed to the application through a of type bp_mem_alloc_hndl_t returned from bp_mem_alloc_create(). It should not be manipulated directly by the application.

Members

`p_name`	`const char *`	Allocator's instance name.
`p_base_addr`	`void *`	Base address.
`p_cur_addr`	`void *`	Highest allocated address, when relevent.
`size`	`size_t`	Total size of the memory region when known.
`wasted_size`	`size_t`	Wasted size statistics used by some allocators.
`lock`	`bp_slock_t`	Memory allocator's lock.
`rlock`	`bp_slock_t`	Memory allocator's reader lock.
`locked_task_id`	`uint32_t`	Task id of the lock owner.
`lock_count`	`uint32_t`	Lock nesting count.
`irqflag`	`bp_irq_flag_t`	Save irq flag.
`p_drv`	`bp_mem_alloc_drv_t *`	Memory allocator's driver.
`p_ext_data`	`void *`	Memory allocator's driver data.

Macro

BP_MEM_ALLOC_HNDL_IS_NULL()

<bp_mem.h>

Evaluates if a memory allocator handle is NULL.

Prototype	`BP_MEM_ALLOC_HNDL_IS_NULL ( hndl );`
Parameters	`hndl` Handle to be checked.
Expansion	`true` if the handle is NULL, `false` otherwise.

Macro

BP_MEM_NULL_HNDL

<bp_mem.h>

NULL allocator handle.

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()

<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()

<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()

<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()

<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()

<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()

<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()

<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()`

<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()`

<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()`

<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()`

<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()

<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()

<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

<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

<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

<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

<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

<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()`

<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`

<bp_gpio.h>

NULL GPIO module handle.

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()`

<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()`

<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()`

<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()

<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()

<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()

<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()

<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()

<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()`

<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()`

<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()`

<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()`

<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()

<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()

<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()

<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()

<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()

<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()

<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()`

<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`

<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`

<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`

<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

<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

<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

<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

<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

<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`

<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`

<bp_i2c.h>

10-bit I2C address mask.

Macro

`BP_I2C_HNDL_IS_NULL()`

<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`

<bp_i2c.h>

Highest 10-bit I2C address.

Macro

`BP_I2C_MAX_SLV_ADDR`

<bp_i2c.h>

Highest 7-bit I2C address.

Macro

`BP_I2C_MIN_10B_SLV_ADDR`

<bp_i2c.h>

Lowest 10-bit I2C address.

Macro

`BP_I2C_NULL_HNDL`

<bp_i2c.h>

NULL I2C handle.

Macro

`BP_I2C_SLV_ADDR_MASK`

<bp_i2c.h>

7-bit I2C address mask.

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()`

<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()`

<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()

<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()`

<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()

<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()

<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()

<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()

<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()

<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()

<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()

<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()

<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()

<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()

<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()`

<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()

<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()`

<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`

<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`

<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

<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

<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

<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

<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

<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

<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()

<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

<bp_spi.h>

NULL SPI module handle.

Macro

BP_SPI_SS_NONE

<bp_spi.h>

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

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()`

<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()`

<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()

<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 that is less than the requested 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()

<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()

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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()`

<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()

<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()`

<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()

<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()

<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()

<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()`

<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()

<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()

<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()

<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`

<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`

<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`

<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

<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

<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

<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

<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_drv_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

<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

<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

<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_ACTION_IS_VALID()

<bp_uart.h>

Checks if UART stop bits configuration value is valid.

Expansion `true` if the stop bits configuration value is valid. `false` otherwise.

Macro

BP_UART_HNDL_IS_NULL()

<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`

<bp_uart.h>

NULL UART handle.

Macro
`BP_UART_PARITY_IS_VALID()`

<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()`

<bp_uart.h>

Checks if UART stop bits configuration value is valid.

Checks if UART stop bits value is valid.

Expansion `true` if the stop bits configuration value is valid. `false` otherwise.
`true` if the stop bits value is valid. `false` otherwise.

Macro
`BP_UART_STOP_BITS_IS_VALID()`

<bp_uart.h>

Checks if UART stop bits configuration value is valid.

Checks if UART stop bits value is valid.

Expansion `true` if the stop bits configuration value is valid. `false` otherwise.
`true` if the stop bits value is valid. `false` otherwise.

Storage Media

The storage media module, or media module for short, is used to access multiple types of non-volatile storage and storage peripherals using a unified interface. The media module can support, through underlying drivers, bare flash memory (NOR, NAND etc...), managed flash memory (SDCard, eMMC, SSD) and others including a RAMdisk driver. The media interface API can be accessed directly, enabling the application to perform raw byte-level accesses to the media. It can also be used by any of the file systems supported by the BASEplatform.

To support the individual intricacies of each type of storage media, the initialization of a media module instance follows a different convention compared to most other BASEplatform modules. The media module does not have the usual create and configuration functions, instead the media driver's create function should be called which will return a media handle. Following the call to the driver specific create, the driver's configuration function should be called to setup the various configurations specific to that type of media. For example, to initialize a NOR Flash memory `bp_qspi_mem_create()` would be called followed by `bp_qspi_mem_cfg_set()` to configure the NOR speed, width and other parameters specific to QSPI memories. Following those steps the generic media API can be used starting with `bp_media_en()` to enable the newly created media.

The media module API is byte oriented giving maximum versatility when using byte addressable memory. However, some media may have restrictions to the size and alignment of read, write and erase operations. When interfacing with such a media, care should be taken to make sure the size and address of the operation is a multiple of the minimum operation size. `bp_media_prop_get()` can be called to query the minimum read, write and erase sizes of the underlying media.

Function

bp_media_dis()

<bp_media.h>

Disables a media instance. Disabling a media instance can have different side effects depending on the underlying media driver. This can include disabling the peripheral clock and putting the external memory device in a low power mode.

The result of calling `bp_media_dis()` or any other functions other than `bp_media_en()` or `bp_media_reset()` on an already disabled media 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_media_is_en()`.

Prototype

```
int  bp_media_dis  ( bp_media_hndl_t  hndl,
                     uint32_t         timeout_ms );
```

Attributes

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

Parameters

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

Returned Errors

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

Function

bp_media_en()

<bp_media.h>

Enables a media instance. Enabling a media instance can have different side effects depending on the media driver. This can include activating the peripheral clock, deassert reset and possibly wake up or enable the external memory device if relevant.

