XCORE RTOS Framework - Programming Guide

Release: 3.0.1

Publication Date: 2023/05/02

Table of Contents

1 XCORE Platform

The xcore platform provides a range of powerful, flexible and economic crossover processors for the use in wide-ranging applications. The XCore platform provides:

- Fast compute
- Flexibility
- Economy
- Scalablity
- Security
- Fast time to market

1.1 Architecture & Hardware Guide

At the heart of the platform, the Architecture & Hardware Guide describes the multicore processors. Multiple xcore processors can themselves be "networked" together with seamless communications.

1.2 Programming Guide

The Programming Guide describes how logical cores of an xcore processor can act independently to behave like highly responsive hardware peripherals, or can work as a team to apply all available CPU cycles onto a single compute task.

1.3 XTC Tools

The xcore processors are accompanied by the XTC Tools. As well as providing a powerful toolchain for application development, the toolkit assists with application deployment and upgrade.

2 Tutorials

2.1 FreeRTOS Application Programming

This document is intended to help you become familiar with FreeRTOS application programming on xcore.

2.1.1 Rationale

Traditionally, xcore multi-core processors have been programmed using the XC language. The XC language allows the programmer to statically place tasks on the available hardware cores and wire them together with channels to provide inter-process communication. The XC language also exposes "events," which are unique to the xcore architecture and are a useful alternative to interrupts.

Using the XC language, it is possible to write dedicated application software with deterministic timing and very low latency between I/O and tasks.

While XC elegantly enables the intrinsic, unique capabilities of the xcore architecture, there often needs to be higher level application type software running alongside it. The programming model that makes the lower level deterministic software possible may not be best suited for many higher level parts of an application that do not require deterministic timing. Where strict real-time execution is not required, higher level abstractions can be used to manage finite hardware resources, and provide a more familiar programming environment.

A symmetric multiprocessing (SMP) real time operating system (RTOS) can be used to simplify xcore application designs, as well as to preserve the hard real-time benefits provided by the xcore architecture for the lower level software functions that require it.

This document assumes familiarity with real time operating systems in general. Familiarity with FreeRTOS specifically should not be required, but will be helpful. For current up to date documentation on FreeRTOS see the following links on the FreeRTOS website.

- Overview
- Developer Documentation
- API

2.1.2 SMP FreeRTOS

To support this new programming model for xcore, XMOS has extended the popular and free FreeRTOS kernel to support SMP. This allows for the kernel's scheduler to be started on any number of available xcore logical cores per tile, leaving the remaining free to support other program elements that combine to create complete systems. Once the scheduler is started, FreeRTOS threads are placed on cores dynamically at runtime, rather than statically at compile time. All the usual FreeRTOS rules for thread scheduling are followed, except that rather than only running the single highest priority thread that is ready at any given time, multiple threads may run simultaneously. The threads chosen to run are always the highest priority threads that are ready. When there are more threads of a single priority that are ready to run than the number of cores available, they are scheduled in a round robin fashion. Dynamic scheduling allows FreeRTOS to optimize physical core usage based on priority and availability at runtime, opening up the potential for using tile wide MIPs more efficiently than what could be manually specified in a static compile time setting.

One of xcore's primary strengths is its guarantee of deterministic behavior and timing. RTOS threads can also benefit from this determinism provided by the xcore architecture. An RTOS thread with interrupts disabled and a high enough priority behaves just as a bare-metal thread. An SMP RTOS kernel does not need to preempt a high priority thread because it has many other cores to utilize to schedule lower priority threads. Using an SMP RTOS allows developers to concentrate on specific requirements of their application without worrying about what affect they might have on non-preemptable thread response times. Furthermore, modification of the program in the future is much easier because the developer does not have to worry about affecting existing responsiveness with changes in unrelated areas. The non-preemptable threads will not be effected by adding lower-priority functionality.

Another xcore strength is it's performance. xcore.ai provides lightning fast general purpose compute, AI acceleration, powerful DSP and instantaneous I/O control. RTOS threads can also benefit from the performance provided by the xcore architecture, allowing an application developer to dynamically shift performance usage from one application feature to another.

The standard FreeRTOS kernel supports dynamic task priorities, while the FreeRTOS-SMP kernel adds the following additional APIs:

- vTaskCoreAffinitySet
- vTaskCoreAffinityGet
- vTaskPreemptionDisable
- vTaskPreemptionEnable

Together, these API enable a developer to take full advantage of xcore's performance.

Some additional configuration options are also available to the FreeRTOS-SMP Kernel:

- configNUM_CORES
- configRUN_MULTIPLE_PRIORITIES
- configUSE_CORE_AFFINITY
- configUSE_TASK_PREEMPTION_DISABLE

See Symmetric Multiprocessing (SMP) with FreeRTOS for additional information on SMP support in the FreeRTOS kernel and SMP specific considerations.

2.1.3 AMP SMP FreeRTOS

To further leverage the xcore hardware and the FreeRTOS programming model, XMOS provides support for asymmetric multiprocessing (AMP) per tile. Each XMOS chip contains at least two tiles, which consist of their own set of logical xcore cores, IO, memory space, and more. XMOS provides a build method and variety of software drivers to allow an application to be created that is an AMP system containing, multiple SMP FreeRTOS kernels.

2.1.4 RTOS Drivers

To help ease development of xcore applications using an SMP RTOS, XMOS provides several SMP RTOS compatible drivers. These include, but are not necessarily limited to:

- Common I/O interfaces
  - GPIO
  - UART
  - I^2C

  - I^2S
  - PDM microphones
  - QSPI flash
  - SPI
  - USB
  - Clock control
- xcore features
  - Intertile channel communication
  - Software defined memory
  - Software defined L2 Cache
- External parts
  - Silicon Labs WF200 series WiFi transceiver

These drivers are all found in the RTOS framework under the path modules/rtos/modules/drivers.

Documentation on each of these drivers can be found under the *RTOS Drivers* section in the RTOS framework documentation pages.

It is worth noting that most of these drivers utilize a lightweight RTOS abstraction layer, meaning that they are not dependent on FreeRTOS. Conceivably they should work on any SMP RTOS, provided an abstraction layer for it is provided. This abstraction layer is found under the path modules/rtos/modules/osal. At the moment the only available SMP RTOS for xcore is the XMOS SMP FreeRTOS, but more may become available in the future.

2.1.5 Software Services

The RTOS framework also includes some higher level RTOS compatible software services, some of which call the aforementioned drivers. These include, but are not necessarily limited to:

- DHCP server
- FAT filesystem
- HTTP parser
- JSON parser
- MQTT client
- SNTP client
- TLS
- USB stack
- WiFi connection manager

Documentation on several software services can be found under the *RTOS Services* section in the RTOS framework documentation pages.

These services are all found in the RTOS framework under the path modules/rtos/modules/sw_services.

2.2 Board Support Configurations

xcore leverages its architecture to provide a flexible chip where many typically silicon based peripherals are found in software. This allows a chip to be reconfigured in a way that provides the specific IO required for a given application, thus resulting in a low cost yet incredibly silicon efficient solution. Board support configurations (bsp_configs) are the description for the hardware IO that exists in a given board. The bsp_configs provide the application programmer with an API to initialize and start the hardware configuration, as well as the supported RTOS driver contexts. The programming model in this FreeRTOS architecture is:

- .xn files provide the mapping of ports, pins, and links
- bsp_configs specify, setup, and start hardware IO and provide the application with RTOS driver contexts
- applications use the bsp_config init/start code as well as RTOS driver contexts, similar to conventional microcontroller programming models.

To support any generic bsp_config, applications should call `platform_init()` before starting the scheduler, and then *platform_start()* after the scheduler is running and before any RTOS drivers are used.

The bsp_configs provided with the RTOS framework in modules/rtos/modules/bsp_config are an excellent starting point. They provide the most common peripheral drivers that are supported by the boards that support RTOS framework based applications. For advanced users, it is recommended that you copy one of these bsp_config into your application project and customize as needed.

2.2.1 Creating Custom bsp_configs

To enable hardware portability, a minimal `bsp_config` should contain the following:

```
custom_config/
  platform/
    driver_instances.c
    driver_instances.h
    platform_conf.h
    platform_init.c
    platform_init.h
    platform_start.c
  custom_config.cmake
  custom_config_xn_file.xn
```

`custom_config.cmake` provides the `CMake` target of the configuration. This target should link the required RTOS framework libraries to support the configuration it defines.

`custom_config_xn_file.xn` provides various hardware parameters including but not limited to the chip package, IO mapping, and network information.

`platform_conf.h` provides default configuration of all header defined configuration macros. These may be overridden by compile definitions or application headers.

`driver_instances.h` provides the declaration of all RTOS drivers in the configuration. It may define XCORE hardware resources, such as ports and clockblocks. It may also define tile placements.

`driver_instances.c` provides the definition of all RTOS drivers in the configuration.

`platform_init.h` provides the declaration of `platform_init(chanend_t other_tile_c)` and `platform_start(void)`

`platform_init.c` provides the initialization of all drivers defined in the configuration through the definition of `platform_init(chanend_t other_tile_c)`. This code is run before the scheduler is started and therefore will not be able to access all RTOS driver functionalities nor kernel objects.

`platform_start.c` provides the starting of all drivers defined in the configuration through the definition of `platform_start(void)`. It may also perform any initialization setup, such as configuring the app_pll or setting up an on board DAC. This code is run once the kernel is running and is therefore subject to preemption and other dynamic scheduling SMP programming considerations.

2.3 RTOS Application DFU

This document is intended to help you use the RTOS DFU driver and RTOS QSPI flash driver in an application.

2.3.1 DFU Driver Overview

This driver provides the application with the boot partition and data partition layout of the flash used by the second stage bootloader. The driver provides a subset of the functionality of libquadflash enabling the application to use any transport method and the RTOS qspi flash driver to read the factory image, read/write a single upgrade image, and read/write the data partition.

2.3.2 Reading the Factory Image

To read back the factory image:

```
unsigned addr = rtos_dfu_image_get_factory_addr(dfu_image_ctx);
unsigned size = rtos_dfu_image_get_factory_size(dfu_image_ctx);

unsigned char *buf = pvPortMalloc(sizeof(unsigned char) * size);

rtos_qspi_flash_read(
      qspi_flash_ctx,
      (uint8_t *)buf,
      addr,
      size);

// buf now contains the factory image contents
```

It is advised to perform this operation in blocks rather than full image size to reduce memory usage. Once the buffer is populated from flash, it can be sent over the desired transport method, such as USB, I^2C, etc.

2.3.3 Reading the Upgrade Image

To read back the upgrade image:

```
unsigned addr = rtos_dfu_image_get_upgrade_addr(dfu_image_ctx);
unsigned size = rtos_dfu_image_get_upgrade_size(dfu_image_ctx);

unsigned char *buf = pvPortMalloc(sizeof(unsigned char) * size);
```

(continues on next page)

(continued from previous page)

```
rtos_qspi_flash_read(
      qspi_flash_ctx,
      (uint8_t *)buf,
      addr,
      size);

// buf now contains the upgrade image contents
```

It is advised to perform this operation in blocks rather than full image size to reduce memory usage. Once the buffer is populated from flash, it can be sent over the desired transport method, such as USB, I^2C, etc.

2.3.4 Writing the Upgrade Image

To overwrite the current upgrade image:

```
// Assuming buf contains the image data
// and size contains the size in bytes

unsigned addr = rtos_dfu_image_get_upgrade_addr(dfu_image_ctx);
unsigned data_partition_base_addr = rtos_dfu_image_get_data_partition_addr(dfu_image_ctx);
unsigned bytes_avail = data_partition_base_addr - addr;

size_t sector_size = rtos_qspi_flash_sector_size_get(qspi_flash_ctx);

if(size < bytes_avail) {
   unsigned char *tmp_buf = pvPortMalloc(sizeof(unsigned char) * sector_size);
   unsigned cur_offset = 0;
   do {
      unsigned length = (size - (cur_offset - addr)) >= sector_size ? sector_size : (size -␣
↪(cur_offset - addr));
      rtos_qspi_flash_lock(qspi_flash_ctx);
      {
         rtos_qspi_flash_read(
                  qspi_flash_ctx,
                  tmp_buf,
                  addr + cur_offset,
                  sector_size);
         memcpy(tmp_buf, data + cur_offset, length);
         rtos_qspi_flash_erase(
                  qspi_flash_ctx,
                  addr + cur_offset,
                  sector_size);
         rtos_qspi_flash_write(
                  qspi_flash_ctx,
                  (uint8_t *) tmp_buf,
                  addr + cur_offset,
                  sector_size);
      }
      rtos_qspi_flash_unlock(qspi_flash_ctx);
      cur_offset += length;
```

(continues on next page)

(continued from previous page)

```
    } while(cur_offset < (size - 1));

    vPortFree(tmp_buf);
} else {
    rtos_printf("Insufficient space for upgrade image\n");
}
```

It is advised to perform this operation in blocks rather than full image size to reduce memory usage. The buffer can be populated over the desired transport method, such as USB, I^2C, etc.

2.3.5 Reading the Data Partition Image

To read back the data partition image:

```
unsigned addr = rtos_dfu_image_get_data_partition_addr(dfu_image_ctx);
unsigned size = rtos_qspi_flash_size_get(qspi_flash_ctx);

unsigned char *buf = pvPortMalloc(sizeof(unsigned char) * size);

rtos_qspi_flash_read(
      qspi_flash_ctx,
      (uint8_t *)buf,
      addr,
      size);

// buf now contains the data partition image contents
```

It is advised to perform this operation in blocks rather than full image size to reduce memory usage. The data partition will likely be too large to read into SRAM in a read single operation. Once the buffer is populated from flash, it can be sent over the desired transport method, such as USB, I^2C, etc.

2.3.6 Writing the Data Partition Image

To overwrite the current data partition image:

```
// Assuming buf contains the image data
// and size contains the size in bytes

unsigned addr = rtos_dfu_image_get_data_partition_addr(dfu_image_ctx);
unsigned end_addr = rtos_qspi_flash_size_get(qspi_flash_ctx);
unsigned bytes_avail = end_addr - addr;

size_t sector_size = rtos_qspi_flash_sector_size_get(qspi_flash_ctx);

if(size < bytes_avail) {
    unsigned char *tmp_buf = pvPortMalloc(sizeof(unsigned char) * sector_size);
    unsigned cur_offset = 0;
    do {
        unsigned length = (size - (cur_offset - addr)) >= sector_size ? sector_size : (size -␣
↪(cur_offset - addr));
```

(continues on next page)

(continued from previous page)

```
      rtos_qspi_flash_lock(qspi_flash_ctx);
      {
         rtos_qspi_flash_read(
                  qspi_flash_ctx,
                  tmp_buf,
                  addr + cur_offset,
                  sector_size);
         memcpy(tmp_buf, data + cur_offset, length);
         rtos_qspi_flash_erase(
                  qspi_flash_ctx,
                  addr + cur_offset,
                  sector_size);
         rtos_qspi_flash_write(
                  qspi_flash_ctx,
                  (uint8_t *) tmp_buf,
                  addr + cur_offset,
                  sector_size);
      }
      rtos_qspi_flash_unlock(qspi_flash_ctx);
      cur_offset += length;
   } while(cur_offset < (size - 1));

   vPortFree(tmp_buf);
} else {
   rtos_printf("Insufficient space for data partition image\n");
}
```

It is advised to perform this operation in blocks rather than full image size to reduce memory usage. The buffer can be populated over the desired transport method, such as USB, I^2C, etc.

3 API Reference

3.1 RTOS Drivers

3.1.1 I/O

GPIO RTOS Driver

This driver can be used to operate GPIO ports on xcore in an RTOS application.

Initialization API The following structures and functions are used to initialize and start a GPIO driver instance.

enum `rtos_gpio_port_id_t`

Enumerator type representing each available GPIO port.

To be used with the RTOS GPIO driver functions.