Calling bp_media_en() on an enabled media instance should be without side effect.

Prototype

```
int  bp_media_en  ( bp_media_hndl_t  hndl,
                    uint32_t         timeout_ms );
```

Attributes

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

Parameters

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

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_IO_ERR
RTNC_FATAL

Function

bp_media_erase()

<bp_media.h>

Erases len bytes starting from media address addr. While the API accepts any addresses and length at the byte granularity, it is the caller's responsibility to observe the erase block size restriction of the underlying media. bp_media_prop_get() can be called to query a the minimum erase block size of a media instance.

The timeout value is the amount of time to wait for the media to be available. The time spent doing the erase operation is not counted to consider a timeout condition. In case RTNC_TIMEOUT is returned it is guaranteed that no data has been erased from the device.

Drivers are allowed to use an internal timeout, independent of the timeout_ms argument, to detect a stuck peripheral when an erase operation is taking longer than expected. An RTNC_FATAL or RTNC_IO_ERR error is returned in those cases, see the driver's documentation for details.

It is unspecified how much data, if any, was actually erased from a failed transfer when RTNC_FATAL or RTNC_IO_ERR is returned.

Prototype

```
int  bp_media_erase  ( bp_media_hndl_t  hndl,
                       uint64_t         addr,
                       uint64_t         len,
                       uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the media module instance to erase.
`addr`	Media start address to erase.
`len`	Length of the area to erase.
`timeout_ms`	Timeout value in millisecond

Returned Errors

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

Function

`bp_media_is_en()`

<bp_media.h>

Returns the enabled/disabled state of a media instance. If the call is successful the state of the media is returned through the argument `p_is_en`.

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

Prototype

```
int  bp_media_is_en  ( bp_media_hndl_t  hndl,
                       bool *           p_is_en );
```

Attributes

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

Parameters

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

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

`bp_media_prop_get()`

<bp_media.h>

Returns the media properties. The properties, including the media size as well as the read, write and erase block sizes are returned through the `p_prop` argument.

Prototype

```
int  bp_media_prop_get  ( bp_media_hndl_t    hndl,
                          bp_media_prop_t *  p_prop );
```

Attributes

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

Parameters

`hndl`	Handle of the media module instance to query.
`p_prop`	Pointer to the returned properties.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

`bp_media_read()`

<bp_media.h>

Reads `len` bytes to the destination buffer `p_src` from media address `addr`. While the API accepts any addresses and length at the byte granularity it is the caller's responsibility to observe the read block size restriction of the underlying media. `bp_media_prop_get()` can be called to query a the minimum read block size of a media instance.

The timeout value is the amount of time to wait for the media to be available. The time spent doing the read operation is not counted to consider a timeout condition. In case `RTNC_TIMEOUT` is returned it is guaranteed that no data has been read from the device.

Drivers are allowed to use an internal timeout, independent of the `timeout_ms` argument, to detect a stuck peripheral when a read operation is taking longer than expected. An `RTNC_FATAL` or `RTNC_IO_ERR` error is returned in those cases, see the driver's documentation for details.

It is unspecified how much data, if any, was actually read from a failed transfer when `RTNC_FATAL` or `RTNC_IO_ERR` is returned.

Prototype

```
int  bp_media_read  ( bp_media_hndl_t  hndl,
                      uint64_t         addr,
                      void *           p_dest,
                      uint64_t         len,
                      uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the media module instance to read from.
`addr`	Source address.
`p_dest`	Pointer to the buffer that will receive the data.
`len`	Length of data to read in bytes.
`timeout_ms`	Timeout value in millisecond

Returned Errors	RTNC_SUCCESS RTNC_TIMEOUT RTNC_IO_ERR RTNC_FATAL

Function

bp_media_reset()

<bp_media.h>

Resets a media instance. Upon a successful call to `bp_media_reset()` the media is left in the created state, equivalent to the state of a newly created instance. Before using the instance again, it must be reconfigured and enabled using the media driver specific configuration function and `bp_media_en()`.

Prototype

```
int  bp_media_reset  ( bp_media_hndl_t  hndl,
                       uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the media instance to reset.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors	RTNC_SUCCESS RTNC_TIMEOUT RTNC_FATAL

Function

bp_media_write()

<bp_media.h>

Writes `len` bytes from source buffer `p_src` to destination address `addr`. While the API accepts any addresses and length at the byte granularity it is the caller's responsibility to observe the write block size restriction of the underlying media. `bp_media_prop_get()` can be called to query the minimum write block size of a media instance.

The timeout value is the amount of time to wait for the media to be available. The time spent doing the write operation is not counted to consider a timeout condition. In case `RTNC_TIMEOUT` is returned it is guaranteed that no data has been written or modified on the device.

Drivers are allowed to use an internal timeout, independent of the `timeout_ms` argument, to detect a stuck peripheral when a write operation is taking longer than expected. An `RTNC_FATAL` or `RTNC_IO_ERR` error is returned in those cases, see the driver's documentation for details.

It is unspecified how much data, if any, was actually written from a failed transfer when `RTNC_FATAL` or `RTNC_IO_ERR` is returned.

Prototype

```
int  bp_media_write  ( bp_media_hndl_t  hndl,
                       uint64_t         addr,
                       const void *     p_src,
                       uint64_t         len,
                       uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the media module instance to write to.
`addr`	Destination address.
`p_src`	Pointer to the data to write.
`len`	Length of data to write in bytes.
`timeout_ms`	Timeout value in millisecond

Returned Errors

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

Data Type

`bp_media_hndl_t`

<bp_media.h>

Media module handle. The pointer contained in the handle is private and should not be accessed by calling code.

Members

`p_hndl`	`bp_media_inst_t *`	Pointer to the internal media module instance data.

Data Type

`bp_media_prop_t`

<bp_media.h>

Media properties structure. Returned by `bp_media_prop_get()` to inform the application or a file system of the media size and read, write and erase operations minimum size and alignment.

In the case of a device that doesn't require erasing, the `erase_block_sz_log2` member will be set to 0.

Members

`size`	`uint32_t`	Media size in bytes.
`erase_blk_sz_log2`	`uint32_t`	Base 2 logarithm of the erase block size.
`write_blk_sz_log2`	`uint32_t`	Base 2 logarithm of the write block size.

`read_blk_sz_log2`	`uint32_t`	Base 2 logarithm of the read block size.

Macro

`BP_MEDIA_HNDL_IS_NULL()`

<bp_media.h>

Evaluates if a media module handle is NULL.

Prototype `BP_MEDIA_HNDL_IS_NULL ( hndl );`

Parameters `hndl` Handle to be checked.

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

Macro

`BP_MEDIA_NULL_HNDL`

<bp_media.h>

NULL media module handle.

QSPI Memory

The QSPI memory media driver is responsible for interfacing with external memory using the QSPI interface. These types of memory include NOR, MRAM and FRAM devices. The QSPI memory driver is meant to be used by the media module to provide an abstract low-level interface to the underlying storage. In turn, a device specific driver for the connected media is used to handle the protocol variations and device-specific configuration of each QSPI memory.

The initialization of the media module and its media driver follows a different convention compared to most BASEplatform module. The media module does not have the usual create and configuration functions, instead the QSPI memory driver create function should be called which will return a media handle. Following the call to the driver specific create, the driver's configuration function should be called to setup the various configurations specific to that type of media. For example, to initialize a QSPI NOR Flash memory bp_qspi_mem_create() would be called followed by `bp_qspi_mem_cfg_set()` to configure the NOR speed, width and other parameters specific to QSPI memories. Following those steps the generic media interface can be used starting with bp_media_en() to enable the newly created media.

At the device creation, a QSPI memory board definition must be passed to `bp_qspi_mem_create()`. This board definition contains, among other things, the part definition as well as which physical QSPI interface to use. The part definition describes the part requirements, such as interface driver, operations timing and capabilities. Unless using a custom or special part or board, these definitions should be provided with the BASEplatform. Creation of a media instance using the QSPI memory driver is thus simply a matter of passing the pre-defined definition board definition of the QSPI memory to use to bp_qspi_mem_create().

The QSPI memory API is byte oriented giving maximum versatility since most QSPI memories such as NOR flash are byte addressable. However, it is important to comply with the write and erase block size restriction when relevant. When interfacing with such a media care should be taken to make sure the size and address of the erase or write operations are a multiple of the minimum operation size. bp_media_prop_get() can be called to query the minimum read, write and erase sizes of the underlying memory.

Function

bp_qspi_mem_cfg_get()

<bp_qspi_mem.h>

Retrieves the QSPI memory configuration. If successful, the QSPI memory configuration is returned through p_cfg.

Prototype

```
int  bp_qspi_mem_cfg_get  ( bp_media_hndl_t      hndl,
                            bp_qspi_mem_cfg_t *  p_cfg,
                            uint32_t             timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the QSPI memory media instance to query.
p_cfg	Pointer to the returned configuration.
timeout_ms	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Function

bp_qspi_mem_cfg_set()

<bp_qspi_mem.h>

Configures a QSPI memory media instance. The QSPI memory configuration including the bus speed, interface width, read command mode as well as the DDR mode of the interface will be set from p_cfg. Once configured, the QSPI memory should be enabled by calling bp_media_en().

The underlying QSPI peripheral and QSPI chip drivers will attempt to configure the closest clock to the specified speed that is less than the requested frequency. If the requested clock speed is higher than the maximum supported speed of the QSPI peripheral or the memory chip the lowest maximum frequency of all the components involved will be set.

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

Trying to configure an unsuported width or mode will return RTNC_NOT_SUPPORTED. Special values of BP_QSPI_MEM_MODE_AUTO and BP_QSPI_DDR_MODE_AUTO can be used to configure the best interface mode supported by both the memory chip and the QSPI interface.

Prototype

```
int  bp_qspi_mem_cfg_set  ( bp_media_hndl_t            hndl,
                            const bp_qspi_mem_cfg_t *  p_cfg,
                            uint32_t                   timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI memory media instance to configure.
`p_cfg`	Pointer to the configuration to apply.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

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

Function

bp_qspi_mem_create()

<bp_qspi_mem.h>

Creates a QSPI memory media instance. As described in the module overview `qspi_mem_create()` will return a media handle through `p_hndl`. The more generic media module interface can then be used to perform I/O operations to the QSPI memory.

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

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

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

When `bp_qspi_mem_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_qspi_mem_create  ( const bp_qspi_mem_board_def_t *  p_def,
                           bp_media_hndl_t *                p_hndl );
```

Attributes

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

Parameters

`p_def`	Definition of the QSPI memory.
`p_hndl`	Pointer to the created media module instance.

Returned Errors	RTNC_SUCCESS RTNC_ALREADY_EXIST RTNC_NO_RESOURCE RTNC_FATAL

Function

bp_qspi_mem_destroy()

<bp_qspi_mem.h>

Destroys a QSPI memory media instance. When supported, bp_qspi_mem_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 QSPI media instance is left unaffected.

It is not necessary, but strongly recommended, to disable a QSPI media instance by calling bp_qspi_mem_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 QSPI media handle after its underlying instance is destroyed is undefined.

Prototype

```
int  bp_qspi_mem_destroy  ( bp_media_hndl_t  hndl,
                            uint32_t         timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the QSPI memory media instance to destroy.
timeout_ms	Timeout value in milliseconds.

Returned Errors	RTNC_SUCCESS RTNC_TIMEOUT RTNC_NOT_SUPPORTED RTNC_FATAL

Function

bp_qspi_mem_dis()

<bp_qspi_mem.h>

Disables a QSPI memory instance. This has the same effect as bp_media_dis(), for portability it is recommended to use bp_media_dis() when disabling a QSPI memory.

Prototype

```
int  bp_qspi_mem_dis  ( bp_media_hndl_t  hndl,
                        uint32_t         timeout_ms );
```

Attributes

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

Parameters	`hndl`	Handle of the QSPI memory instance to disable.
	`timeout_ms`	Timeout value in milliseconds.