Values:

enumerator `rtos_gpio_port_none`

enumerator `rtos_gpio_port_1A`

enumerator `rtos_gpio_port_1B`

enumerator `rtos_gpio_port_1C`

enumerator `rtos_gpio_port_1D`

enumerator `rtos_gpio_port_1E`

enumerator `rtos_gpio_port_1F`

enumerator `rtos_gpio_port_1G`

enumerator `rtos_gpio_port_1H`

enumerator `rtos_gpio_port_1I`

enumerator `rtos_gpio_port_1J`

enumerator `rtos_gpio_port_1K`

enumerator `rtos_gpio_port_1L`

enumerator `rtos_gpio_port_1M`

enumerator `rtos_gpio_port_1N`

enumerator `rtos_gpio_port_1O`

enumerator `rtos_gpio_port_1P`

enumerator `rtos_gpio_port_4A`

enumerator `rtos_gpio_port_4B`

enumerator `rtos_gpio_port_4C`

enumerator `rtos_gpio_port_4D`

enumerator `rtos_gpio_port_4E`

enumerator `rtos_gpio_port_4F`

enumerator `rtos_gpio_port_8A`

enumerator `rtos_gpio_port_8B`

enumerator `rtos_gpio_port_8C`

enumerator `rtos_gpio_port_8D`

enumerator `rtos_gpio_port_16A`

enumerator `rtos_gpio_port_16B`

enumerator `rtos_gpio_port_16C`

enumerator `rtos_gpio_port_16D`

enumerator `rtos_gpio_port_32A`

enumerator `rtos_gpio_port_32B`

enumerator `RTOS_GPIO_TOTAL_PORT_CNT`

Total number of I/O ports

typedef struct *rtos_gpio_struct* `rtos_gpio_t`

Typedef to the RTOS GPIO driver instance struct.

typedef void (*`rtos_gpio_isr_cb_t`)(*rtos_gpio_t* *ctx, void *app_data, *rtos_gpio_port_id_t* port_id, uint32_t value)

Function pointer type for application provided RTOS GPIO interrupt callback functions.

These callback functions are called when there is a GPIO port interrupt.

Note: this is the latched value that triggered the interrupt, not the current value.

Param ctx
A pointer to the associated GPIO driver instance.

Param app_data
A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

Param port_id
The GPIO port that triggered the interrupt.

Param value
The value on the GPIO port that caused the interrupt.

inline *rtos_gpio_port_id_t* `rtos_gpio_port`(port_t p)

Helper function to convert an xcore I/O port resource ID to an RTOS GPIO driver port ID.

Parameters

- `p` – An xcore I/O port resource ID.

Returns
the equivalent RTOS GPIO driver port ID.

void `rtos_gpio_start`(*rtos_gpio_t* *ctx)

Starts an RTOS GPIO driver instance. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before any of the core GPIO driver functions are called with this instance.

rtos_gpio_init() must be called on this GPIO driver instance prior to calling this.

Parameters

- `ctx` – A pointer to the GPIO driver instance to start.

void `rtos_gpio_init`(*rtos_gpio_t* *ctx)

Initializes an RTOS GPIO driver instance. There should only be one per tile. This instance represents all the GPIO ports owned by the calling tile. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before calling *rtos_gpio_start()* or any of the core GPIO driver functions with this instance.

Parameters

- `ctx` – A pointer to the GPIO driver instance to initialize.

`RTOS_GPIO_ISR_CALLBACK_ATTR`

This attribute must be specified on all RTOS GPIO interrupt callback functions provided by the application.

struct `rtos_gpio_isr_info_t`

#include <rtos_gpio.h> Struct to hold interrupt state data for GPIO ports.

The members in this struct should not be accessed directly.

struct `rtos_gpio_struct`

#include <rtos_gpio.h> Struct representing an RTOS GPIO driver instance.

The members in this struct should not be accessed directly.

Core API The following functions are the core GPIO driver functions that are used after it has been initialized and started.

inline void `rtos_gpio_port_enable`(*rtos_gpio_t* *ctx, *rtos_gpio_port_id_t* port_id)

Enables a GPIO port. This must be called on a port before using it with any other GPIO driver function.

Parameters

- `ctx` – A pointer to the GPIO driver instance to use.
- `port_id` – The GPIO port to enable.

inline uint32_t `rtos_gpio_port_in`(*rtos_gpio_t* *ctx, *rtos_gpio_port_id_t* port_id)

Inputs the value present on a GPIO port's pins.

Parameters

- `ctx` – A pointer to the GPIO driver instance to use.
- `port_id` – The GPIO port to read from.

Returns

the value on the port's pins.

inline void `rtos_gpio_port_out`(*rtos_gpio_t* *ctx, *rtos_gpio_port_id_t* port_id, uint32_t value)

Outputs a value to a GPIO port's pins.

Parameters

- `ctx` – A pointer to the GPIO driver instance to use.
- `port_id` – The GPIO port to write to.
- `value` – The value to write to the GPIO port.

inline void `rtos_gpio_isr_callback_set`(*rtos_gpio_t* *ctx, *rtos_gpio_port_id_t* port_id, *rtos_gpio_isr_cb_t* cb, void *app_data)

Sets the application callback function to be called when there is an interrupt on a GPIO port.

This must be called prior to enabling interrupts on `port_id`. It is also safe to be called while interrupts are enabled on it.

Parameters

- `ctx` – A pointer to the GPIO driver instance to use.
- `port_id` – Interrupts triggered by this port will call the application callback function `cb`.

- `cb` – The application callback function to call when there is an interrupt triggered by the port `port_id`.
- `app_data` – A pointer to application specific data to pass to the application callback function `cb`.

inline void `rtos_gpio_interrupt_enable`(*rtos_gpio_t* *ctx, *rtos_gpio_port_id_t* port_id)

Enables interrupts on a GPIO port. Interrupts are triggered whenever the value on the port changes.

Parameters

- `ctx` – A pointer to the GPIO driver instance to use.
- `port_id` – The GPIO port to enable interrupts on.

inline void `rtos_gpio_interrupt_disable`(*rtos_gpio_t* *ctx, *rtos_gpio_port_id_t* port_id)

Disables interrupts on a GPIO port.

Parameters

- `ctx` – A pointer to the GPIO driver instance to use.
- `port_id` – The GPIO port to disable interrupts on.

inline void `rtos_gpio_port_drive`(*rtos_gpio_t* *ctx, *rtos_gpio_port_id_t* port_id)

Configures a port in drive mode. Output values will be driven on the pins. This is the default drive state of a port. This has the side effect of disabling the port's internal pull-up and pull down resistors.

Parameters

- `ctx` – A pointer to the GPIO driver instance to use.
- `port_id` – The GPIO port to set to drive mode.

inline void `rtos_gpio_port_drive_low`(*rtos_gpio_t* *ctx, *rtos_gpio_port_id_t* port_id)

Configures a port in drive low mode. When the output value is 0 the pin is driven low, otherwise no value is driven. This has the side effect of enabled the port's internal pull-up resistor.

Parameters

- `ctx` – A pointer to the GPIO driver instance to use.
- `port_id` – The GPIO port to set to drive mode low.

inline void `rtos_gpio_port_drive_high`(*rtos_gpio_t* *ctx, *rtos_gpio_port_id_t* port_id)

Configures a port in drive high mode. When the output value is 1 the pin is driven high, otherwise no value is driven. This has the side effect of enabled the port's internal pull-down resistor.

Parameters

- `ctx` – A pointer to the GPIO driver instance to use.
- `port_id` – The GPIO port to set to drive mode high.

inline void `rtos_gpio_port_pull_none`(*rtos_gpio_t* *ctx, *rtos_gpio_port_id_t* port_id)

Disables the port's internal pull-up and pull down resistors.

Parameters

- `ctx` – A pointer to the GPIO driver instance to use.
- `port_id` – The GPIO port to set to pull none mode.

inline void `rtos_gpio_port_pull_up`(*rtos_gpio_t* *ctx, *rtos_gpio_port_id_t* port_id)

Enables the port's internal pull-up resistor.

Parameters

- `ctx` – A pointer to the GPIO driver instance to use.
- `port_id` – The GPIO port to set to pull up mode.

inline void `rtos_gpio_port_pull_down`(*rtos_gpio_t* *ctx, *rtos_gpio_port_id_t* port_id)

Enables the port's internal pull-down resistor.

Parameters

- `ctx` – A pointer to the GPIO driver instance to use.
- `port_id` – The GPIO port to set to pull down mode.

inline void `rtos_gpio_write_control_word`(*rtos_gpio_t* *ctx, *rtos_gpio_port_id_t* port_id, uint32_t value)

Configures the port control word value

Parameters

- `ctx` – A pointer to the GPIO driver instance to use.
- `port_id` – The GPIO port to modify
- `value` – The value to set the control word to

RPC Initialization API The following functions may be used to share a GPIO driver instance with other xcore tiles. Tiles that the driver instance is shared with may call any of the core functions listed above.

void `rtos_gpio_rpc_client_init`(*rtos_gpio_t* *gpio_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *host_intertile_ctx)

Initializes an RTOS GPIO driver instance on a client tile. This allows a tile that does not own the actual driver instance to use a driver instance on another tile. This will be called instead of *rtos_gpio_init()*. The host tile that owns the actual instance must simultaneously call *rtos_gpio_rpc_host_init()*.

Parameters

- `gpio_ctx` – A pointer to the GPIO driver instance to initialize.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `gpio_ctx`.
- `host_intertile_ctx` – A pointer to the intertile driver instance to use for performing the communication between the client and host tiles. This must have the same scope as `gpio_ctx`.

void `rtos_gpio_rpc_host_init`(*rtos_gpio_t* *gpio_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *client_intertile_ctx[], size_t remote_client_count)

Performs additional initialization on a GPIO driver instance to allow client tiles to use the GPIO driver instance. Each client tile that will use this instance must simultaneously call *rtos_gpio_rpc_client_init()*.

Parameters

- `gpio_ctx` – A pointer to the GPIO driver instance to share with clients.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `gpio_ctx`.

- `client_intertile_ctx` – An array of pointers to the intertile driver instances to use for performing the communication between the host tile and each client tile. This must have the same scope as `gpio_ctx`.
- `remote_client_count` – The number of client tiles to share this driver instance with.

void `rtos_gpio_rpc_config`(*rtos_gpio_t* *gpio_ctx, unsigned intertile_port, unsigned host_task_priority)

Configures the RPC for a GPIO driver instance. This must be called by both the host tile and all client tiles.

On the client tiles this must be called after calling *rtos_gpio_rpc_client_init()*. After calling this, the client tile may immediately begin to call the core GPIO functions on this driver instance. It does not need to wait for the host to call *rtos_gpio_start()*.

On the host tile this must be called both after calling *rtos_gpio_rpc_host_init()* and before calling *rtos_gpio_start()*.

Parameters

- `gpio_ctx` – A pointer to the GPIO driver instance to configure the RPC for.
- `intertile_port` – The port number on the intertile channel to use for transferring the RPC requests and responses for this driver instance. This port must not be shared by any other functions. The port must be the same for the host and all its clients.
- `host_task_priority` – The priority to use for the task on the host tile that handles RPC requests from the clients.

I²C RTOS Driver

This driver can be used to instantiate and control an I²C master or slave mode I/O interface on xcore in an RTOS application.

I²C Master RTOS Driver This driver can be used to instantiate and control an I²C master I/O interface on xcore in an RTOS application.

I²C Master Initialization API The following structures and functions are used to initialize and start an I²C driver instance.

typedef struct *rtos_i2c_master_struct* `rtos_i2c_master_t`

Typedef to the RTOS I2C master driver instance struct.

void `rtos_i2c_master_start`(*rtos_i2c_master_t* *i2c_master_ctx)

Starts an RTOS I2C master driver instance. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before any of the core I2C master driver functions are called with this instance.

rtos_i2c_master_init() must be called on this I2C master driver instance prior to calling this.

Parameters

- `i2c_master_ctx` – A pointer to the I2C master driver instance to start.

void `rtos_i2c_master_init`(*rtos_i2c_master_t* *i2c_master_ctx, const port_t p_scl, const uint32_t scl_bit_position, const uint32_t scl_other_bits_mask, const port_t p_sda, const uint32_t sda_bit_position, const uint32_t sda_other_bits_mask, hwtimer_t tmr, const unsigned kbits_per_second)

Initializes an RTOS I2C master driver instance. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before calling *rtos_i2c_master_start()* or any of the core I2C master driver functions with this instance.

Parameters

- `i2c_master_ctx` – A pointer to the I2C master driver instance to initialize.
- `p_scl` – The port containing SCL. This may be either the same as or different than `p_sda`.
- `scl_bit_position` – The bit number of the SCL line on the port `p_scl`.
- `scl_other_bits_mask` – A value that is ORed into the port value driven to `p_scl` both when SCL is high and low. The bit representing SCL (as well as SDA if they share the same port) must be set to 0.
- `p_sda` – The port containing SDA. This may be either the same as or different than `p_scl`.
- `sda_bit_position` – The bit number of the SDA line on the port `p_sda`.
- `sda_other_bits_mask` – A value that is ORed into the port value driven to `p_sda` both when SDA is high and low. The bit representing SDA (as well as SCL if they share the same port) must be set to 0.
- `tmr` – This is unused and should be set to 0. This will be removed.
- `kbits_per_second` – The speed of the I2C bus. The maximum value allowed is 400.

struct `rtos_i2c_master_struct`

#include <rtos_i2c_master.h> Struct representing an RTOS I2C master driver instance.

The members in this struct should not be accessed directly.

I²C Master Core API The following functions are the core I²C driver functions that are used after it has been initialized and started.

inline i2c_res_t `rtos_i2c_master_write`(*rtos_i2c_master_t* *ctx, uint8_t device_addr, uint8_t buf[], size_t n, size_t *num_bytes_sent, int send_stop_bit)

Writes data to an I2C bus as a master.

Parameters

- `ctx` – A pointer to the I2C master driver instance to use.
- `device_addr` – The address of the device to write to.
- `buf` – The buffer containing data to write.
- `n` – The number of bytes to write.
- `num_bytes_sent` – The function will set this value to the number of bytes actually sent. On success, this will be equal to `n` but it will be less if the slave sends an early NACK on the bus and the transaction fails.
- `send_stop_bit` – If this is non-zero then a stop bit will be sent on the bus after the transaction. This is usually required for normal operation. If this parameter is zero then no stop bit will be omitted. In this case, no other task can use the component until a stop bit has been sent.

Return values

- `<tt>I2C_ACK</tt>` – if the write was acknowledged by the device.

- <tt>I2C_NACK</tt>otherwise. –

inline i2c_res_t rtos_i2c_master_read(*rtos_i2c_master_t* *ctx, uint8_t device_addr, uint8_t buf[], size_t n, int send_stop_bit)

Reads data from an I2C bus as a master.

Parameters

- ctx – A pointer to the I2C master driver instance to use.
- device_addr – The address of the device to read from.
- buf – The buffer to fill with data.
- n – The number of bytes to read.
- send_stop_bit – If this is non-zero then a stop bit. will be sent on the bus after the transaction. This is usually required for normal operation. If this parameter is zero then no stop bit will be omitted. In this case, no other task can use the component until a stop bit has been sent.

Return values

- <tt>I2C_ACK</tt> – if the read was acknowledged by the device.
- <tt>I2C_NACK</tt>otherwise. –

inline void rtos_i2c_master_stop_bit_send(*rtos_i2c_master_t* *ctx)

Send a stop bit to an I2C bus as a master.

This function will cause a stop bit to be sent on the bus. It should be used to complete/abort a transaction if the send_stop_bit argument was not set when calling the *rtos_i2c_master_read()* or *rtos_i2c_master_write()* functions.

Parameters

- ctx – A pointer to the I2C master driver instance to use.

inline i2c_regop_res_t rtos_i2c_master_reg_write(*rtos_i2c_master_t* *ctx, uint8_t device_addr, uint8_t reg_addr, uint8_t data)

Write to an 8-bit register on an I2C device.

This function writes to an 8-bit addressed, 8-bit register in an I2C device. The function writes the data by sending the register address followed by the register data to the device at the specified device address.

Parameters

- ctx – A pointer to the I2C master driver instance to use.
- device_addr – The address of the device to write to.
- reg_addr – The address of the register to write to.
- data – The 8-bit value to write.

Return values

- <tt>I2C_REGOP_DEVICE_NACK</tt> – if the address is NACKed.
- <tt>I2C_REGOP_INCOMPLETE</tt> – if not all data was ACKed.
- <tt>I2C_REGOP_SUCCESS</tt> – on successful completion of the write.

inline i2c_regop_res_t `rtos_i2c_master_reg_read`(*rtos_i2c_master_t* *ctx, uint8_t device_addr, uint8_t reg_addr, uint8_t *data)

Reads from an 8-bit register on an I2C device.

This function reads from an 8-bit addressed, 8-bit register in an I2C device. The function reads the data by sending the register address followed reading the register data from the device at the specified device address.

Note that no stop bit is transmitted between the write and the read. The operation is performed as one transaction using a repeated start.

Parameters

- `ctx` – A pointer to the I2C master driver instance to use.
- `device_addr` – The address of the device to read from.
- `reg_addr` – The address of the register to read from.
- `data` – A pointer to the byte to fill with data read from the register.

Return values

- `<tt>I2C_REGOP_DEVICE_NACK</tt>` – if the device NACKed.
- `<tt>I2C_REGOP_SUCCESS</tt>` – on successful completion of the read.