Returned Errors
`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO`
`RTNC_FATAL`

Function

`bp_qspi_mem_en()`

<bp_qspi_mem.h>

Enables a QSPI memory instance. This has the same effect as `bp_media_en()`, for portability it is recommended to use `bp_media_en()` when enabling a QSPI memory.

Prototype

```
int  bp_qspi_mem_en  ( bp_media_hndl_t  hndl,
                       uint32_t         timeout_ms );
```

Attributes

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

Parameters	`hndl`	Handle of the QSPI memory instance to enable.
	`timeout_ms`	Timeout value in milliseconds.

Returned Errors
`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO`
`RTNC_FATAL`

Function

`bp_qspi_mem_erase()`

<bp_qspi_mem.h>

Erases `len` bytes to the starting from the QSPI media address `addr`. While the API accepts any addresses and length at the byte granularity it is the caller's responsibility to observe the erase block size restriction of the underlying QSPI media. `bp_qspi_mem_prop_get()` or the generic `bp_media_prop_get()` can be called to query the media minimum erase block size.

The timeout value is the amount of time to wait for the media to be available. The time spent doing the read operation is not counted to consider a timeout condition. In case `RTNC_TIMEOUT` is returned it is guaranteed that no data has been erased from the device.

Peripheral drivers are allowed to use an internal timeout, independent of the `timeout_ms` argument, to detect a stuck peripheral when a write operation is taking longer than expected. An `RTNC_FATAL` or `RTNC_IO` error is returned in those cases, see the driver's documentation for details.

It is unspecified how much data, if any, was actually erased from a failed transfer when `RTNC_FATAL` or `RTNC_IO` is returned.

Prototype

```
int  bp_qspi_mem_erase  ( bp_media_hndl_t  hndl,
                          uint64_t         addr,
                          uint64_t         len,
                          uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI media instance to erase.
`addr`	Media start address to erase.
`len`	Length of the area to erase.
`timeout_ms`	Timeout value in millisecond

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO`
`RTNC_FATAL`

Function

`bp_qspi_mem_is_en()`

<bp_qspi_mem.h>

Returns the enabled/disabled state of a QSPI memory media instance. If the call is successful the state of the media is returned through the argument `p_is_en`. This function is equivalent to `bp_media_is_en()`, for portability it is recommended to use `bp_media_is_en()` to query the state of a QSPI memory.

Prototype

```
int  bp_qspi_mem_is_en  ( bp_media_hndl_t  hndl,
                          bool *           p_is_en );
```

Attributes

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

Parameters

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

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

`bp_qspi_mem_media_prop_get()`

<bp_qspi_mem.h>

Returns the media properties. The properties, including the media size as well as the read, write and erase block sizes are returned through the `p_prop` argument.

Prototype

```
int  bp_qspi_mem_media_prop_get  ( bp_media_hndl_t    hndl,
                                   bp_media_prop_t *  p_prop );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI media instance to query.
`p_prop`	Pointer to the returned properties.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

`bp_qspi_mem_prop_get()`

<bp_qspi_mem.h>

Returns the QSPI memory specific media properties. The properties, including the media size as well as the read, write and erase block sizes are returned through the `p_prop` argument.

Prototype

```
int  bp_qspi_mem_prop_get  ( bp_media_hndl_t       hndl,
                             bp_qspi_mem_prop_t *  p_prop );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI media instance to query.
`p_prop`	Pointer to the returned properties.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Function

`bp_qspi_mem_read()`

<bp_qspi_mem.h>

Reads `len` bytes to the destination buffer `p_src` from media address `addr`. While the API accepts any addresses and length at the byte granularity it is the caller's responsibility to observe the read block size restriction of the underlying QSPI memory. `bp_qspi_mem_prop_get()` or the generic `bp_media_prop_get()` can be called to query the media minimum read block size.

The timeout value is the amount of time to wait for the media to be available. The time spent doing the read operation is not counted to consider a timeout condition. In case `RTNC_TIMEOUT` is returned it is guaranteed that no data has been read from the device.

Peripheral drivers are allowed to use an internal timeout, independent of the `timeout_ms` argument, to detect a stuck peripheral when a read operation is taking longer than expected. An `RTNC_FATAL` or `RTNC_IO` error is returned in those cases, see the driver's documentation for details.

It is unspecified how much data, if any, was actually read from a failed transfer when `RTNC_FATAL` or `RTNC_IO` is returned.

Prototype

```
int  bp_qspi_mem_read  ( bp_media_hndl_t  hndl,
                         uint64_t         addr,
                         void *           p_dest,
                         uint64_t         len,
                         uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI media instance to read from.
`addr`	Source address.
`p_dest`	Pointer to the buffer that will receive the data.
`len`	Length of data to read in bytes.
`timeout_ms`	Timeout value in millisecond

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO`
`RTNC_FATAL`

Function

bp_qspi_mem_reset()

<bp_qspi_mem.h>

Resets a media instance. Upon a successful call to `bp_qspi_mem_reset()` the media is left in the created state, equivalent to the state a newly created instance. Before using the QSPI memory again it must be reconfigured and enabled using `bp_qspi_mem_cfg_set()` and `bp_qspi_media_en()`.

Prototype

```
int  bp_qspi_mem_reset  ( bp_media_hndl_t  hndl,
                          uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI media instance to reset.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors	RTNC_SUCCESS RTNC_TIMEOUT RTNC_FATAL

Function

bp_qspi_mem_write()

<bp_qspi_mem.h>

Writes `len` bytes from source buffer `p_src` to destination address `addr`. While the API accepts any addresses and length at the byte granularity it is the caller's responsibility to observe the write block size restriction of the underlying QSPI memory.`bp_qspi_mem_prop_get()` or the generic `bp_media_prop_get()` can be called to query the media minimum write block size.

The timeout value is the amount of time to wait for the media to be available. The time spent doing the write operation is not counted to consider a timeout condition. In case `RTNC_TIMEOUT` is returned it is guaranteed that no data has been written or modified on the device.

Peripheral drivers are allowed to use an internal timeout, independent of the `timeout_ms` argument, to detect a stuck peripheral when a write operation is taking longer than expected. An `RTNC_FATAL` or `RTNC_IO` error is returned in those cases, see the driver's documentation for details.

It is unspecified how much data, if any, was actually written from a failed transfer when `RTNC_FATAL` or `RTNC_IO` is returned.

Prototype

```
int  bp_qspi_mem_write  ( bp_media_hndl_t  hndl,
                          uint64_t         addr,
                          const void *     p_src,
                          uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI media instance to write to.
`addr`	Destination address.
`p_src`	Pointer to the data to write.
`timeout_ms`	Timeout value in millisecond

Returned Errors	RTNC_SUCCESS RTNC_TIMEOUT RTNC_IO RTNC_FATAL

Function

bp_qspi_mem_xip_dis()

<bp_qspi_mem.h>

Disables execute in place(XIP) mode. If a QSPI memory and interface was in XIP mode calling bp_qspi_mem_xip_dis() will cause an exit from XIP mode.