I²C Master RPC Initialization API The following functions may be used to share a I²C driver instance with other xcore tiles. Tiles that the driver instance is shared with may call any of the core functions listed above.

void `rtos_i2c_master_rpc_client_init`(*rtos_i2c_master_t* *i2c_master_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *host_intertile_ctx)

Initializes an RTOS I2C master driver instance on a client tile. This allows a tile that does not own the actual driver instance to use a driver instance on another tile. This will be called instead of *rtos_i2c_master_init()*. The host tile that owns the actual instance must simultaneously call *rtos_i2c_master_rpc_host_init()*.

Parameters

- `i2c_master_ctx` – A pointer to the I2C master driver instance to initialize.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `i2c_master_ctx`.
- `host_intertile_ctx` – A pointer to the intertile driver instance to use for performing the communication between the client and host tiles. This must have the same scope as `i2c_master_ctx`.

void `rtos_i2c_master_rpc_host_init`(*rtos_i2c_master_t* *i2c_master_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *client_intertile_ctx[], size_t remote_client_count)

Performs additional initialization on an I2C master driver instance to allow client tiles to use the I2C master driver instance. Each client tile that will use this instance must simultaneously call *rtos_i2c_master_rpc_client_init()*.

Parameters

- `i2c_master_ctx` – A pointer to the I2C master driver instance to share with clients.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `i2c_master_ctx`.

- `client_intertile_ctx` – An array of pointers to the intertile driver instances to use for performing the communication between the host tile and each client tile. This must have the same scope as `i2c_master_ctx`.
- `remote_client_count` – The number of client tiles to share this driver instance with.

void `rtos_i2c_master_rpc_config`(*rtos_i2c_master_t* *i2c_master_ctx, unsigned intertile_port, unsigned host_task_priority)

Configures the RPC for an I2C master driver instance. This must be called by both the host tile and all client tiles.

On the client tiles this must be called after calling *rtos_i2c_master_rpc_client_init()*. After calling this, the client tile may immediately begin to call the core I2C master functions on this driver instance. It does not need to wait for the host to call *rtos_i2c_master_start()*.

On the host tile this must be called both after calling *rtos_i2c_master_rpc_host_init()* and before calling *rtos_i2c_master_start()*.

Parameters

- `i2c_master_ctx` – A pointer to the I2C master driver instance to configure the RPC for.
- `intertile_port` – The port number on the intertile channel to use for transferring the RPC requests and responses for this driver instance. This port must not be shared by any other functions. The port must be the same for the host and all its clients.
- `host_task_priority` – The priority to use for the task on the host tile that handles RPC requests from the clients.

I²C Slave RTOS Driver This driver can be used to instantiate and control an I²C slave I/O interface on xcore in an RTOS application.

I²C Slave API The following structures and functions are used to initialize and start an I²C driver instance.

typedef struct *rtos_i2c_slave_struct* `rtos_i2c_slave_t`

Typedef to the RTOS I2C slave driver instance struct.

typedef void (*`rtos_i2c_slave_start_cb_t`)(*rtos_i2c_slave_t* *ctx, void *app_data)

Function pointer type for application provided RTOS I2C slave start callback functions.

These callback functions are optionally called by an I2C slave driver's thread when it is first started. This gives the application a chance to perform startup initialization from within the driver's thread.

Param ctx
A pointer to the associated I2C slave driver instance.

Param app_data
A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

typedef void (*`rtos_i2c_slave_rx_cb_t`)(*rtos_i2c_slave_t* *ctx, void *app_data, uint8_t *data, size_t len)

Function pointer type for application provided RTOS I2C slave receive callback functions.

These callback functions are called when an I2C slave driver instance has received data from a master device.

Param ctx
A pointer to the associated I2C slave driver instance.

Param app_data
A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

Param data
A pointer to the data received from the master.

Param len
The number of valid bytes in `data`.

typedef size_t (*`rtos_i2c_slave_tx_start_cb_t`)(*rtos_i2c_slave_t* *ctx, void *app_data, uint8_t **data)

Function pointer type for application provided RTOS I2C slave transmit start callback functions.

These callback functions are called when an I2C slave driver instance needs to transmit data to a master device. This callback must provide the data to transmit and the length.

Param ctx
A pointer to the associated I2C slave driver instance.

Param app_data
A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

Param data
A pointer to the data buffer to transmit to the master. The driver sets this to its internal data buffer, which has a size of RTOS_I2C_SLAVE_BUF_LEN, prior to calling this callback. This may be set to a different buffer by the callback. The callback must fill this buffer with the data to send to the master.

Return
The number of bytes to transmit to the master from `data`. If the master reads more bytes than this, the driver will wrap around to the start of the buffer and send it again.

typedef void (*`rtos_i2c_slave_tx_done_cb_t`)(*rtos_i2c_slave_t* *ctx, void *app_data, uint8_t *data, size_t len)

Function pointer type for application provided RTOS I2C slave transmit done callback functions.

These callback functions are optionally called when an I2C slave driver instance is done transmitting data to a master device. A buffer to the data sent and the actual number of bytes sent are provided to the callback.

The application may want to use this, for example, if the buffer that was sent was malloc'd. This callback can be used to free the buffer.

Param ctx
A pointer to the associated I2C slave driver instance.

Param app_data
A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

Param data
A pointer to the data transmitted to the master.

Param len
The number of bytes transmitted to the master from `data`.

typedef void (*`rtos_i2c_slave_rx_byte_check_cb_t`)(*rtos_i2c_slave_t* *ctx, void *app_data, uint8_t data, i2c_slave_ack_t *cur_status)

Function pointer type for application provided function to check bytes received from master individually.

This callback function is called once per byte received from the master device.

The application may want to use this, for example, to check byte by byte and force a NACK for an unexpected payload.

The user provided functions must be marked with RTOS_I2C_SLAVE_MASTER_SENT_BYTE_CHECK_CALLBACK_ATTR.

Param ctx
A pointer to the associated I2C slave driver instance.

Param app_data
A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

Param data
A copy of the most recent byte of data transmitted from the master.

Param cur_status
A pointer to the current ACK/NACK response for this byte. The application may change this to I2C_SLAVE_ACK or I2C_SLAVE_NACK. If cur_status is returned as an invalid value, the driver will implicitly NACK.

typedef void (*`rtos_i2c_slave_write_addr_request_cb_t`)(*rtos_i2c_slave_t* *ctx, void *app_data, i2c_slave_ack_t *cur_status)

Function pointer type for application provided function to alert application that there is a write transaction incoming from master

This allows an application to NACK if it is not ready for handling write requests.

The user provided functions must be marked with RTOS_I2C_SLAVE_WRITE_ADDR_REQUEST_CALLBACK_ATTR.

Param ctx
A pointer to the associated I2C slave driver instance.

Param app_data
A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

Param cur_status
A pointer to the current ACK/NACK response for this byte. The application may change this to I2C_SLAVE_ACK or I2C_SLAVE_NACK. If cur_status is returned as an invalid value, the driver will implicitly NACK. By default the driver will implicitly ACK.

void `rtos_i2c_slave_start`(*rtos_i2c_slave_t* *i2c_slave_ctx, void *app_data, *rtos_i2c_slave_start_cb_t* start, *rtos_i2c_slave_rx_cb_t* rx, *rtos_i2c_slave_tx_start_cb_t* tx_start, *rtos_i2c_slave_tx_done_cb_t* tx_done, *rtos_i2c_slave_rx_byte_check_cb_t* rx_byte_check, *rtos_i2c_slave_write_addr_request_cb_t* write_addr_req, unsigned interrupt_core_id, unsigned priority)

Starts an RTOS I2C slave driver instance. This must only be called by the tile that owns the driver instance. It must be called after starting the RTOS from an RTOS thread.

rtos_i2c_slave_init() must be called on this I2C slave driver instance prior to calling this.

Parameters

- `i2c_slave_ctx` – A pointer to the I2C slave driver instance to start.
- `app_data` – A pointer to application specific data to pass to the callback functions.

- `start` – The callback function that is called when the driver's thread starts. This is optional and may be NULL.
- `rx` – The callback function to receive data from the bus master.
- `tx_start` – The callback function to transmit data to the bus master.
- `tx_done` – The callback function that is notified when transmits are complete. This is optional and may be NULL.
- `rx_byte_check` – The callback function to check received bytes individually.
- `write_addr_req` – The callback function to alert an incoming write request
- `interrupt_core_id` – The ID of the core on which to enable the I2C interrupt.
- `priority` – The priority of the task that gets created by the driver to call the callback functions.

void `rtos_i2c_slave_init`(*rtos_i2c_slave_t* *i2c_slave_ctx, uint32_t io_core_mask, const port_t p_scl, const port_t p_sda, uint8_t device_addr)

Initializes an RTOS I2C slave driver instance. This must only be called by the tile that owns the driver instance. It should be called before starting the RTOS, and must be called before calling *rtos_i2c_slave_start()*.

Parameters

- `i2c_slave_ctx` – A pointer to the I2C slave driver instance to initialize.
- `io_core_mask` – A bitmask representing the cores on which the low level I2C I/O thread created by the driver is allowed to run. Bit 0 is core 0, bit 1 is core 1, etc.
- `p_scl` – The port containing SCL. This must be a 1-bit port and different than `p_sda`.
- `p_sda` – The port containing SDA. This must be a 1-bit port and different than `p_scl`.
- `device_addr` – The 7-bit address of the slave device.

`RTOS_I2C_SLAVE_BUF_LEN`

The maximum number of bytes that a the RTOS I2C slave driver can receive from a master in a single write transaction.

`RTOS_I2C_SLAVE_CALLBACK_ATTR`

This attribute must be specified on all RTOS I2C slave callback functions provided by the application.

`RTOS_I2C_SLAVE_RX_BYTE_CHECK_CALLBACK_ATTR`

This attribute must be specified on all RTOS I2C slave rtos_i2c_slave_rx_byte_check_cb_t provided by the application.

`RTOS_I2C_SLAVE_WRITE_ADDR_REQUEST_CALLBACK_ATTR`

This attribute must be specified on all RTOS I2C slave rtos_i2c_slave_write_addr_request_cb_t provided by the application.

struct `rtos_i2c_slave_struct`

#include <rtos_i2c_slave.h> Struct representing an RTOS I2C slave driver instance.

The members in this struct should not be accessed directly.

I²S RTOS Driver

This driver can be used to instantiate and control an I²S master or slave mode I/O interface on xcore in an RTOS application.

Initialization API The following structures and functions are used to initialize and start an I²S driver instance.

I²S Master Initialization API The following structures and functions are used to initialize and start an I²S master driver instance.

void `rtos_i2s_master_init`(*rtos_i2s_t* *i2s_ctx, uint32_t io_core_mask, port_t p_dout[], size_t num_out, port_t p_din[], size_t num_in, port_t p_bclk, port_t p_lrclk, port_t p_mclk, xclock_t bclk)

Initializes an RTOS I2S driver instance in master mode. This must only be called by the tile that owns the driver instance. It should be called before starting the RTOS, and must be called before calling *rtos_i2s_start()* or any of the core I2S driver functions with this instance.

Parameters

- `i2s_ctx` – A pointer to the I2S driver instance to initialize.
- `io_core_mask` – A bitmask representing the cores on which the low level I2S I/O thread created by the driver is allowed to run. Bit 0 is core 0, bit 1 is core 1, etc.
- `p_dout` – An array of data output ports.
- `num_out` – The number of output data ports.
- `p_din` – An array of data input ports.
- `num_in` – The number of input data ports.
- `p_bclk` – The bit clock output port.
- `p_lrclk` – The word clock output port.
- `p_mclk` – Input port which supplies the master clock.
- `bclk` – A clock that will get configured for use with the bit clock.

void `rtos_i2s_master_ext_clock_init`(*rtos_i2s_t* *i2s_ctx, uint32_t io_core_mask, port_t p_dout[], size_t num_out, port_t p_din[], size_t num_in, port_t p_bclk, port_t p_lrclk, xclock_t bclk)

Initializes an RTOS I2S driver instance in master mode but that uses an externally generated bit clock. This must only be called by the tile that owns the driver instance. It should be called before starting the RTOS, and must be called before calling *rtos_i2s_start()* or any of the core I2S driver functions with this instance.

Parameters

- `i2s_ctx` – A pointer to the I2S driver instance to initialize.
- `io_core_mask` – A bitmask representing the cores on which the low level I2S I/O thread created by the driver is allowed to run. Bit 0 is core 0, bit 1 is core 1, etc.
- `p_dout` – An array of data output ports.
- `num_out` – The number of output data ports.
- `p_din` – An array of data input ports.
- `num_in` – The number of input data ports.
- `p_bclk` – The bit clock output port.

- `p_lrclk` – The word clock output port.
- `bclk` – A clock that is configured externally to be used as the bit clock

I²S Slave Initialization API The following structures and functions are used to initialize and start an I²S slave driver instance.

void `rtos_i2s_slave_init`(*rtos_i2s_t* *i2s_ctx, uint32_t io_core_mask, port_t p_dout[], size_t num_out, port_t p_din[], size_t num_in, port_t p_bclk, port_t p_lrclk, xclock_t bclk)

Initializes an RTOS I2S driver instance in slave mode. This must only be called by the tile that owns the driver instance. It should be called before starting the RTOS, and must be called before calling *rtos_i2s_start()* or any of the core I2S driver functions with this instance.

Parameters

- `i2s_ctx` – A pointer to the I2S driver instance to initialize.
- `io_core_mask` – A bitmask representing the cores on which the low level I2S I/O thread created by the driver is allowed to run. Bit 0 is core 0, bit 1 is core 1, etc.
- `p_dout` – An array of data output ports.
- `num_out` – The number of output data ports.
- `p_din` – An array of data input ports.
- `num_in` – The number of input data ports.
- `p_bclk` – The bit clock input port.
- `p_lrclk` – The word clock input port.
- `bclk` – A clock that will get configured for use with the bit clock.

typedef struct *rtos_i2s_struct* `rtos_i2s_t`

Typedef to the RTOS I2S driver instance struct.

typedef size_t (*`rtos_i2s_send_filter_cb_t`)(*rtos_i2s_t* *ctx, void *app_data, int32_t *i2s_frame, size_t i2s_frame_size, int32_t *send_buf, size_t samples_available)

Function pointer type for application provided RTOS I2S send filter callback functions.

These callback functions are called when an I2S driver instance needs output the next audio frame to its interface. By default, audio frames in the driver's send buffer are output directly to its interface. However, this gives the application an opportunity to override this and provide filtering.

These functions must not block.

Param ctx
A pointer to the associated I2C slave driver instance.

Param app_data
A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

Param i2s_frame
A pointer to the buffer where the callback should write the next frame to send.

Param i2s_frame_size
The number of samples that should be written to `i2s_frame`.

Param send_buf
A pointer to the next frame in the driver's send buffer. The callback should use this as the input to its filter.

Param samples_available
The number of samples available in `send_buf`.

Return
the number of samples read out of `send_buf`.

typedef size_t (*`rtos_i2s_receive_filter_cb_t`)(*rtos_i2s_t* *ctx, void *app_data, int32_t *i2s_frame, size_t i2s_frame_size, int32_t *receive_buf, size_t sample_spaces_free)

Function pointer type for application provided RTOS I2S receive filter callback functions.

These callback functions are called when an I2S driver instance has received the next audio frame from its interface. By default, audio frames received from the driver's interface are put directly into its receive buffer. However, this gives the application an opportunity to override this and provide filtering.

These functions must not block.

Param ctx
A pointer to the associated I2C slave driver instance.

Param app_data
A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

Param i2s_frame
A pointer to the buffer where the callback should read the next received frame from The callback should use this as the input to its filter.

Param i2s_frame_size
The number of samples that should be read from `i2s_frame`.

Param receive_buf
A pointer to the next frame in the driver's send buffer. The callback should use this as the input to its filter.

Param sample_spaces_free
The number of sample spaces free in `receive_buf`.

Return
the number of samples written to `receive_buf`.

inline int `rtos_i2s_mclk_bclk_ratio(`const unsigned audio_clock_frequency, const unsigned sample_rate`)`

Helper function to calculate the MCLK/BCLK ratio given the audio clock frequency at the master clock pin and the desired sample rate.

Parameters

- `audio_clock_frequency` – The frequency of the audio clock at the port p_mclk.
- `sample_rate` – The desired sample rate.

Returns
the MCLK/BCLK ratio that should be provided to *rtos_i2s_start()*.

inline void `rtos_i2s_send_filter_cb_set(`*rtos_i2s_t* *ctx, *rtos_i2s_send_filter_cb_t* send_filter_cb, void *send_filter_app_data`)`

inline void `rtos_i2s_receive_filter_cb_set`(*rtos_i2s_t* *ctx, *rtos_i2s_receive_filter_cb_t* receive_filter_cb, void *receive_filter_app_data)

void `rtos_i2s_start`(*rtos_i2s_t* *i2s_ctx, unsigned mclk_bclk_ratio, i2s_mode_t mode, size_t recv_buffer_size, size_t send_buffer_size, unsigned interrupt_core_id)

Starts an RTOS I2S driver instance. This must only be called by the tile that owns the driver instance. It must be called after starting the RTOS from an RTOS thread, and must be called before any of the core I2S driver functions are called with this instance.