Note that there is no way to know if a XIP transfer is currently in progress, as such it is important that all accesses to the memory mapped region are completed before attempting to exit XIP mode.

Prototype

```
int  bp_qspi_mem_xip_dis  ( bp_media_hndl_t  hndl,
                            uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI memory media instance to configure.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

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

Function

bp_qspi_mem_xip_en()

<bp_qspi_mem.h>

Puts a QSPI memory and interface into execute in place(XIP) mode. Once in XIP mode the memory mapped QSPI device can be read from the SoC specific memory region. Once in XIP mode it can only be exited by calling bp_qspi_mem_xip_dis().

While a QSPI memory and interface are in XIP mode it is not possible to perform writes or most other operations. The result of performing any operations other than a memory mapped write, a reset or disabling XIP mode through `bp_qspi_mem_xip_dis()` is undefined.

Prototype

```
int  bp_qspi_mem_xip_en  ( bp_media_hndl_t  hndl,
                           uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI memory media instance to configure.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

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

Function

bp_qspi_mem_xip_is_en()

<bp_qspi_mem.h>

Queries the enabled/disabled state of the execute in place(XIP) mode. The state of the QSPI memory `hndl` is returned through `p_is_en`.

Prototype

```
int  bp_qspi_mem_xip_is_en  ( bp_media_hndl_t  hndl,
                              bool *           p_is_en );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI memory media instance to query.
`p_is_en`	Pointer to the returned state, `true` if XIP mode is enabled, `false` otherwise.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_SUPPORTED`
`RTNC_FATAL`

Data Type

bp_qspi_mem_ddr_mode_t

<bp_qspi_mem.h>

QSPI memory interface Dual Data Rate (DDR) modes. Used to enable/disable or report the DDR configuration of a QSPI memory. See `bp_qspi_mem_cfg_set()` and `bp_qspi_mem_cfg_get()` as well as `bp_qspi_mem_cfg_t` for usage details.

Values

`BP_QSPI_MEM_DDR_MODE_DIS`	DDR disabled.
`BP_QSPI_MEM_DDR_MODE_EN`	DDR enabled.
`BP_QSPI_MEM_DDR_MODE_AUTO`	Enable/disable DDR mode according to hardware capabilities.
`BP_QSPI_MEM_DDR_MODE_NULL`	Special invalid value.

Data Type

bp_qspi_mem_mode_t

<bp_qspi_mem.h>

QSPI memory interface modes. Used to configure or report the width and read types of a QSPI memory. See `bp_qspi_mem_cfg_set()` and `bp_qspi_mem_cfg_get()` as well as `bp_qspi_mem_cfg_t` for usage details.

Values

`BP_QSPI_MEM_MODE_SINGLE`	Single I/O read.

`BP_QSPI_MEM_MODE_SINGLE_FAST`	Single I/O fast read.
`BP_QSPI_MEM_MODE_DUAL_OUT`	Dual output read.
`BP_QSPI_MEM_MODE_DUAL_IN_OUT`	Dual I/O read.
`BP_QSPI_MEM_MODE_QUAD_OUT`	Quad output read.
`BP_QSPI_MEM_MODE_QUAD_IN_OUT`	Quad I/O read.
`BP_QSPI_MEM_MODE_AUTO`	Automatic mode selection from hardware capabilities.
`BP_QSPI_MEM_MODE_NULL`	Special invalid value.

Data Type

bp_qspi_mem_timing_tbl_t

<bp_qspi_mem.h> *Prototype* `bp_qspi_mem_timing_tbl_t ( );`

Attributes

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

Parameters

Data Type

bp_qspi_mem_board_def_t

<bp_qspi_mem.h>

QSPI memory board definition structure. This is used to describe a complete QSPI memory including the interface name, the QSPI peripheral to use as well as the memory device driver and characteristics.

This structure is passed to `bp_qspi_mem_create()` to create a QSPI memory media instance.

Members

`p_name`	`const char *`	QSPI memory media instance name.
`p_qspi_name`	`const char *`	Name of the QSPi memory interface to use.
`p_part_def`	`const bp_qspi_mem_part_def_t *`	< QSPI memory part definition.

Data Type

bp_qspi_mem_cfg_t

<bp_qspi_mem.h>

QSPI memory configuration structure. Used by `bp_qspi_mem_cfg_set()` and 'bp_qspi_mem_cfg_get() to set and report the QSPI memory configuration. The configuration includes the clock frequency, bus mode and width as well as DDR mode.

Members

`clk_freq`	`uint32_t`	Desired clock frequency.
`mode`	`bp_qspi_mem_mode_t`	QSPI memory interface mode.
`ddr_mode`	`bp_qspi_mem_ddr_mode_t`	QSPI memory interface DDR mode.

Data Type
`bp_qspi_mem_drv_hndl_t`

<bp_qspi_mem.h>

QSPI memory 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_qspi_mem_drv_create_t` and the driver documentation for details.

Members

`p_hndl`	`void *`	Pointer to the internal QSPI memory driver data.

Data Type
`bp_qspi_mem_inst_t`

<bp_qspi_mem.h>

QSPI memory instance. The fields contained within the `bp_qspi_mem_inst_t` structure are private and should not be accessed by the application.

Data Type
`bp_qspi_mem_op_timing_tbl_t`

<bp_qspi_mem.h>

QSPI memory device operation timing table. This table is used within a `bp_qspi_mem_part_def_t` structure to define the timing of various QSPI memory operations.

Members

`page_prgm_min_us`	`uint32_t`	Minimum page programing time in microseconds.
`page_prgm_poll_dly_us`	`uint32_t`	Page programming poll delay in microseconds.
`page_prgm_max_us`	`uint32_t`	Maximum page programing time in microseconds.
`chip_erase_min_ms`	`uint32_t`	Minimum whole chip erase time in milliseconds.
`chip_erase_poll_dly_ms`	`uint32_t`	Whole chip erase poll delay in milliseconds.
`chip_erase_max_ms`	`uint32_t`	Maximum whole chip erase time in milliseconds.
`subsec_erase_min_ms`	`uint32_t`	Minimum subsector erase time in milliseconds.
`subsec_erase_poll_dly_ms`	`uint32_t`	Subsector erase poll delay in milliseconds.

subsec_erase_max_ms	uint32_t	Maximum subsector erase time in milliseconds.
sec_erase_min_ms	uint32_t	Minimum sector erase time in milliseconds.
sec_erase_poll_dly_ms	uint32_t	sector erase poll delay in milliseconds.
sec_erase_max_ms	uint32_t	Maximum sector erase time in milliseconds.

Data Type

bp_qspi_mem_part_def_t

<bp_qspi_mem.h>

QSPI memory part definition structure. This part definition structure is used to specify various paramteres of a QSPI memory such as a NOR flash. This includes the device driver, operation timing as well as the ID of the memory chip.

This structure is used within a QSPI memory board definition structure to describe a complete QSPI memory which can then be used to create a QSPI memory instance using bp_qspi_mem_create(). See bp_qspi_mem_board_def_t for additional details on the usage of this structure.

Members

p_part_name	const char *	Part name.
p_qspi_mem_drv	const bp_qspi_mem_drv_t *	Driver associated with this part.
sdr_timing_tbl	const bp_qspi_mem_timing_tbl_t *	Single data rate interface timing table.
ddr_timing_tbl	const bp_qspi_mem_timing_tbl_t *	Dual data rate interface timing table.
p_op_timing_tbl	const bp_qspi_mem_op_timing_tbl_t *	Operations timing table.
manuf_id	uint8_t	JEDEC manufacturer ID.
device_id	uint8_t	JEDEV device ID.
sdr_timing_tbl_depth	uint8_t	Number of rows of the single data rate interface timing table.
ddr_timing_tbl_depth	uint8_t	Number of rows of the dual data rate interface timing table.
p_drv_part_def	const void *	Pointer to additional driver specific data.

Data Type

`bp_qspi_mem_prop_t`

<bp_qspi_mem.h>

QSPI memory properties. Returned by `bp_qspi_mem_prop_get()` to inform the application or a file system of the media size and read, write and erase operation minimum sizes and alignment.

In the case of a device that doesn't require erasing, the `erase_block_sz_log2` member will be set to 0.

Macro

`BP_QSPI_MEM_DDR_MODE_IS_VALID()`

<bp_qspi_mem.h>

Checks if the QSPI memory DDR mode value is valid.

Expansion	`true` if the mode value is valid. `false` otherwise.

Macro

`BP_QSPI_MEM_MODE_IS_VALID()`

<bp_qspi_mem.h>

Checks if the QSPI memory mode value is valid.

Expansion	`true` if the mode value is valid. `false` otherwise.

Macro

`BP_QSPI_MEM_TIMING_TBL_COL_CNT`

<bp_qspi_mem.h>

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_*`

<rtnc.h>

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.
`RTNC_INVALID_FMT`	Invalid format.
`RTNC_INVALID_PATH`	Invalid path.
`RTNC_CORRUPT`	Data corrupted.
`RTNC_FULL`	Container full.
`RTNC_OVERFLOW`	Overflow
`RTNC_FAIL`	Operation failed.

Architecture Definitions

Definitions used by the architecture module to set the CPU architecture, compiler and endianness.

Macro

BP_ARCH_CPU_ARM_V5

<bp_arch_def.h>

ARM v5, for example the ARM9.

Macro

BP_ARCH_CPU_ARM_V6

<bp_arch_def.h>

ARM v6, for example the ARM11.

Macro

BP_ARCH_CPU_ARM_V6M

<bp_arch_def.h>

ARM v6m, for example the Cortex-M0.

Macro

BP_ARCH_CPU_ARM_V7AR

<bp_arch_def.h>

ARM v7ar, for example the Cortex-A9 or Cortex-R5.

Macro

BP_ARCH_CPU_ARM_V7M

<bp_arch_def.h>

ARM v7m, for example the Cortex-M4.

Macro

BP_ARCH_CPU_ARM_V8A

<bp_arch_def.h>

ARM v8a, for example the Cortex-A53.

Macro

BP_ARCH_CPU_ARM_V8M

<bp_arch_def.h>

ARM v8a, for example the Cortex-M23.

Macro

BP_ARCH_CPU_ARM_V8R

<bp_arch_def.h>

ARM v8r, for example the Cortex-R52.

Macro

BP_ARCH_CPU_LINUX

<bp_arch_def.h>

Linux, any architecture.

Macro

BP_ARCH_CPU_MICROBLAZE

<bp_arch_def.h>

Xilinx Microblaze soft processor.

Macro

BP_ARCH_CPU_NONE

<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_POWERISA2

<bp_arch_def.h>

PowerISA 2.xx.

Macro

BP_ARCH_CPU_SPARCV8

<bp_arch_def.h>

SPARC v8.

Macro

BP_ARCH_CPU_SPARCV9

<bp_arch_def.h>

SPARC v9.

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

<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

<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

<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

<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

<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

<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

<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

<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

<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

<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`

<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()`

<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`

<bp_gpio_drv.h>

NULL GPIO driver handle.

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

<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

<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

<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

<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`

<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`

<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

<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

<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

<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`

<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`

<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

<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

<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()`

<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`

<bp_i2c_drv.h>

NULL I2C driver handle.

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

<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

<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

<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

<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

<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

<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

<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`

<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`

<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

<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

<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

<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`

<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`

<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

<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()

<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

<bp_spi_drv.h>

NULL SPI driver handle.

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's 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

<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`

<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 driver 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`

<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

<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

<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

<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

<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

<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

<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

<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

<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

<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

<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

<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`

<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`

<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

<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

<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()

<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

<bp_uart_drv.h>

NULL UART driver handle.

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()

<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()

<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()

<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()

<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

NOR Driver

The QSPI memory driver, not to be confused with the QSPI memory media driver, provides the interface with a specific QSPI memory chip. The QSPI memory driver abstracts various aspects of interacting a QSPI memory such as the command set, initialization sequence and error handling.

Data Type

bp_qspi_mem_drv_cfg_get_t

<bp_qspi_mem_drv.h>

QSPI memory driver's configuration get function.

Prototype