One of *rtos_i2s_master_init()*, rtos_i2s_master_ext_clock_init, or *rtos_i2s_slave_init()* must be called on this I2S driver instance prior to calling this.

Parameters

- `i2s_ctx` – A pointer to the I2S driver instance to start.
- `mclk_bclk_ratio` – The master clock to bit clock ratio. This may be computed by the helper function *rtos_i2s_mclk_bclk_ratio()*. This is only used if the I2S instance was initialized with *rtos_i2s_master_init()*. Otherwise it is ignored.
- `mode` – The mode of the LR clock. See i2s_mode_t.
- `recv_buffer_size` – The size in frames of the input buffer. Each frame is two samples (left and right channels) per input port. For example, a size of two here when num_in is three would create a buffer that holds up to 12 samples.
- `send_buffer_size` – The size in frames of the output buffer. Each frame is two samples (left and right channels) per output port. For example, a size of two here when num_out is three would create a buffer that holds up to 12 samples. Frames transmitted by *rtos_i2s_tx()* are stored in this buffers before they are sent out to the I2S interface.
- `interrupt_core_id` – The ID of the core on which to enable the I2S interrupt.

`RTOS_I2S_APP_SEND_FILTER_CALLBACK_ATTR`

This attribute must be specified on all RTOS I2S send filter callback functions provided by the application.

`RTOS_I2S_APP_RECEIVE_FILTER_CALLBACK_ATTR`

This attribute must be specified on all RTOS I2S receive filter callback functions provided by the application.

struct `rtos_i2s_struct`

#include <rtos_i2s.h> Struct representing an RTOS I2S driver instance.

The members in this struct should not be accessed directly.

Core API The following functions are the core I^{2}S driver functions that are used after it has been initialized and started.

inline size_t `rtos_i2s_rx`(*rtos_i2s_t* *ctx, int32_t *i2s_sample_buf, size_t frame_count, unsigned timeout)

Receives sample frames from the I2S interface.

This function will block until new frames are available.

Parameters

- `ctx` – A pointer to the I2S driver instance to use.
- `i2s_sample_buf` – A buffer to copy the received sample frames into.

- `frame_count` – The number of frames to receive from the buffer. This must be less than or equal to the size of the input buffer specified to *rtos_i2s_start()*.
- `timeout` – The amount of time to wait before the requested number of frames becomes available.

Returns
The number of frames actually received into `i2s_sample_buf`.

inline size_t `rtos_i2s_tx`(*rtos_i2s_t* *ctx, int32_t *i2s_sample_buf, size_t frame_count, unsigned timeout)

Transmits sample frames out to the I2S interface.

The samples are stored into a buffer and are not necessarily sent out to the I2S interface before this function returns.

Parameters

- `ctx` – A pointer to the I2S driver instance to use.
- `i2s_sample_buf` – A buffer containing the sample frames to transmit out to the I2S interface.
- `frame_count` – The number of frames to transmit out from the buffer. This must be less than or equal to the size of the output buffer specified to *rtos_i2s_start()*.
- `timeout` – The amount of time to wait before there is enough space in the send buffer to accept the frames to be transmitted.

Returns
The number of frames actually stored into the buffer.

RPC Initialization API The following functions may be used to share a I^2S driver instance with other xcore tiles. Tiles that the driver instance is shared with may call any of the core functions listed above.

void `rtos_i2s_rpc_client_init`(*rtos_i2s_t* *i2s_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *host_intertile_ctx)

Initializes an RTOS I2S driver instance on a client tile. This allows a tile that does not own the actual driver instance to use a driver instance on another tile. This will be called instead of on of the RTOS I2S init functions. The host tile that owns the actual instance must simultaneously call *rtos_i2s_rpc_host_init()*.

Parameters

- `i2s_ctx` – A pointer to the I2S driver instance to initialize.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `i2s_ctx`.
- `host_intertile_ctx` – A pointer to the intertile driver instance to use for performing the communication between the client and host tiles. This must have the same scope as `i2s_ctx`.

void `rtos_i2s_rpc_host_init`(*rtos_i2s_t* *i2s_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *client_intertile_ctx[], size_t remote_client_count)

Performs additional initialization on a I2S driver instance to allow client tiles to use the I2S driver instance. Each client tile that will use this instance must simultaneously call *rtos_i2s_rpc_client_init()*.

Parameters

- `i2s_ctx` – A pointer to the I2S driver instance to share with clients.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `i2s_ctx`.

- `client_intertile_ctx` – An array of pointers to the intertile driver instances to use for performing the communication between the host tile and each client tile. This must have the same scope as `i2s_ctx`.
- `remote_client_count` – The number of client tiles to share this driver instance with.

void `rtos_i2s_rpc_config`(*rtos_i2s_t* *i2s_ctx, unsigned intertile_port, unsigned host_task_priority)

Configures the RPC for a I2S driver instance. This must be called by both the host tile and all client tiles.

On the client tiles this must be called after calling *rtos_i2s_rpc_client_init()*. After calling this, the client tile may immediately begin to call the core I2S functions on this driver instance. It does not need to wait for the host to call *rtos_i2s_start()*.

On the host tile this must be called both after calling *rtos_i2s_rpc_host_init()* and before calling *rtos_i2s_start()*.

Parameters

- `i2s_ctx` – A pointer to the I2S driver instance to configure the RPC for.
- `intertile_port` – The port number on the intertile channel to use for transferring the RPC requests and responses for this driver instance. This port must not be shared by any other functions. The port must be the same for the host and all its clients.
- `host_task_priority` – The priority to use for the task on the host tile that handles RPC requests from the clients.

Microphone Array RTOS Driver

This driver can be used to instantiate and control a dual DDR PDM microphone interface on xcore in an RTOS application.

Initialization API The following structures and functions are used to initialize and start a microphone array driver instance.

enum `rtos_mic_array_format_t`

Typedef for the RTOS mic array driver audio format

Values:

enumerator `RTOS_MIC_ARRAY_CHANNEL_SAMPLE`

enumerator `RTOS_MIC_ARRAY_SAMPLE_CHANNEL`

enumerator `RTOS_MIC_ARRAY_FORMAT_COUNT`

typedef struct *rtos_mic_array_struct* `rtos_mic_array_t`

Typedef to the RTOS mic array driver instance struct.

void `rtos_mic_array_start`(*rtos_mic_array_t* *mic_array_ctx, size_t buffer_size, unsigned interrupt_core_id)

Starts an RTOS mic array driver instance. This must only be called by the tile that owns the driver instance. It must be called after starting the RTOS from an RTOS thread, and must be called before any of the core mic array driver functions are called with this instance.

rtos_mic_array_init() must be called on this mic array driver instance prior to calling this.

Parameters

- `mic_array_ctx` – A pointer to the mic array driver instance to start.
- `buffer_size` – The size in frames of the input buffer. Each frame is two samples (one for each microphone) plus one sample per reference channel. This must be at least MIC_ARRAY_CONFIG_SAMPLES_PER_FRAME. Samples are pulled out of this buffer by the application by calling *rtos_mic_array_rx()*.
- `interrupt_core_id` – The ID of the core on which to enable the mic array interrupt.

void `rtos_mic_array_init`(*rtos_mic_array_t* *mic_array_ctx, uint32_t io_core_mask, *rtos_mic_array_format_t* format)

Initializes an RTOS mic array driver instance. This must only be called by the tile that owns the driver instance. It should be called before starting the RTOS, and must be called before calling *rtos_mic_array_start()* or any of the core mic array driver functions with this instance.

Parameters

- `mic_array_ctx` – A pointer to the mic array driver instance to initialize.
- `io_core_mask` – A bitmask representing the cores on which the low level mic array I/O thread created by the driver is allowed to run. Bit 0 is core 0, bit 1 is core 1, etc.
- `format` – Format of the output data

struct `rtos_mic_array_struct`

#include <rtos_mic_array.h> Struct representing an RTOS mic array driver instance.

The members in this struct should not be accessed directly.

Core API The following functions are the core microphone array driver functions that are used after it has been initialized and started.

inline size_t `rtos_mic_array_rx`(*rtos_mic_array_t* *ctx, int32_t **sample_buf, size_t frame_count, unsigned timeout)

Receives sample frames from the PDM mic array interface.

This function will block until new frames are available.

Parameters

- `ctx` – A pointer to the mic array driver instance to use.
- `sample_buf` – A buffer to copy the received sample frames into.
- `frame_count` – The number of frames to receive from the buffer. This must be less than or equal to the size of the buffer specified to *rtos_mic_array_start()* if in RTOS_MIC_ARRAY_SAMPLE_CHANNEL mode. This must be equal to MIC_ARRAY_CONFIG_SAMPLES_PER_FRAME if in RTOS_MIC_ARRAY_CHANNEL_SAMPLE mode.
- `timeout` – The amount of time to wait before the requested number of frames becomes available.

Returns

The number of frames actually received into `sample_buf`.

RPC Initialization API The following functions may be used to share a microphone array driver instance with other xcore tiles. Tiles that the driver instance is shared with may call any of the core functions listed above.

void `rtos_mic_array_rpc_client_init`(*rtos_mic_array_t* *mic_array_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *host_intertile_ctx)

Initializes an RTOS mic array driver instance on a client tile. This allows a tile that does not own the actual driver instance to use a driver instance on another tile. This will be called instead of *rtos_mic_array_init()*. The host tile that owns the actual instance must simultaneously call *rtos_mic_array_rpc_host_init()*.

Parameters

- `mic_array_ctx` – A pointer to the mic array driver instance to initialize.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `mic_array_ctx`.
- `host_intertile_ctx` – A pointer to the intertile driver instance to use for performing the communication between the client and host tiles. This must have the same scope as `mic_array_ctx`.

void `rtos_mic_array_rpc_host_init`(*rtos_mic_array_t* *mic_array_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *client_intertile_ctx[], size_t remote_client_count)

Performs additional initialization on a mic array driver instance to allow client tiles to use the mic array driver instance. Each client tile that will use this instance must simultaneously call *rtos_mic_array_rpc_client_init()*.

Parameters

- `mic_array_ctx` – A pointer to the mic array driver instance to share with clients.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `mic_array_ctx`.
- `client_intertile_ctx` – An array of pointers to the intertile driver instances to use for performing the communication between the host tile and each client tile. This must have the same scope as `mic_array_ctx`.
- `remote_client_count` – The number of client tiles to share this driver instance with.

void `rtos_mic_array_rpc_config`(*rtos_mic_array_t* *mic_array_ctx, unsigned intertile_port, unsigned host_task_priority)

Configures the RPC for a mic array driver instance. This must be called by both the host tile and all client tiles.

On the client tiles this must be called after calling *rtos_mic_array_rpc_client_init()*. After calling this, the client tile may immediately begin to call the core mic array functions on this driver instance. It does not need to wait for the host to call *rtos_mic_array_start()*.

On the host tile this must be called both after calling *rtos_mic_array_rpc_host_init()* and before calling *rtos_mic_array_start()*.

Parameters

- `mic_array_ctx` – A pointer to the mic array driver instance to configure the RPC for.
- `intertile_port` – The port number on the intertile channel to use for transferring the RPC requests and responses for this driver instance. This port must not be shared by any other functions. The port must be the same for the host and all its clients.
- `host_task_priority` – The priority to use for the task on the host tile that handles RPC requests from the clients.

QSPI Flash RTOS Driver

This driver can be used to instantiate and control a Quad SPI flash I/O interface on xcore in an RTOS application.

Initialization API The following structures and functions are used to initialize and start a QSPI flash driver instance.

typedef struct *rtos_qspi_flash_struct* `rtos_qspi_flash_t`

Typedef to the RTOS QSPI flash driver instance struct.

void `rtos_qspi_flash_start`(*rtos_qspi_flash_t* *ctx, unsigned priority)

Starts an RTOS QSPI flash driver instance. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before any of the core QSPI flash driver functions are called with this instance.

rtos_qspi_flash_init() must be called on this QSPI flash driver instance prior to calling this.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to start.
- `priority` – The priority of the task that gets created by the driver to handle the QSPI flash interface.

void `rtos_qspi_flash_op_core_affinity_set`(*rtos_qspi_flash_t* *ctx, uint32_t op_core_mask)

Sets the core affinity for a RTOS QSPI flash driver instance. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, and should be called before any of the core QSPI flash driver functions are called with this instance.

Since interrupts are disabled during the QSPI transaction on the op thread, a core mask is provided to allow users to avoid collisions with application ISRs.

rtos_qspi_flash_start() must be called on this QSPI flash driver instance prior to calling this.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to start.
- `op_core_mask` – A bitmask representing the cores on which the QSPI I/O thread created by the driver is allowed to run. Bit 0 is core 0, bit 1 is core 1, etc.

void `rtos_qspi_flash_init`(*rtos_qspi_flash_t* *ctx, xclock_t clock_block, port_t cs_port, port_t sclk_port, port_t sio_port, fl_QuadDeviceSpec *spec)

Initializes an RTOS QSPI flash driver instance. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before calling *rtos_qspi_flash_start()* or any of the core QSPI flash driver functions with this instance.

This function will initialize a flash driver using lib_quadflash for all operations.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to initialize.
- `clock_block` – The clock block to use for the qspi_io interface.
- `cs_port` – The chip select port. MUST be a 1-bit port.
- `sclk_port` – The SCLK port. MUST be a 1-bit port.
- `sio_port` – The SIO port. MUST be a 4-bit port.

- `spec` – A pointer to the flash part specification. This may be set to NULL to use the XTC default

void `rtos_qspi_flash_fast_read_init`(*rtos_qspi_flash_t* *ctx, xclock_t clock_block, port_t cs_port, port_t sclk_port, port_t sio_port, fl_QuadDeviceSpec *spec, qspi_fast_flash_read_transfer_mode_t read_mode, uint8_t read_divide, uint32_t calibration_pattern_addr)

Initializes an RTOS QSPI flash driver instance. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before calling *rtos_qspi_flash_start()* or any of the core QSPI flash driver functions with this instance.

This function will initialize a flash driver using lib_quadflash for erase and writes, and lib_qspi_fast_read for reads. If calibration fails the driver will enable lib_quadflash for reads and allow the application to decide what to do about the failed calibration. The status of the calibration can be checked at runtime by calling *rtos_qspi_flash_calibration_valid_get()*.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to initialize.
- `clock_block` – The clock block to use for the qspi_io interface.
- `cs_port` – The chip select port. MUST be a 1-bit port.
- `sclk_port` – The SCLK port. MUST be a 1-bit port.
- `sio_port` – The SIO port. MUST be a 4-bit port.
- `spec` – A pointer to the flash part specification. This may be set to NULL to use the XTC default
- `read_mode` – The transfer mode to use for port reads. Invalid values will default to qspi_fast_flash_read_transfer_raw
- `read_divide` – The divisor to use for QSPI SCLK.
- `calibration_pattern_addr` – The address of the default calibration pattern. This driver requires the default calibration pattern supplied with lib_qspi_fast_read and does not support custom patterns.

`RTOS_QSPI_FLASH_READ_CHUNK_SIZE`

struct `rtos_qspi_flash_struct`

#include <rtos_qspi_flash.h> Struct representing an RTOS QSPI flash driver instance.

The members in this struct should not be accessed directly.

Core API The following functions are the core QSPI flash driver functions that are used after it has been initialized and started.

inline void `rtos_qspi_flash_lock`(*rtos_qspi_flash_t* *ctx)

Obtains a lock for exclusive access to the QSPI flash. This allows a thread to perform a sequence of operations (such as read, modify, erase, write) without the risk of another thread issuing a command in the middle of the sequence and corrupting the data in the flash.

If only a single atomic operation needs to be performed, such as a read, it is not necessary to call this to obtain the lock first. Each individual operation obtains and releases the lock automatically so that they cannot run while another thread has the lock.

The lock MUST be released when it is no longer needed by calling *rtos_qspi_flash_unlock()*.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to lock.

inline void `rtos_qspi_flash_unlock`(*rtos_qspi_flash_t* *ctx)

Releases a lock for exclusive access to the QSPI flash. The lock must have already been obtained by calling *rtos_qspi_flash_lock()*.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to unlock.

inline void `rtos_qspi_flash_read`(*rtos_qspi_flash_t* *ctx, uint8_t *data, unsigned address, size_t len)

This reads data from the flash in quad I/O mode. All four lines are used to send the address and to read the data.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to use.
- `data` – Pointer to the buffer to save the read data to.
- `address` – The byte address in the flash to begin reading at. Only bits 23:0 contain the address. Bits 31:24 are ignored.
- `len` – The number of bytes to read and save to `data`.

inline void `rtos_qspi_flash_read_mode`(*rtos_qspi_flash_t* *ctx, uint8_t *data, unsigned address, size_t len, qspi_fast_flash_read_transfer_mode_t mode)

This reads data from the flash in quad I/O mode. All four lines are used to send the address and to read the data.