```
int  bp_qspi_mem_drv_cfg_get_t  ( bp_qspi_mem_drv_hndl_t  hndl,
                                  bp_qspi_mem_cfg_t *      p_cfg );
```

Attributes

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

Parameters

hndl	Handle of the QSPI memory driver instance to query.
p_cfg	Pointer to the returned configuration.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Data Type

bp_qspi_mem_drv_cfg_set_t

<bp_qspi_mem_drv.h>

QSPI memory driver's configuration set function.

Prototype

```
int  bp_qspi_mem_drv_cfg_set_t  ( bp_qspi_mem_drv_hndl_t    hndl,
                                  const bp_qspi_mem_cfg_t *  p_cfg,
                                  uint32_t                   timeout_ms );
```

Attributes

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

Parameters		
	`hndl`	Handle of the QSPI memory driver instance to configure.
	`p_cfg`	Pointer to the configuration to apply.
	`timeout_ms`	Timeout value in milliseconds.

Returned Errors

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

Data Type

`bp_qspi_mem_drv_create_t`

<bp_qspi_mem_drv.h>

QSPI memory driver's create function.

Prototype

```
int  bp_qspi_mem_drv_create_t  ( const bp_qspi_mem_board_def_t *  p_def,
                                  bp_qspi_mem_drv_hndl_t *         p_hndl );
```

Attributes

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

Parameters		
	`p_def`	Board definition of the QSPI memory.
	`p_hndl`	Pointer to the returned handle.

Returned Errors

`RTNC_SUCCESS`
`RTNC_RESOURCE`
`RTNC_FATAL`

Data Type

`bp_qspi_mem_drv_destroy_t`

<bp_qspi_mem_drv.h>

QSPI memory driver's destroy function.

Prototype

```
int  bp_qspi_mem_drv_destroy_t  ( bp_qspi_mem_drv_hndl_t  hndl,
                                   uint32_t                timeout_ms );
```

Attributes

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

Parameters		
	`hndl`	Handle of the QSPI memory driver instance to destroy.
	`timeout_ms`	Timeout value in milliseconds.

Returned Errors
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_NOT_SUPPORTED
RTNC_FATAL

Data Type

bp_qspi_mem_drv_dis_t

<bp_qspi_mem_drv.h>

QSPI memory driver's disable function.

Prototype

```
int bp_qspi_mem_drv_dis_t ( bp_qspi_mem_drv_hndl_t  hndl,
                            uint32_t                timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the QSPI memory driver instance to disable.
timeout_ms	Timeout value in milliseconds.

Returned Errors
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_IO
RTNC_FATAL

Data Type

bp_qspi_mem_drv_en_t

<bp_qspi_mem_drv.h>

QSPI memory driver's enable function.

Prototype

```
int bp_qspi_mem_drv_en_t ( bp_qspi_mem_drv_hndl_t  hndl,
                           uint32_t                timeout_ms );
```

Attributes

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

Parameters

hndl	Handle of the QSPI memory driver instance to enable.
timeout_ms	Timeout value in milliseconds.

Returned Errors
RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_IO
RTNC_FATAL

Data Type

bp_qspi_mem_drv_erase_t

<bp_qspi_mem_drv.h>

QSPI memory driver's erase function.

Prototype

```
int  bp_qspi_mem_drv_erase_t  ( bp_qspi_mem_drv_hndl_t  hndl,
                                uint64_t                addr,
                                uint64_t                len,
                                uint32_t                timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI memory driver instance to erase.
`addr`	Media start address to erase.
`len`	Length of the area to erase.
`timeout_ms`	Timeout value in millisecond

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO`
`RTNC_FATAL`

Data Type

bp_qspi_mem_drv_is_en_t

<bp_qspi_mem_drv.h>

QSPI memory driver's is enabled function.

Prototype

```
int  bp_qspi_mem_drv_is_en_t  ( bp_qspi_mem_drv_hndl_t  hndl,
                                bool *                  p_is_en );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI memory driver instance to query.
`p_is_en`	Interface state, true if enabled false otherwise.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

bp_qspi_mem_drv_read_t

<bp_qspi_mem_drv.h>

QSPI memory driver's read function.

Prototype

```
int  bp_qspi_mem_drv_read_t  ( bp_qspi_mem_drv_hndl_t  hndl,
                               uint64_t                addr,
                               void *                  p_dest,
                               uint64_t                len,
                               uint32_t                timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI driver instance to read from.
`addr`	Source address.
`p_dest`	Pointer to the buffer that will receive the data.
`len`	Length of data to read in bytes.
`timeout_ms`	Timeout value in millisecond

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO`
`RTNC_FATAL`

Data Type
bp_qspi_mem_drv_reset_t

<bp_qspi_mem_drv.h>

QSPI memory driver's reset function.

Prototype

```
int  bp_qspi_mem_drv_reset_t  ( bp_qspi_mem_drv_hndl_t  hndl,
                                uint32_t                timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI memory driver instance to reset.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_FATAL`

Data Type
bp_qspi_mem_drv_write_t

<bp_qspi_mem_drv.h>

QSPI memory driver's write function.

Prototype

```
int  bp_qspi_mem_drv_write_t  ( bp_qspi_mem_drv_hndl_t  hndl,
                                uint64_t                addr,
                                const void *            p_src,
                                uint64_t                len,
                                uint32_t                timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI driver instance to write to.
`addr`	Destination address.
`p_src`	Pointer to the data to write.
`len`	
`timeout_ms`	

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO`
`RTNC_FATAL`

Data Type

`bp_qspi_mem_drv_xip_dis_t`

<bp_qspi_mem_drv.h>

QSPI memory driver's execute in place disable function.

Prototype

```
int  bp_qspi_mem_drv_xip_dis_t  ( bp_qspi_mem_drv_hndl_t  hndl,
                                  uint32_t                timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI memory driver instance to configure.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

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

Data Type

bp_qspi_mem_drv_xip_en_t

<bp_qspi_mem_drv.h>

QSPI memory driver's execute in place enable function.