Note: This only works with fast flash read and successful calibration. See *rtos_qspi_flash_fast_read_init()* versus *rtos_qspi_flash_init()*

If used with non fast flash read setups, this function will behave exactly the same as *rtos_qspi_flash_read()*, regardless of the value of `mode`.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to use.
- `data` – Pointer to the buffer to save the read data to.
- `address` – The byte address in the flash to begin reading at. Only bits 23:0 contain the address. Bits 31:24 are ignored.
- `len` – The number of bytes to read and save to `data`.
- `mode` – The transfer mode for this read operation `data`.

int `rtos_qspi_flash_read_ll`(*rtos_qspi_flash_t* *ctx, uint8_t *data, unsigned address, size_t len)

This is a lower level version of *rtos_qspi_flash_read()* that is safe to call from within ISRs. If a task currently own the flash lock, or if another core is actively doing a read with this function, then the read will not be performed and an error returned. It is up to the application to determine what it should do in this situation and to avoid a potential deadlock.

This function may only be called on the same tile as the underlying peripheral.

This function uses the lib_quadflash API to perform the read. It is up to the application to ensure that XCORE resources are properly configured.

Note: It is not possible to call this from a task that currently owns the flash lock taken with *rtos_qspi_flash_lock()*. In general it is not advisable to call this from an RTOS task unless the small amount of overhead time that is introduced by *rtos_qspi_flash_read()* is unacceptable.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to use.
- `data` – Pointer to the buffer to save the read data to.
- `address` – The byte address in the flash to begin reading at. Only bits 23:0 contain the address. Bits 31:24 are ignored.
- `len` – The number of bytes to read and save to `data`.

Return values

- `0` – if the flash was available and the read operation was performed.
- `-1` – if the flash was unavailable and the read could not be performed.

int `rtos_qspi_flash_fast_read_ll`(*rtos_qspi_flash_t* *ctx, uint8_t *data, unsigned address, size_t len)

This is a lower level version of *rtos_qspi_flash_read()* that is safe to call from within ISRs. If a task currently own the flash lock, or if another core is actively doing a read with this function, then the read will not be performed and an error returned. It is up to the application to determine what it should do in this situation and to avoid a potential deadlock.

This function may only be called on the same tile as the underlying peripheral.

This function uses the lib_qspi_fast_read API to perform the read. It is up to the application to ensure that XCORE resources are properly configured.

Note: It is not possible to call this from a task that currently owns the flash lock taken with *rtos_qspi_flash_lock()*. In general it is not advisable to call this from an RTOS task unless the small amount of overhead time that is introduced by *rtos_qspi_flash_read()* is unacceptable.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to use.
- `data` – Pointer to the buffer to save the read data to.
- `address` – The byte address in the flash to begin reading at. Only bits 23:0 contain the address. Bits 31:24 are ignored.
- `len` – The number of bytes to read and save to `data`.

Return values

- `0` – if the flash was available and the read operation was performed.
- `-1` – if the flash was unavailable and the read could not be performed.

int `rtos_qspi_flash_fast_read_mode_ll`(*rtos_qspi_flash_t* *ctx, uint8_t *data, unsigned address, size_t len, qspi_fast_flash_read_transfer_mode_t mode)

This is a lower level version of *rtos_qspi_flash_read_mode()* that is safe to call from within ISRs. If a task currently own the flash lock, or if another core is actively doing a read with this function, then the read will

not be performed and an error returned. It is up to the application to determine what it should do in this situation and to avoid a potential deadlock.

This function may only be called on the same tile as the underlying peripheral.

This function uses the lib_qspi_fast_read API to perform the read. It is up to the application to ensure that XCORE resources are properly configured.

Note: It is not possible to call this from a task that currently owns the flash lock taken with *rtos_qspi_flash_lock()*. In general it is not advisable to call this from an RTOS task unless the small amount of overhead time that is introduced by *rtos_qspi_flash_read_mode()* is unacceptable.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to use.
- `data` – Pointer to the buffer to save the read data to.
- `address` – The byte address in the flash to begin reading at. Only bits 23:0 contain the address. Bits 31:24 are ignored.
- `len` – The number of bytes to read and save to `data`.
- `mode` – The transfer mode for this read operation `data`.

Return values

- `0` – if the flash was available and the read operation was performed.
- `-1` – if the flash was unavailable and the read could not be performed.

void `rtos_qspi_flash_fast_read_setup_ll`(*rtos_qspi_flash_t* *ctx)

This is a lower level function that enables the user to setup the ports for fast flash access.

This function may only be called on the same tile as the underlying peripheral.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to use.

void `rtos_qspi_flash_fast_read_shutdown_ll`(*rtos_qspi_flash_t* *ctx)

This is a lower level function that enables the user to shutdown low level usage to resume normal QSPI thread operation.

This function may only be called on the same tile as the underlying peripheral.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to use.

inline void `rtos_qspi_flash_write`(*rtos_qspi_flash_t* *ctx, const uint8_t *data, unsigned address, size_t len)

This writes data to the QSPI flash. The standard page program command is sent and only SIO0 (MOSI) is used to send the address and data.

The driver handles sending the write enable command, as well as waiting for the write to complete.

This function may return before the write operation is complete, as the actual write operation is queued and executed by a thread created by the driver.

Note: this function does NOT erase the flash first. Erase operations must be explicitly requested by the application.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to use.
- `data` – Pointer to the data to write to the flash.
- `address` – The byte address in the flash to begin writing at. Only bits 23:0 contain the address. The byte in bits 31:24 is not sent.
- `len` – The number of bytes to write to the flash.

inline void `rtos_qspi_flash_erase`(*rtos_qspi_flash_t* *ctx, unsigned address, size_t len)

This erases data from the QSPI flash. If the address range to erase spans multiple sectors, then all of these sectors will be erased by issuing multiple erase commands.

The driver handles sending the write enable command, as well as waiting for the write to complete.

This function may return before the write operation is complete, as the actual erase operation is queued and executed by a thread created by the driver.

Note: The smallest amount of data that can be erased is a 4k sector. This means that data outside the address range specified by `address` and `len` will be erased if the address range does not both begin and end at 4k sector boundaries.

Parameters

- `ctx` – A pointer to the QSPI flash driver instance to use.
- `address` – The byte address to begin erasing. This does not need to begin at a sector boundary, but if it does not, note that the entire sector that contains this address will still be erased.
- `len` – The minimum number of bytes to erase. If `address` + `len` - 1 does not correspond to the last address within a sector, note that the entire sector that contains this address will still be erased.

inline size_t `rtos_qspi_flash_size_get`(*rtos_qspi_flash_t* *qspi_flash_ctx)

This gets the size in bytes of the flash chip.

Parameters

- `A` – pointer to the QSPI flash driver instance to query.

Returns

the size in bytes of the flash chip.

inline size_t `rtos_qspi_flash_page_size_get`(*rtos_qspi_flash_t* *qspi_flash_ctx)

This gets the size in bytes of each page in the flash chip.

Parameters

- `A` – pointer to the QSPI flash driver instance to query.

Returns

the size in bytes of the flash page.

inline size_t `rtos_qspi_flash_page_count_get`(*rtos_qspi_flash_t* *qspi_flash_ctx)

This gets the number of pages in the flash chip.

Parameters

- `A` – pointer to the QSPI flash driver instance to query.

Returns

the number of pages in the flash chip.

inline size_t `rtos_qspi_flash_sector_size_get`(*rtos_qspi_flash_t* *qspi_flash_ctx)

This gets the sector size of the flash chip

Parameters

- `A` – pointer to the QSPI flash driver instance to query.

Returns

the size in bytes of the smallest sector

inline unsigned `rtos_qspi_flash_calibration_valid_get`(*rtos_qspi_flash_t* *qspi_flash_ctx)

Gets the value of the calibration valid.

Parameters

- `A` – pointer to the QSPI flash driver instance to query.

Returns

1 if calibration was successful 0 otherwise

RPC Initialization API The following functions may be used to share a QSPI flash driver instance with other xcore tiles. Tiles that the driver instance is shared with may call any of the core functions listed above.

void `rtos_qspi_flash_rpc_client_init`(*rtos_qspi_flash_t* *qspi_flash_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *host_intertile_ctx)

Initializes an RTOS QSPI flash driver instance on a client tile. This allows a tile that does not own the actual driver instance to use a driver instance on another tile. This will be called instead of *rtos_qspi_flash_init()*. The host tile that owns the actual instance must simultaneously call *rtos_qspi_flash_rpc_host_init()*.

Parameters

- `qspi_flash_ctx` – A pointer to the QSPI flash driver instance to initialize.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `qspi_flash_ctx`.
- `host_intertile_ctx` – A pointer to the intertile driver instance to use for performing the communication between the client and host tiles. This must have the same scope as `qspi_flash_ctx`.

void `rtos_qspi_flash_rpc_host_init`(*rtos_qspi_flash_t* *qspi_flash_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *client_intertile_ctx[], size_t remote_client_count)

Performs additional initialization on a QSPI flash driver instance to allow client tiles to use the QSPI flash driver instance. Each client tile that will use this instance must simultaneously call *rtos_qspi_flash_rpc_client_init()*.

Parameters

- `qspi_flash_ctx` – A pointer to the QSPI flash driver instance to share with clients.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `qspi_flash_ctx`.

- `client_intertile_ctx` – An array of pointers to the intertile driver instances to use for performing the communication between the host tile and each client tile. This must have the same scope as `qspi_flash_ctx`.
- `remote_client_count` – The number of client tiles to share this driver instance with.

void `rtos_qspi_flash_rpc_config`(*rtos_qspi_flash_t* *qspi_flash_ctx, unsigned intertile_port, unsigned host_task_priority)

Configures the RPC for a QSPI flash driver instance. This must be called by both the host tile and all client tiles.

On the client tiles this must be called after calling *rtos_qspi_flash_rpc_client_init()*. After calling this, the client tile may immediately begin to call the core QSPI flash functions on this driver instance. It does not need to wait for the host to call *rtos_qspi_flash_start()*.

On the host tile this must be called both after calling *rtos_qspi_flash_rpc_host_init()* and before calling *rtos_qspi_flash_start()*.

Parameters

- `qspi_flash_ctx` – A pointer to the QSPI flash driver instance to configure the RPC for.
- `intertile_port` – The port number on the intertile channel to use for transferring the RPC requests and responses for this driver instance. This port must not be shared by any other functions. The port must be the same for the host and all its clients.
- `host_task_priority` – The priority to use for the task on the host tile that handles RPC requests from the clients.

SPI RTOS Driver

This driver can be used to instantiate and control a SPI master or slave mode I/O interface on xcore in an RTOS application.

SPI Master RTOS Driver This driver can be used to instantiate and control a SPI master I/O interface on xcore in an RTOS application.

SPI Master Initialization API The following structures and functions are used to initialize and start a SPI master driver instance.

typedef struct *rtos_spi_master_struct* `rtos_spi_master_t`

Typedef to the RTOS SPI master driver instance struct.

typedef struct *rtos_spi_master_device_struct* `rtos_spi_master_device_t`

Typedef to the RTOS SPI device instance struct.

void `rtos_spi_master_start`(*rtos_spi_master_t* *spi_master_ctx, unsigned priority)

Starts an RTOS SPI master driver instance. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before any of the core SPI master driver functions are called with this instance.

rtos_spi_master_init() must be called on this SPI master driver instance prior to calling this.

Parameters

- `spi_master_ctx` – A pointer to the SPI master driver instance to start.
- `priority` – The priority of the task that gets created by the driver to handle the SPI master interface.

void `rtos_spi_master_init`(*rtos_spi_master_t* *bus_ctx, xclock_t clock_block, port_t cs_port, port_t sclk_port, port_t mosi_port, port_t miso_port)

Initializes an RTOS SPI master driver instance. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before calling *rtos_spi_master_start()* or any of the core SPI master driver functions with this instance.

Parameters

- `bus_ctx` – A pointer to the SPI master driver instance to initialize.
- `clock_block` – The clock block to use for the SPI master interface.
- `cs_port` – The SPI interface's chip select port. This may be a multi-bit port.
- `sclk_port` – The SPI interface's SCLK port. Must be a 1-bit port.
- `mosi_port` – The SPI interface's MOSI port. Must be a 1-bit port.
- `miso_port` – The SPI interface's MISO port. Must be a 1-bit port.

void `rtos_spi_master_device_init`(*rtos_spi_master_device_t* *dev_ctx, *rtos_spi_master_t* *bus_ctx, uint32_t cs_pin, int cpol, int cpha, spi_master_source_clock_t source_clock, uint32_t clock_divisor, spi_master_sample_delay_t miso_sample_delay, uint32_t miso_pad_delay, uint32_t cs_to_clk_delay_ticks, uint32_t clk_to_cs_delay_ticks, uint32_t cs_to_cs_delay_ticks)

Initialize a SPI device. Multiple SPI devices may be initialized per RTOS SPI master driver instance. Each must be on a unique pin of the interface's chip select port. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before calling *rtos_spi_master_start()* or any of the core SPI master driver functions with this instance.

Parameters

- `dev_ctx` – A pointer to the SPI device instance to initialize.
- `bus_ctx` – A pointer to the SPI master driver instance to attach the device to.
- `cs_pin` – The bit number of the chip select port that is connected to the device's chip select pin.
- `cpol` – The clock polarity required by the device.
- `cpha` – The clock phase required by the device.
- `source_clock` – The source clock to derive SCLK from. See spi_master_source_clock_t.
- `clock_divisor` – The value to divide the source clock by. The frequency of SCLK will be set to:
  - (F_src) / (4 * clock_divisor) when clock_divisor > 0
  - (F_src) / (2) when clock_divisor = 0 Where F_src is the frequency of the source clock.
- `miso_sample_delay` – When to sample MISO. See spi_master_sample_delay_t.
- `miso_pad_delay` – The number of core clock cycles to delay sampling the MISO pad during a transaction. This allows for more fine grained adjustment of sampling time. The value may be between 0 and 5.
- `cs_to_clk_delay_ticks` – The minimum number of reference clock ticks between assertion of chip select and the first clock edge.

- `clk_to_cs_delay_ticks` – The minimum number of reference clock ticks between the last clock edge and de-assertion of chip select.
- `cs_to_cs_delay_ticks` – The minimum number of reference clock ticks between transactions, which is between de-assertion of chip select and the end of one transaction, and its re-assertion at the beginning of the next.

struct `rtos_spi_master_struct`

#include <rtos_spi_master.h> Struct representing an RTOS SPI master driver instance.

The members in this struct should not be accessed directly.

struct `rtos_spi_master_device_struct`

#include <rtos_spi_master.h> Struct representing an RTOS SPI device instance.

The members in this struct should not be accessed directly.

SPI Master Core API The following functions are the core SPI master driver functions that are used after it has been initialized and started.

inline void `rtos_spi_master_transaction_start`(*rtos_spi_master_device_t* *ctx)

Starts a transaction with the specified SPI device on a SPI bus. This leaves chip select asserted.

Note: When this is called, the servicer thread will be locked to the core that it executed on until *rtos_spi_master_transaction_end()* is called. This is because the underlying I/O software utilized fast mode and high priority.

Parameters

- `ctx` – A pointer to the SPI device instance.

inline void `rtos_spi_master_transfer`(*rtos_spi_master_device_t* *ctx, uint8_t *data_out, uint8_t *data_in, size_t len)

Transfers data to and from the specified SPI device on a SPI bus. The transaction must already have been started by calling *rtos_spi_master_transaction_start()* on the same device instance. This may be called multiple times during a single transaction.

This function may return before the transfer is complete when data_in is NULL, as the actual transfer operation is queued and executed by a thread created by the driver.

Parameters

- `ctx` – A pointer to the SPI device instance.
- `data_out` – Pointer to the data to transfer to the device. This may be NULL if there is no data to send.
- `data_in` – Pointer to the buffer to save the received data to. This may be NULL if the received data is not needed.
- `len` – The number of bytes to transfer in each direction. This number of bytes must be available in both the `data_out` and `data_in` buffers if they are not NULL.

inline void `rtos_spi_master_delay_before_next_transfer`(*rtos_spi_master_device_t* *ctx, uint32_t delay_ticks)

If there is a minimum amount of idle time that is required by the device between transfers within a single transaction, then this may be called between each transfer where a delay is required.

This function will return immediately. If the call for the next transfer happens before the minimum time specified has elapsed, the delay will occur then before the transfer begins.

Note: This must be called during a transaction, otherwise the behavior is unspecified.