Prototype

```
int bp_qspi_mem_drv_xip_en_t ( bp_qspi_mem_drv_hndl_t hndl,
                               uint32_t               timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI memory driver instance to configure.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

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

Data Type

bp_qspi_mem_drv_xip_is_en_t

<bp_qspi_mem_drv.h>

QSPI memory driver's execute in place query function.

Prototype

```
int bp_qspi_mem_drv_xip_is_en_t ( bp_qspi_mem_drv_hndl_t hndl,
                                  bool *                 p_is_en );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI memory driver instance to query.
`p_is_en`	Pointer to the returned state, true if XIP mode is enabled, false otherwise.

Returned Errors

`RTNC_SUCCESS`
`RTNC_NOT_SUPPORTED`
`RTNC_FATAL`

Data Type

bp_qspi_mem_prop_get_t

<bp_qspi_mem_drv.h>

QSPI memory driver's properties get function.

Prototype

```
int bp_qspi_mem_prop_get_t ( bp_qspi_mem_drv_hndl_t  hndl,
                             bp_qspi_mem_prop_t *    p_prop );
```

Attributes

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

Parameters

`hndl`	Handle of the QSPI memory driver instance to query.
`p_prop`	Pointer to the returned properties.

Returned Errors `RTNC_SUCCESS`
`RTNC_FATAL`

Macro

BP_QSPI_MEM_DRV_HNDL_IS_NULL()

<bp_qspi_mem_drv.h>

Evaluates if a QSPI memory driver handle is NULL.

Prototype

```
BP_QSPI_MEM_DRV_HNDL_IS_NULL ( hndl );
```

Parameters `hndl` Handle to be checked.

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

Macro

BP_QSPI_MEM_DRV_NULL_HNDL

<bp_qspi_mem_drv.h>

NULL QSPI memory driver handle.

Media Driver

The media driver declarations found in this module serve as the basis of every media driver implementation. The media drivers are usually used in combination with the media module to provide a unified interface toward various non-volatile storage media.

Note that many media driver implementations layer a generic media driver on top of a storage technology specific driver. For example accessing a QSPI NOR media is done through a QSPI memory driver which in turn provides a generic media driver interface for the media module.

The media module API function `bp_media_drv_hndl_get()` function can be used to retrieve the driver handle associated with a media module instance, and can subsequently be used to call the driver directly. See the individual media driver's documentation for details of the extended functions.

Data Type

bp_media_drv_dis_t

<bp_media_drv.h>

Media driver's disable function.

Prototype

```
int  bp_media_drv_dis_t  ( bp_media_hndl_t  hndl,
                           uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the media driver instance to disable.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO`
`RTNC_FATAL`

Data Type

bp_media_drv_en_t

<bp_media_drv.h>

Media driver's enable function.

Prototype

```
int  bp_media_drv_en_t  ( bp_media_hndl_t  hndl,
                          uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the media driver instance to enable.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO`
`RTNC_FATAL`

Data Type

bp_media_drv_erase_t

<bp_media_drv.h>

Media driver's erase function.

Prototype

```
int  bp_media_drv_erase_t  ( bp_media_hndl_t  hndl,
                             uint64_t         addr,
                             uint64_t         len,
                             uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the media module driver instance to erase.
`addr`	Media start address to erase.
`len`	Length of the area to erase.
`timeout_ms`	Timeout value in millisecond

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO`
`RTNC_FATAL`

Data Type

bp_media_drv_is_en_t

<bp_media_drv.h>

Media driver's is_enabled function.

Prototype

```
int  bp_media_drv_is_en_t  ( bp_media_hndl_t  hndl,
                             bool *           p_is_en );
```

Attributes

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

Parameters

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

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

bp_media_drv_prop_get_t

<bp_media_drv.h>

Media driver's property get function.

Prototype

```
int  bp_media_drv_prop_get_t  ( bp_media_hndl_t    hndl,
                                bp_media_prop_t *  p_prop );
```

Attributes

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

Parameters

`hndl`	Handle of the media driver instance to query.
`p_prop`	Pointer to the returned properties.

Returned Errors

`RTNC_SUCCESS`
`RTNC_FATAL`

Data Type

`bp_media_drv_read_t`

<bp_media_drv.h>

Media driver's read function.

Prototype

```
int  bp_media_drv_read_t  ( bp_media_hndl_t  hndl,
                            uint64_t         addr,
                            void *           p_dest,
                            uint64_t         len,
                            uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the media driver instance to read from.
`addr`	Source address.
`p_dest`	Pointer to the buffer that will receive the data.
`len`	Length of data to read in bytes.
`timeout_ms`	Timeout value in millisecond

Returned Errors

`RTNC_SUCCESS`
`RTNC_TIMEOUT`
`RTNC_IO`
`RTNC_FATAL`

Data Type

`bp_media_drv_reset_t`

<bp_media_drv.h>

Media driver's reset function.

Prototype

```
int  bp_media_drv_reset_t  ( bp_media_hndl_t  hndl,
                             uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the media driver instance to reset.
`timeout_ms`	Timeout value in milliseconds.

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_FATAL

Data Type

bp_media_drv_write_t

<bp_media_drv.h>

Media driver's write function.

Prototype

```
int  bp_media_drv_write_t  ( bp_media_hndl_t  hndl,
                             uint64_t         addr,
                             const void *     p_src,
                             uint64_t         len,
                             uint32_t         timeout_ms );
```

Attributes

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

Parameters

`hndl`	Handle of the media driver instance to write to.
`addr`	Destination address.
`p_src`	Pointer to the data to write.
`len`	Length of data to write in bytes.
`timeout_ms`	Timeout value in millisecond

Returned Errors

RTNC_SUCCESS
RTNC_TIMEOUT
RTNC_IO
RTNC_FATAL

Chapter 18

Document Revision History

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