Note: Technically the next transfer will occur no earlier than `delay_ticks` after this function is called, so this should be called immediately following a transfer, rather than immediately before the next.

Parameters

- `ctx` – A pointer to the SPI device instance.
- `delay_ticks` – The number of reference clock ticks to delay.

inline void `rtos_spi_master_transaction_end`(*rtos_spi_master_device_t* *ctx)

Ends a transaction with the specified SPI device on a SPI bus. This leaves chip select de-asserted.

Parameters

- `ctx` – A pointer to the SPI device instance.

SPI Master RPC Initialization API The following functions may be used to share a SPI master driver instance with other xcore tiles. Tiles that the driver instance is shared with may call any of the core functions listed above.

void `rtos_spi_master_rpc_client_init`(*rtos_spi_master_t* *spi_master_ctx, *rtos_spi_master_device_t* *spi_device_ctx[], size_t spi_device_count, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *host_intertile_ctx)

Initializes an RTOS SPI master driver instance on a client tile, as well as any number of SPI device instances. This allows a tile that does not own the actual driver instance to use a driver instance on another tile. This will be called instead of *rtos_spi_master_init()* and *rtos_spi_master_device_init()*. The host tile that owns the actual instances must simultaneously call *rtos_spi_master_rpc_host_init()*.

Parameters

- `spi_master_ctx` – A pointer to the SPI master driver instance to initialize.
- `spi_device_ctx` – An array of pointers to SPI device instances to initialize.
- `spi_device_count` – The number of SPI device instances to initialize.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `spi_master_ctx`.
- `host_intertile_ctx` – A pointer to the intertile driver instance to use for performing the communication between the client and host tiles. This must have the same scope as `spi_master_ctx`.

void `rtos_spi_master_rpc_host_init`(*rtos_spi_master_t* *spi_master_ctx, *rtos_spi_master_device_t* *spi_device_ctx[], size_t spi_device_count, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *client_intertile_ctx[], size_t remote_client_count)

Performs additional initialization on a SPI master driver instance to allow client tiles to use the SPI master driver instance. Each client tile that will use this instance must simultaneously call *rtos_spi_master_rpc_client_init()*.

Parameters

- `spi_master_ctx` – A pointer to the SPI master driver instance to share with clients.
- `spi_device_ctx` – An array of pointers to SPI device instances to share with clients.
- `spi_device_count` – The number of SPI device instances to share.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `spi_master_ctx`.
- `client_intertile_ctx` – An array of pointers to the intertile driver instances to use for performing the communication between the host tile and each client tile. This must have the same scope as `spi_master_ctx`.
- `remote_client_count` – The number of client tiles to share this driver instance with.

void `rtos_spi_master_rpc_config`(*rtos_spi_master_t* *spi_master_ctx, unsigned intertile_port, unsigned host_task_priority)

Configures the RPC for a SPI master driver instance. This must be called by both the host tile and all client tiles.

On the client tiles this must be called after calling *rtos_spi_master_rpc_client_init()*. After calling this, the client tile may immediately begin to call the core SPI master functions on this driver instance. It does not need to wait for the host to call *rtos_spi_master_start()*.

On the host tile this must be called both after calling *rtos_spi_master_rpc_host_init()* and before calling *rtos_spi_master_start()*.

Parameters

- `spi_master_ctx` – A pointer to the SPI master driver instance to configure the RPC for.
- `intertile_port` – The port number on the intertile channel to use for transferring the RPC requests and responses for this driver instance. This port must not be shared by any other functions. The port must be the same for the host and all its clients.
- `host_task_priority` – The priority to use for the task on the host tile that handles RPC requests from the clients.

SPI Slave RTOS Driver This driver can be used to instantiate and control a SPI slave I/O interface on xcore in an RTOS application.

SPI Slave API The following structures and functions are used to initialize and start a SPI slave driver instance.

typedef struct *rtos_spi_slave_struct* `rtos_spi_slave_t`

Typedef to the RTOS SPI slave driver instance struct.

typedef void (*`rtos_spi_slave_start_cb_t`)(*rtos_spi_slave_t* *ctx, void *app_data)

Function pointer type for application provided RTOS SPI slave start callback functions.

These callback functions are optionally called by a SPI slave driver's thread when it is first started. This gives the application a chance to perform startup initialization from within the driver's thread. It is a good place for the first call to *spi_slave_xfer_prepare()*.

Param ctx
A pointer to the associated SPI slave driver instance.

Param app_data
A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

typedef void (*`rtos_spi_slave_xfer_done_cb_t`)(*rtos_spi_slave_t* *ctx, void *app_data)

Function pointer type for application provided RTOS SPI slave transfer done callback functions.

These callback functions are optionally called when a SPI slave driver instance is done transferring data with a master device.

An application can use this to be notified immediately when a transfer has completed. It can then call *spi_slave_xfer_complete()* with a timeout of 0 from within this callback to get the transfer results.

Param ctx
A pointer to the associated SPI slave driver instance.

Param app_data
A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

typedef struct *xfer_done_queue_item* `xfer_done_queue_item_t`

Internally used struct representing an received data packet.

The members in this struct should not be accessed directly.

void `spi_slave_xfer_prepare`(*rtos_spi_slave_t* *ctx, void *rx_buf, size_t rx_buf_len, void *tx_buf, size_t tx_buf_len)

Prepares an RTOS SPI slave driver instance with buffers for subsequent transfers. Before this is called for the first time, any transfers initiated by a master device with result in all received data over MOSI being dropped, and all data sent over MISO being zeros.

This only needs to be called when the buffers need to be changed. If all transfers will use the same buffers, then this only needs to be called once during initialization.

If the application has not processed the previous transaction, the buffers will be held, and default buffers set by *spi_slave_xfer_prepare_default_buffers()* will be used if a new transaction starts.

Parameters

- `ctx` – A pointer to the SPI slave driver instance to use.
- `rx_buf` – The buffer to receive data into for any subsequent transfers.
- `rx_buf_` – The length in bytes of `rx_buf`. If the master transfers more than this during a single transfer, then the bytes that do not fit within `rx_buf` will be lost.
- `tx_buf` – The buffer to send data from for any subsequent transfers.
- `tx_buf_len` – The length in bytes of `tx_buf`. If the master transfers more than this during a single transfer, zeros will be sent following the last byte `tx_buf`.

void `spi_slave_xfer_prepare_default_buffers`(*rtos_spi_slave_t* *ctx, void *rx_buf, size_t rx_buf_len, void *tx_buf, size_t tx_buf_len)

Prepares an RTOS SPI slave driver instance with default buffers for subsequent transfers. Before this is called for the first time, any transfers initiated by a master device with result in all received data over MOSI being dropped, and all data sent over MISO being zeros.

This only needs to be called when the buffers need to be changed.

The default buffer will be used in the event that the application has not yet processed the previous transfer. This enables the application to have a default buffer to implement a sort of NACK over SPI in the event that the device was busy and had not yet finished handling the previous transaction before a new one started.

Parameters

- `ctx` – A pointer to the SPI slave driver instance to use.
- `rx_buf` – The buffer to receive data into for any subsequent transfers.
- `rx_buf_` – The length in bytes of `rx_buf`. If the master transfers more than this during a single transfer, then the bytes that do not fit within `rx_buf` will be lost.
- `tx_buf` – The buffer to send data from for any subsequent transfers.
- `tx_buf_len` – The length in bytes of `tx_buf`. If the master transfers more than this during a single transfer, zeros will be sent following the last byte `tx_buf`.

int `spi_slave_xfer_complete`(*rtos_spi_slave_t* *ctx, void **rx_buf, size_t *rx_len, void **tx_buf, size_t *tx_len, unsigned timeout)

Waits for a SPI transfer to complete. Returns either when the timeout is reached, or when a transfer completes, whichever comes first. If a transfer does complete, then the buffers and the number of bytes read from or written to them are returned via the parameters.

Note: The duration of this callback will effect the minimum duration between SPI transactions

---

Parameters

- `ctx` – A pointer to the SPI slave driver instance to use.
- `rx_buf` – The receive buffer used for the completed transfer. This is set by the function upon completion of a transfer.
- `rx_len` – The number of bytes written to rx_buf. This is set by the function upon completion of a transfer.
- `tx_buf` – The transmit buffer used for the completed transfer. This is set by the function upon completion of a transfer.
- `tx_len` – The number of bytes sent from tx_buf. This is set by the function upon completion of a transfer.
- `timeout` – The number of RTOS ticks to wait before the next transfer is complete. When called from within the "xfer_done" callback, this should be 0.

Return values

- `0` – if a transfer completed. All buffers and lengths are set in this case.
- `-1` – if no transfer completed before the timeout expired. No buffers or lengths are returned in this case.

void `spi_slave_default_buf_xfer_ended_enable`(*rtos_spi_slave_t* *ctx)

Sets the driver to use callbacks for all default transactions. This will result in transfers done with the default buffer generating callbacks to the application to xfer_done. This will require default buffer transaction items to be processed with *spi_slave_xfer_complete()*

Note: This is the default setting

Parameters

- `ctx` – A pointer to the SPI slave driver instance to use.

void `spi_slave_default_buf_xfer_ended_disable`(*rtos_spi_slave_t* *ctx)

Sets the driver to drop all default transactions. This will result in transfers done with the default buffer not generating callbacks to the application to xfer_done. This will also stop default buffer transaction items from being required to be processed with *spi_slave_xfer_complete()*

Parameters

- `ctx` – A pointer to the SPI slave driver instance to use.

void `rtos_spi_slave_start`(*rtos_spi_slave_t* *spi_slave_ctx, void *app_data, *rtos_spi_slave_start_cb_t* start, *rtos_spi_slave_xfer_done_cb_t* xfer_done, unsigned interrupt_core_id, unsigned priority)

Starts an RTOS SPI slave driver instance. This must only be called by the tile that owns the driver instance. It must be called after starting the RTOS from an RTOS thread.

rtos_spi_slave_init() must be called on this SPI slave driver instance prior to calling this.

Parameters

- `spi_slave_ctx` – A pointer to the SPI slave driver instance to start.
- `app_data` – A pointer to application specific data to pass to the callback functions.
- `start` – The callback function that is called when the driver's thread starts. This is optional and may be NULL.
- `xfer_done` – The callback function that is notified when transfers are complete. This is optional and may be NULL.
- `interrupt_core_id` – The ID of the core on which to enable the SPI interrupt. This core should not be shared with threads that disable interrupts for long periods of time, nor enable other interrupts.
- `priority` – The priority of the task that gets created by the driver to call the callback functions. If both callback functions are NULL, then this is unused.

void `rtos_spi_slave_init`(*rtos_spi_slave_t* *spi_slave_ctx, uint32_t io_core_mask, xclock_t clock_block, int cpol, int cpha, port_t p_sclk, port_t p_mosi, port_t p_miso, port_t p_cs)

Initializes an RTOS SPI slave driver instance. This must only be called by the tile that owns the driver instance. It should be called before starting the RTOS, and must be called before calling *rtos_spi_slave_start()*.

For timing parameters and maximum clock rate, refer to the underlying HIL IO API.

Parameters

- `spi_slave_ctx` – A pointer to the SPI slave driver instance to initialize.
- `io_core_mask` – A bitmask representing the cores on which the low level SPI I/O thread created by the driver is allowed to run. Bit 0 is core 0, bit 1 is core 1, etc.
- `clock_block` – The clock block to use for the SPI slave.
- `cpol` – The clock polarity to use.
- `cpha` – The clock phase to use.
- `p_sclk` – The SPI slave's SCLK port. Must be a 1-bit port.
- `p_mosi` – The SPI slave's MOSI port. Must be a 1-bit port.
- `p_miso` – The SPI slave's MISO port. Must be a 1-bit port.

- `p_cs` – The SPI slave's CS port. Must be a 1-bit port.

`RTOS_SPI_SLAVE_CALLBACK_ATTR`
: This attribute must be specified on all RTOS SPI slave callback functions provided by the application.

`HIL_IO_SPI_SLAVE_HIGH_PRIO`
: Set SPI Slave thread to high priority

`HIL_IO_SPI_SLAVE_FAST_MODE`
: Set SPI Slave thread to run in fast mode

struct `xfer_done_queue_item`
: *#include <rtos_spi_slave.h>* Internally used struct representing an received data packet.

  The members in this struct should not be accessed directly.

struct `rtos_spi_slave_struct`
: *#include <rtos_spi_slave.h>* Struct representing an RTOS SPI slave driver instance.

  The members in this struct should not be accessed directly.

UART RTOS Driver

This driver can be used to instantiate and control an UART Rx or UART Tx I/O interface on xCORE in an RTOS application.

UART Tx RTOS Driver This driver can be used to instantiate and control an UART Tx I/O interface on xCORE in an RTOS application.

UART Tx API The following structures and functions are used to initialize and start a UART Tx driver instance.

typedef struct *rtos_uart_tx_struct* `rtos_uart_tx_t`
: Typedef to the RTOS UART tx driver instance struct.

inline void `rtos_uart_tx_write`(*rtos_uart_tx_t* *ctx, const uint8_t buf[], size_t n)
: Writes data to an initialized and started UART instance. Unlike the UART rx, an xcore logical core is not reserved. The UART transmission is a function call and the the function blocks until the stop bit of the last byte to be transmittted has completed. Interrupts are masked during this time to avoid stretching of the waveform. Consequently, the tx consumes cycles from the caller thread.

  Parameters

  - `ctx` – A pointer to the UART Tx driver instance to use.
  - `buf` – The buffer containing data to write.
  - `n` – The number of bytes to write.

void `rtos_uart_tx_init`(*rtos_uart_tx_t* *ctx, const port_t tx_port, const uint32_t baud_rate, const uint8_t num_data_bits, const uart_parity_t parity, const uint8_t stop_bits, hwtimer_t tmr)

Initialises an RTOS UART tx driver instance. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before calling *rtos_uart_tx_start()* or any of the core UART tx driver functions with this instance.

Parameters

- `ctx` – A pointer to the UART tx driver instance to initialise.
- `tx_port` – The port containing the transmit pin
- `baud_rate` – The baud rate of the UART in bits per second.
- `num_data_bits` – The number of data bits per frame sent.
- `parity` – The type of parity used. See uart_parity_t above.
- `stop_bits` – The number of stop bits asserted at the of the frame.
- `tmr` – The resource id of the timer to be used by the UART tx.

void `rtos_uart_tx_start`(*rtos_uart_tx_t* *ctx)

Starts an RTOS UART tx driver instance. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before any of the core UART tx driver functions are called with this instance.

rtos_uart_tx_init() must be called on this UART tx driver instance prior to calling this.

Parameters

- `ctx` – A pointer to the UART tx driver instance to start.

struct `rtos_uart_tx_struct`

#include <rtos_uart_tx.h> Struct representing an RTOS UART tx driver instance.

The members in this struct should not be accessed directly.

UART Tx RPC Initialization API The following functions may be used to share a UART Tx driver instance with other xCORE tiles. Tiles that the driver instance is shared with may call any of the core functions listed above.

void `rtos_uart_tx_rpc_client_init`(*rtos_uart_tx_t* *uart_tx_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *host_intertile_ctx)

Initializes an RTOS UART tx driver instance on a client tile. This allows a tile that does not own the actual driver instance to use a driver instance on another tile. This will be called instead of *rtos_uart_tx_init()*. The host tile that owns the actual instance must simultaneously call *rtos_uart_tx_rpc_host_init()*.

Parameters

- `uart_tx_ctx` – A pointer to the UART tx driver instance to initialize.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `uart_tx_ctx`.
- `host_intertile_ctx` – A pointer to the intertile driver instance to use for performing the communication between the client and host tiles. This must have the same scope as `uart_tx_ctx`.

void `rtos_uart_tx_rpc_host_init`(*rtos_uart_tx_t* *uart_tx_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *client_intertile_ctx[], size_t remote_client_count)

Performs additional initialization on an UART tx driver instance to allow client tiles to use the UART tx driver instance. Each client tile that will use this instance must simultaneously call *rtos_uart_tx_rpc_client_init()*.

Parameters

- `uart_tx_ctx` – A pointer to the UART tx driver instance to share with clients.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `uart_tx_ctx`.
- `client_intertile_ctx` – An array of pointers to the intertile driver instances to use for performing the communication between the host tile and each client tile. This must have the same scope as `uart_tx_ctx`.
- `remote_client_count` – The number of client tiles to share this driver instance with.

void `rtos_uart_tx_rpc_config`(*rtos_uart_tx_t* *uart_tx_ctx, unsigned intertile_port, unsigned host_task_priority)

Configures the RPC for an UART tx driver instance. This must be called by both the host tile and all client tiles.

On the client tiles this must be called after calling *rtos_uart_tx_rpc_client_init()*. After calling this, the client tile may immediately begin to call the core UART tx functions on this driver instance. It does not need to wait for the host to call *rtos_uart_tx_start()*.

On the host tile this must be called both after calling *rtos_uart_tx_rpc_host_init()* and before calling *rtos_uart_tx_start()*.

Parameters

- `uart_tx_ctx` – A pointer to the UART tx driver instance to configure the RPC for.
- `intertile_port` – The port number on the intertile channel to use for transferring the RPC requests and responses for this driver instance. This port must not be shared by any other functions. The port must be the same for the host and all its clients.
- `host_task_priority` – The priority to use for the task on the host tile that handles RPC requests from the clients.

UART Rx RTOS Driver This driver can be used to instantiate and control an UART Rx I/O interface on xcore in an RTOS application.

UART Rx API The following structures and functions are used to initialize and start a UART Rx driver instance.

typedef struct *rtos_uart_rx_struct* `rtos_uart_rx_t`

Typedef to the RTOS UART rx driver instance struct.

typedef void (*`rtos_uart_rx_started_cb_t`)(*rtos_uart_rx_t* *ctx)

Function pointer type for application provided RTOS UART rx start callback functions.

This callback function is optionally (may be NULL) called by an UART rx driver's thread when it is first started. This gives the application a chance to perform startup initialization from within the driver's thread.

Param ctx
A pointer to the associated UART rx driver instance.

typedef void (*`rtos_uart_rx_complete_cb_t`)(*rtos_uart_rx_t* *ctx)

Function pointer type for application provided RTOS UART rx receive callback function.

This callback functions are called when an UART rx driver instance has received data to a specified depth. Please use the xStreamBufferReceive(rtos_uart_rx_ctx->isr_byte_buffer, ... to read the bytes.

Param ctx
A pointer to the associated UART rx driver instance.

typedef void (*`rtos_uart_rx_error_t`)(*rtos_uart_rx_t* *ctx, uint8_t err_flags)

Function pointer type for application provided RTOS UART rx error callback functions.

This callback function is optionally (may be NULL_ called when an UART rx driver instance experiences an error in reception. These error types are defined in uart.h of the underlying HIL driver but can be of the following types for the RTOS rx: UART_START_BIT_ERROR, UART_PARITY_ERROR, UART_FRAMING_ERROR, UART_OVERRUN_ERROR.

Param ctx
A pointer to the associated UART rx driver instance.

Param err_flags
An 8b word containing error flags set during reception of last frame. See rtos_uart_rx.h for the bit field definitions.

size_t `rtos_uart_rx_read`(*rtos_uart_rx_t* *uart_rx_ctx, uint8_t *buf, size_t n, rtos_osal_tick_t timeout)

Reads data from a UART Rx instance. It will read up to n bytes or timeout, whichever comes first.

Parameters

- `uart_rx_ctx` – A pointer to the UART Rx driver instance to use.
- `buf` – The buffer to be written with the read UART bytes.
- `n` – The number of bytes to write.
- `timeout` – How long in ticks before the read operation should timeout.

Returns
The number of bytes read.

void `rtos_uart_rx_reset_buffer`(*rtos_uart_rx_t* *uart_rx_ctx)

Resets the receive buffer. Clears the contents and sets number of items rto zero.

Parameters

- `uart_rx_ctx` – A pointer to the UART Rx driver instance to use.

void `rtos_uart_rx_init`(*rtos_uart_rx_t* *uart_rx_ctx, uint32_t io_core_mask, port_t rx_port, uint32_t baud_rate, uint8_t data_bits, uart_parity_t parity, uint8_t stop_bits, hwtimer_t tmr)

Initializes an RTOS UART rx driver instance. This must only be called by the tile that owns the driver instance. It should be called before starting the RTOS, and must be called before calling *rtos_uart_rx_start()*. Note that UART rx requires a whole logical core for the underlying HIL UART Rx instance.

Parameters

- `uart_rx_ctx` – A pointer to the UART rx driver instance to initialize.
- `io_core_mask` – A bitmask representing the cores on which the low UART Rx thread created by the driver is allowed to run. Bit 0 is core 0, bit 1 is core 1, etc.
- `rx_port` – The port containing the receive pin

- `baud_rate` – The baud rate of the UART in bits per second.
- `data_bits` – The number of data bits per frame sent.
- `parity` – The type of parity used. See uart_parity_t above.
- `stop_bits` – The number of stop bits asserted at the of the frame.
- `tmr` – The resource id of the timer to be used by the UART Rx.

void `rtos_uart_rx_start`(*rtos_uart_rx_t* *uart_rx_ctx, void *app_data, *rtos_uart_rx_started_cb_t* start, *rtos_uart_rx_complete_cb_t* rx_complete, *rtos_uart_rx_error_t* error, unsigned interrupt_core_id, unsigned priority, size_t app_rx_buff_size)

Starts an RTOS UART rx driver instance. This must only be called by the tile that owns the driver instance. It must be called after starting the RTOS and from an RTOS thread.

rtos_uart_rx_init() must be called on this UART rx driver instance prior to calling this.

Parameters

- `uart_rx_ctx` – A pointer to the UART rx driver instance to start.
- `app_data` – A pointer to application specific data to pass to the callback functions available in *rtos_uart_rx_struct*.
- `start` – The callback function that is called when the driver's thread starts. This is optional and may be NULL.
- `rx_complete` – The callback function to indicate data received by the UART.
- `error` – The callback function called when a reception error has occured.
- `interrupt_core_id` – The ID of the core on which to enable the UART rx interrupt.
- `priority` – The priority of the task that gets created by the driver to call the callback functions.
- `app_rx_buff_size` – The size in bytes of the RTOS xstreambuffer used to buffer received words for the application.

`UR_COMPLETE_CB_CODE`

The callback code bit positions available for RTOS UART Rx.

`UR_STARTED_CB_CODE`

`UR_START_BIT_ERR_CB_CODE`

`UR_PARITY_ERR_CB_CODE`

`UR_FRAMING_ERR_CB_CODE`

`UR_OVERRUN_ERR_CB_CODE`

`UR_COMPLETE_CB_FLAG`

The callback code flag masks available for RTOS UART Rx.

`UR_STARTED_CB_FLAG`

`UR_START_BIT_ERR_CB_FLAG`

`UR_PARITY_ERR_CB_FLAG`

`UR_FRAMING_ERR_CB_FLAG`

`UR_OVERRUN_ERR_CB_FLAG`

`RX_ERROR_FLAGS`

`RX_ALL_FLAGS`

`RTOS_UART_RX_BUF_LEN`

The size of the byte buffer between the ISR and the appthread. It needs to be able to hold sufficient bytes received until the app_thread is able to service it. This is not the same as app_byte_buffer_size which can be of any size, specified by the user at device start. At 1Mbps we get a byte every 10us so 64B allows 640us for the app thread to respond. Note buffer is size n+1 as required by lib_uart.

`RTOS_UART_RX_CALLBACK_ATTR`

This attribute must be specified on all RTOS UART rx callback functions provided by the application to allow compiler stack calculation.

`RTOS_UART_RX_CALL_ATTR`

This attribute must be specified on all RTOS UART functions provided by the application to allow compiler stack calculation.

struct `rtos_uart_rx_struct`

#include <rtos_uart_rx.h> Struct representing an RTOS UART rx driver instance.

The members in this struct should not be accessed directly.

USB RTOS Driver

This driver can be used to instantiate and control a USB device interface on xcore in an RTOS application.

Unlike most other xcore I/O interface RTOS drivers, only a single USB driver instance may be started. It also does not require an initialization step prior to starting the driver. This is due to an implementation detail in lib_xud, which is what the RTOS USB driver uses at its core.

Driver API The following structures and functions are used to start and use a USB driver instance.

`RTOS_USB_OUT_EP`

This is used to index into the second dimension of many of the RTOS USB driver's endpoint arrays.

`RTOS_USB_IN_EP`

enum `rtos_usb_packet_type_t`

Values:

enumerator `rtos_usb_data_packet`

enumerator `rtos_usb_setup_packet`

enumerator `rtos_usb_sof_packet`

typedef struct *rtos_usb_struct* `rtos_usb_t`

Typedef to the RTOS USB driver instance struct.

typedef void (*`rtos_usb_isr_cb_t`)(*rtos_usb_t* *ctx, void *app_data, uint32_t ep_address, size_t xfer_len, *rtos_usb_packet_type_t* packet_type, XUD_Result_t res)

Function pointer type for application provided RTOS USB interrupt callback function.

This callback function is called when there is a USB transfer interrupt.

Param ctx
A pointer to the associated USB driver instance.

Param app_data
A pointer to application specific data provided by the application. Used to share data between this callback function and the application.

Param ep_address
The address of the USB endpoint that the transfer has completed on.

Param xfer_len
The length of the data transferred.

Param packet_type
The type of packet transferred. See rtos_usb_packet_type_t.

Param res
The result of the transfer. See XUD_Result_t.

int `rtos_usb_endpoint_ready`(*rtos_usb_t* *ctx, uint32_t endpoint_addr, unsigned timeout)

Checks to see if a particular endpoint is ready to use.

Parameters

- `ctx` – A pointer to the USB driver instance to use.
- `endpoint_addr` – The address of the endpoint to check.
- `timeout` – The maximum amount of time to wait for the endpoint to become ready before returning.

Return values

- `XUD_RES_OKAY` – if the endpoint is ready to use.
- `XUD_RES_ERR` – if the endpoint is not ready to use.

XUD_Result_t `rtos_usb_all_endpoints_ready`(*rtos_usb_t* *ctx, unsigned timeout)

Checks to see if all endpoints are ready to use.

Parameters

- `ctx` – A pointer to the USB driver instance to use.
- `timeout` – The maximum amount of time to wait for all endpoints to become ready before returning.

Return values

- `XUD_RES_OKAY` – if all the endpoints are ready to use.
- `XUD_RES_ERR` – if not all the endpoints are ready to use.

XUD_Result_t `rtos_usb_endpoint_transfer_start`(*rtos_usb_t* *ctx, uint32_t endpoint_addr, uint8_t *buffer, size_t len, bool is_setup)

Requests a transfer on a USB endpoint. This function returns immediately. When the transfer is complete, the application's ISR callback provided to *rtos_usb_start()* will be called.

Parameters

- `ctx` – A pointer to the USB driver instance to use.
- `endpoint_addr` – The address of the endpoint to perform the transfer on.
- `buffer` – A pointer to the buffer to transfer data into for OUT endpoints, or from for IN endpoints. For OUT endpoint, the buffer needs an additional +4 bytes of space, this additional data should not be reflected in the `len` parameter.
- `len` – The maximum number of bytes to receive for OUT endpoints, or the actual number of bytes to send for IN endpoints.
- `is_setup` – To be set when preparing for the transfer of a setup packet.

Return values

- `XUD_RES_OKAY` – if the transfer was requested successfully.
- `XUD_RES_RST` – if the transfer was not requested and the USB bus needs to be reset. In this case, the application should reset the USB bus.

XUD_BusSpeed_t `rtos_usb_endpoint_reset`(*rtos_usb_t* *ctx, uint32_t endpoint_addr)

This function will complete a reset on an endpoint. The address of the endpoint to reset must be provided, and may be either direction (IN or OUT) endpoint. If there is an associated endpoint of the opposite direction, however, it will also be reset.

The return value should be inspected to find the new bus-speed.

Parameters

- `endpoint_addr` – IN or OUT endpoint address to reset.

Return values

- `XUD_SPEED_HS` – the host has accepted that this device can execute at high speed.
- `XUD_SPEED_FS` – the device is running at full speed.

static inline XUD_Result_t `rtos_usb_device_address_set`(*rtos_usb_t* *ctx, uint32_t addr)

Sets the USB device's bus address. This function must be called after a `setDeviceAddress` request is made by the host, and after the ZLP status is sent.

Parameters

- `ctx` – A pointer to the USB driver instance to use.
- `addr` – The device address requested by the host.

static inline void `rtos_usb_endpoint_state_reset`(*rtos_usb_t* *ctx, uint32_t endpoint_addr)

Reset a USB endpoint's state including data PID toggle.

Parameters

- `ctx` – A pointer to the USB driver instance to use.
- `endpoint_addr` – The address of the endpoint to reset.

static inline void `rtos_usb_endpoint_stall_set`(*rtos_usb_t* *ctx, uint32_t endpoint_addr)

Stalls a USB endpoint. The stall is cleared automatically when a setup packet is received on the endpoint. Otherwise it can be cleared manually with rtos_usb_endpoint_stall_clear().

Parameters

- `ctx` – A pointer to the USB driver instance to use.
- `endpoint_addr` – The address of the endpoint to stall.

static inline void `rtos_usb_endpoint_stall_clear`(*rtos_usb_t* *ctx, uint32_t endpoint_addr)

Clears the stall condition on USB endpoint.

Parameters

- `ctx` – A pointer to the USB driver instance to use.
- `endpoint_addr` – The address of the endpoint to clear the stall on.

void `rtos_usb_start`(*rtos_usb_t* *ctx, size_t endpoint_count, XUD_EpType endpoint_out_type[], XUD_EpType endpoint_in_type[], XUD_BusSpeed_t speed, XUD_PwrConfig power_source, unsigned interrupt_core_id, int sof_interrupt_core_id)

Starts the USB driver instance's low level USB I/O thread and enables its interrupts on the requested core. This must only be called by the tile that owns the driver instance. It must be called after starting the RTOS from an RTOS thread.

rtos_usb_init() must be called on this USB driver instance prior to calling this.

Parameters

- `ctx` – A pointer to the USB driver instance to start.
- `endpoint_count` – The number of endpoints that will be used by the application. A single endpoint here includes both its IN and OUT endpoints. For example, if the application uses EP0_IN, EP0_OUT, EP1_IN, EP2_IN, EP2_OUT, EP3_OUT, then the endpoint count specified here should be 4 (endpoint 0 through endpoint 3) regardless of the lack of EP1_OUT and EP3_IN. If these two endpoints were used, the count would still be 4.

  If for whatever reason, the application needs to use a particular endpoint number, say only EP6 in addition to EP0, then the count here needs to be 7, even though endpoints 1 through 5 are unused. All unused endpoints must be marked as disabled in the two endpoint type lists `endpoint_out_type` and `endpoint_in_type`.
- `endpoint_out_type` – A list of the endpoint types for each output endpoint. Index 0 represents the type for EP0_OUT, and so on. See XUD_EpType in lib_xud. If the endpoint is unused, it must be set to XUD_EPTYPE_DIS.
- `endpoint_in_type` – A list of the endpoint types for each input endpoint. Index 0 represents the type for EP0_IN, and so on. See XUD_EpType in lib_xud. If the endpoint is unused, it must be set to XUD_EPTYPE_DIS.

- `speed` – The speed at which the bus should operate. Either XUD_SPEED_FS or XUD_SPEED_HS. See XUD_BusSpeed_t in lib_xud.
- `power_source` – The source of the device's power. Either bus powered (XUD_PWR_BUS) or self powered (XUD_PWR_SELF). See XUD_PwrConfig in lib_xud.
- `interrupt_core_id` – The ID of the core on which to enable the USB interrupts.
- `sof_interrupt_core_id` – The ID of the core on which to enable the SOF interrupt. Set to < 0 to disable the SoF interrupt if it is not needed.

void `rtos_usb_init`(*rtos_usb_t* *ctx, uint32_t io_core_mask, *rtos_usb_isr_cb_t* isr_cb, void *isr_app_data)

Initializes an RTOS USB driver instance. This must only be called by the tile that owns the driver instance. It should be called prior to starting the RTOS, and must be called before any of the core USB driver functions are called with this instance.

This will create an RTOS thread that runs lib_xud's main loop. This thread is created with the highest priority and with preemption disabled.

Note: Due to implementation details of lib_xud, it is only possible to have one USB instance per application. Functionally this is not an issue, as no xcore chips have more than one USB interface.

Note: If using the Tiny USB stack, then this function should not be called directly by the application. The xcore device port for Tiny USB takes care of calling this, as well as all other USB driver functions.

Parameters

- `ctx` – A pointer to the USB driver instance to start.
- `io_core_mask` – A bitmask representing the cores on which the low level USB I/O thread created by the driver is allowed to run. Bit 0 is core 0, bit 1 is core 1, etc.
- `isr_cb` – The callback function for the driver to call when transfers are completed.
- `isr_app_data` – A pointer to application specific data to pass to the application's ISR callback function `isr_cb`.

XUD_Result_t `rtos_usb_simple_transfer_complete`(*rtos_usb_t* *ctx, uint32_t endpoint_addr, size_t *len, unsigned timeout)

This function may be called to wait for a transfer on a particular endpoint to complete. This requires that the USB instance was initialized with *rtos_usb_simple_init()*.

Parameters

- `ctx` – A pointer to the USB driver instance to use.
- `endpoint_addr` – The address of the endpoint to wait for.
- `len` – The actual number of bytes transferred. For IN endpoints, this will be the same as the length requested by *rtos_usb_endpoint_transfer_start()*. For OUT endpoints, it may be less.
- `timeout` – The maximum amount of time to wait for the transfer to complete before returning.

Return values

- `XUD_RES_OKAY` – if the transfer was completed successfully.

- `XUD_RES_RST` – if the transfer was not able to complete and the USB bus needs to be reset. In this case, the application should reset the USB bus.
- `XUD_RES_ERR` – if there was an unexpected error transferring the data.

void `rtos_usb_simple_init`(*rtos_usb_t* *ctx, uint32_t io_core_mask)

Initializes an RTOS USB driver instance. This must only be called by the tile that owns the driver instance. It should be called prior to starting the RTOS, and must be called before any of the core USB driver functions are called with this instance.

This initialization function may be used instead of *rtos_usb_init()* if the application is not using a USB stack. This allows application threads to wait for transfers to complete with the *rtos_usb_simple_transfer_complete()* function. The application cannot provide its own ISR callback when initialized with this function. This provides a similar programming interface as a traditional bare metal xcore application using lib_xud.

This will create an RTOS thread that runs lib_xud's main loop. This thread is created with the highest priority and with preemption disabled.

Note: Due to implementation details of lib_xud, it is only possible to have one USB instance per application. Functionally this is not an issue, as no xcore chips have more than one USB interface.

Parameters

- `ctx` – A pointer to the USB driver instance to start.
- `io_core_mask` – A bitmask representing the cores on which the low level USB I/O thread created by the driver is allowed to run. Bit 0 is core 0, bit 1 is core 1, etc.

`RTOS_USB_ENDPOINT_COUNT_MAX`

The maximum number of USB endpoint numbers supported by the RTOS USB driver.

`RTOS_USB_ISR_CALLBACK_ATTR`

This attribute must be specified on the RTOS USB interrupt callback function provided by the application.

struct `rtos_usb_ep_xfer_info_t`

#include <rtos_usb.h> Struct to hold USB transfer state data per endpoint, used as the argument to the ISR.

The members in this struct should not be accessed directly.

struct `rtos_usb_struct`

#include <rtos_usb.h> Struct representing an RTOS USB driver instance.

The members in this struct should not be accessed directly.

Trace Driver

This driver can be used to instantiate an xscope-based trace module in an RTOS application. The trace module currently supports both a demonstrative ASCII-mode and Percepio's Tracealzyer on FreeRTOS. Both modes are dependent on RTOS-specific hooks/macros to handle the majority of RTOS event recording and integration.

For general usage of the FreeRTOS trace functionality please refer to FreeRTOS' documentation here: RTOS Trace Macros

For basic information on printf debugging using xscope please refer to the tools guide here: XSCOPE debugging

Trace Configuration In order to use the trace driver module, the following common steps must be performed:

1. Add *rtos::drivers::trace* as a linked library for the desired CMake target application.
2. The target application's compiler arguments must include the *-fxscope* option.
3. The target application's list of sources must include an *.xscope* file with the first probe specified as:

```
<Probe name="freertos_trace" type="CONTINUOUS" datatype="NONE" units="NONE" enabled=
↪"true"/>
```

4. Include *xcore_trace.h* at the end of the RTOS configuration file (i.e. *FreeRTOSConfig.h*).
5. Enable both *configUSE_TRACE_FACILITY* and *configGENERATE_RUN_TIME_STATS* in *FreeRTOSConfig.h*.
6. Continue reading the following sections based on which trace mode is to be used. Additional configuration steps are required.

Tracealyzer Mode The trace driver supports Percepio's Tracealyzer, a feature rich tool for working with trace files. This implementation supports Tracealyzer's *streaming mode*; currently, *snapshot mode* is not supported. The current underlying trace recording implementation interfaces with the *xscope_core_bytes* API function (on Probe 0).

To select Tracealyzer as the trace module's event recorder, the following must be set. This can be applied at the CMake project level:

```
#define USE_TRACE_MODE TRACE_MODE_TRACEALYZER_STREAMING
```

Note: *xcore_trace.h* contains the definition for these modes.

Tracealyzer Initialization In addition to the configuration steps outlined above, Percepio's Tracealyzer streaming mode needs additional function calls to start recording trace data. In the most basic use-case, the following functions should be called on the XCORE tile that is to record trace data:

```
xTraceInitialize();
xTraceEnable(TRC_START);
```

Note: *xTraceInitialize* must be called before any RTOS interaction (before any traced objects are being interacted with). It is advisable to call it as soon as possible in the application.

Tracealyzer Usage The Percepio's Tracealzyer C-unit outputs to a stream-able file format called Percepio Streaming Format (PSF). The *xscope2psf* utility aids in the extraction of the PSF file from the underlying xscope communication (making it readily available on the host's filesystem). This tool can be configured to read from a VCD (value change dump) file that is generated when specifying the *xgdb* option *–xscope-port <ip:port>*, or it can be configured as an xscope-endpoint when specifying the *–xscope-port <ip:port>* option. Both options can be processed by the Tracealyzer graphical tool either as a post processing step or live.

Note: *xscope2psf* currently resides in a Tracealyzer example application here: example. This is likely to change in the future. Refer to either the README or the application's help documentation for usage details.

Note: Currently, the only supported PSF Streaming *target connection* type is *File System*. Ensure this connection type is specified under Tracealyzer's *Recording Settings*.

For general usage of Tracealyzer please refer to the Percepio's documentation here: Manual

ASCII Mode The trace driver supports a basic ASCII mode that is primarily meant as an example for expanding support to other tracing tools/frameworks. In this mode, only the following FreeRTOS trace hooks are supported:

- *traceTASK_SWITCHED_IN*
- *traceTASK_SWITCHED_OUT*

This implementation will produce xscope logs for the RTOS task switching. The underlying xscope API *xscope_core_bytes* is used for communicating this information.

To select ASCII mode as the trace module's event recorder, the following must be set. This can be applied at the CMake project level:

```
#define USE_TRACE_MODE TRACE_MODE_XSCOPE_ASCII
```

Note: *xcore_trace.h* contains the definition for these modes.

ASCII Mode Initialization No additional steps are required for ASCII mode to start recording trace events to xscope.

ASCII Mode Usage To begin capturing ASCII mode traces, run *xgdb* with the *–xscope-file* option. Task switching events will be recorded to the specified VCD (value change dump) file.

3.1.2 XCORE

Clock Control RTOS Driver

This driver can be used to operate GPIO ports on xcore in an RTOS application.

Initialization API The following structures and functions are used to initialize and start a GPIO driver instance.

typedef struct *rtos_clock_control_struct* `rtos_clock_control_t`

Typedef to the RTOS Clock Control driver instance struct.

void `rtos_clock_control_start`(*rtos_clock_control_t* *ctx)

Starts an RTOS clock control driver instance. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before any of the core clock control driver functions are called with this instance.

rtos_clock_control_init() must be called on this clock control driver instance prior to calling this.

Parameters

- `ctx` – A pointer to the clock control driver instance to start.

void `rtos_clock_control_init`(*rtos_clock_control_t* *ctx)

Initializes an RTOS clock control driver instance. There should only be one per tile. This must only be called by the tile that owns the driver instance. It may be called either before or after starting the RTOS, but must be called before calling *rtos_clock_control_start()* or any of the core clock control driver functions with this instance.

Parameters

- `ctx` – A pointer to the GPIO driver instance to initialize.

struct `rtos_clock_control_struct`

#include <rtos_clock_control.h> Struct representing an RTOS clock control driver instance.

The members in this struct should not be accessed directly.

Core API The following functions are the core GPIO driver functions that are used after it has been initialized and started.

inline void `rtos_clock_control_set_ref_clk_div`(*rtos_clock_control_t* *ctx, unsigned divider)

Sets the reference clock divider register value for the tile that owns this driver instance.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.
- `divider` – The value + 1 to write to XS1_SSWITCH_REF_CLK_DIVIDER_NUM

inline unsigned `rtos_clock_control_get_ref_clk_div`(*rtos_clock_control_t* *ctx)

Gets the reference clock divider register value for the tile that owns this driver instance.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.

inline void `rtos_clock_control_set_processor_clk_div`(*rtos_clock_control_t* *ctx, unsigned divider)

Sets the tile clock divider register value for the tile that owns this driver instance.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.
- `divider` – The value + 1 to write to XS1_PSWITCH_PLL_CLK_DIVIDER_NUM

inline unsigned `rtos_clock_control_get_processor_clk_div`(*rtos_clock_control_t* *ctx)

Gets the tile clock divider register value for the tile that owns this driver instance.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.

inline void `rtos_clock_control_set_switch_clk_div`(*rtos_clock_control_t* *ctx, unsigned divider)

Sets the switch clock divider register value for the tile that owns this driver instance.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.
- `divider` – The value + 1 to write to XS1_SSWITCH_CLK_DIVIDER_NUM

inline unsigned `rtos_clock_control_get_switch_clk_div`(*rtos_clock_control_t* *ctx)

Gets the switch clock divider register value for the tile that owns this driver instance.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.

inline unsigned `rtos_clock_control_get_ref_clock`(*rtos_clock_control_t* *ctx)

Gets the calculated reference clock frequency for the tile that owns this driver instance.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.

inline unsigned `rtos_clock_control_get_processor_clock`(*rtos_clock_control_t* *ctx)

Gets the calculated core clock frequency for the tile that owns this driver instance.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.

inline unsigned `rtos_clock_control_get_switch_clock`(*rtos_clock_control_t* *ctx)

Gets the calculated switch clock frequency for the tile that owns this driver instance.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.

inline void `rtos_clock_control_scale_links`(*rtos_clock_control_t* *ctx, unsigned start_addr, unsigned end_addr, unsigned delay_intra, unsigned delay_inter)

Sets the intra token delay and inter token delay to the xlinks within an address range, inclusive, for the tile that owns this driver instance.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.
- `start_addr` – The starting link address
- `end_addr` – The ending address
- `delay_intra` – The intra token delay value
- `delay_inter` – The inter token delay value

inline void `rtos_clock_control_reset_links`(*rtos_clock_control_t* *ctx, unsigned start_addr, unsigned end_addr)

Resets the xlinks within an address range, inclusive for the tile that owns this driver instance.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.
- `start_addr` – The starting link address
- `end_addr` – The ending address

inline void `rtos_clock_control_set_node_pll_ratio`(*rtos_clock_control_t* *ctx, unsigned pre_div, unsigned mul, unsigned post_div)

Sets the tile clock PLL control register value on the tile that owns this driver instance. The value set is calculated from the divider stage 1, multiplier stage, and divider stage 2 values provided.

VCO freq = fosc * (F + 1) / (2 * (R + 1)) VCO must be between 260MHz and 1.3GHz for XS2 Core freq = VCO / (OD + 1)

Refer to the xcore Clock Frequency Control document for more details.

Note: This function will not reset the chip and wait for the PLL to settle before re-enabling the chip to allow for large frequency jumps. This will cause a delay during settings.

Note: It is up to the application to ensure that it is safe to change the clock.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.
- `pre_div` – The value of R
- `mul` – The value of F
- `post_div` – The value of OD

inline void `rtos_clock_control_get_node_pll_ratio`(*rtos_clock_control_t* *ctx, unsigned *pre_div, unsigned *mul, unsigned *post_div)

Gets the divider stage 1, multiplier stage, and divider stage 2 values from the tile clock PLL control register values on the tile that owns this driver instance.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.
- `pre_div` – A pointer to be populated with the value of R
- `mul` – A pointer to be populated with the value of F
- `post_div` – A pointer to be populated with the value of OD

inline void `rtos_clock_control_get_local_lock`(*rtos_clock_control_t* *ctx)

Gets the local lock for clock control on the tile that owns this driver instance. This is intended for applications to use to prevent clock changes around critical sections.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.

inline void `rtos_clock_control_release_local_lock`(*rtos_clock_control_t* *ctx)

Releases the local lock for clock control on the tile that owns this driver instance.

Parameters

- `ctx` – A pointer to the clock control driver instance to use.

RPC Initialization API The following functions may be used to share a GPIO driver instance with other xcore tiles. Tiles that the driver instance is shared with may call any of the core functions listed above.

void `rtos_clock_control_rpc_client_init`(*rtos_clock_control_t* *cc_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *host_intertile_ctx)

Initializes an RTOS clock control driver instance on a client tile. This allows a tile that does not own the actual driver instance to use a driver instance on another tile. This will be called instead of *rtos_clock_control_init()*. The host tile that owns the actual instance must simultaneously call *rtos_clock_control_rpc_host_init()*.

Parameters

- `cc_ctx` – A pointer to the clock control driver instance to initialize.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `cc_ctx`.
- `host_intertile_ctx` – A pointer to the intertile driver instance to use for performing the communication between the client and host tiles. This must have the same scope as `cc_ctx`.

void `rtos_clock_control_rpc_host_init`(*rtos_clock_control_t* *cc_ctx, rtos_driver_rpc_t *rpc_config, *rtos_intertile_t* *client_intertile_ctx[], size_t remote_client_count)

Performs additional initialization on a clock control driver instance to allow client tiles to use the clock control driver instance. Each client tile that will use this instance must simultaneously call *rtos_clock_control_rpc_client_init()*.

Parameters

- `cc_ctx` – A pointer to the clock control driver instance to share with clients.
- `rpc_config` – A pointer to an RPC config struct. This must have the same scope as `cc_ctx`.
- `client_intertile_ctx` – An array of pointers to the intertile driver instances to use for performing the communication between the host tile and each client tile. This must have the same scope as `cc_ctx`.
- `remote_client_count` – The number of client tiles to share this driver instance with.

void `rtos_clock_control_rpc_config`(*rtos_clock_control_t* *cc_ctx, unsigned intertile_port, unsigned host_task_priority)

Configures the RPC for a clock control driver instance. This must be called by both the host tile and all client tiles.

On the client tiles this must be called after calling *rtos_clock_control_rpc_client_init()*. After calling this, the client tile may immediately begin to call the core clock control functions on this driver instance. It does not need to wait for the host to call *rtos_clock_control_start()*.

On the host tile this must be called both after calling *rtos_clock_control_rpc_host_init()* and before calling *rtos_clock_control_start()*.

Parameters

- `cc_ctx` – A pointer to the clock control driver instance to configure the RPC for.
- `intertile_port` – The port number on the intertile channel to use for transferring the RPC requests and responses for this driver instance. This port must not be shared by any other functions. The port must be the same for the host and all its clients.
- `host_task_priority` – The priority to use for the task on the host tile that handles RPC requests from the clients.

Device Firmware Update RTOS Driver

This driver can be used to instantiate and manipulate various flash partitions on xcore in an RTOS application.

For application usage refer to the tutorial *RTOS Application DFU*.

Initialization API The following structures and functions are used to initialize and start a DFU driver instance.

void `rtos_dfu_image_init`(*rtos_dfu_image_t* *dfu_image_ctx, fl_QSPIPorts *qspi_ports, fl_QuadDeviceSpec *qspi_specs, unsigned int len)

Initializes an RTOS DFU image driver instance. This must be called before initializing the RTOS QSPI driver instance.

This will search the flash for program images via libquadflash and store then for application DFU use.

Parameters

- `dfu_image_ctx` – A pointer to the DFU image driver instance to initialize.
- `qspi_ports` – A pointer to the fl_QSPIPorts context to determine which resources to use.
- `qspi_specs` – A pointer to an array of fl_QuadDeviceSpec to try to connect to.
- `len` – The number of fl_QuadDeviceSpec contained in qspi_specs

struct `rtos_dfu_image_t`

#include <rtos_dfu_image.h> Struct representing an RTOS DFU image driver instance.

The members in this struct should not be accessed directly.

Core API The following functions are the core DFU driver functions that are used after it has been initialized and started.

inline unsigned `rtos_dfu_image_get_data_partition_addr`(*rtos_dfu_image_t* *dfu_image_ctx)

Get the starting address of the data partition

Parameters

- `ctx` – A pointer to the DFU image driver instance to use.

Returns

The byte address

inline unsigned `rtos_dfu_image_get_factory_addr`(*rtos_dfu_image_t* *dfu_image_ctx)

Get the starting address of the factory image

Parameters

- `ctx` – A pointer to the DFU image driver instance to use.

Returns

The byte address

inline unsigned `rtos_dfu_image_get_factory_size`(*rtos_dfu_image_t* *dfu_image_ctx)

Get the size of the factory image

Parameters

- `ctx` – A pointer to the DFU image driver instance to use.

Returns

The size in bytes

inline unsigned `rtos_dfu_image_get_factory_version`(*rtos_dfu_image_t* *dfu_image_ctx)

Get the version of the factory image

Parameters

- `ctx` – A pointer to the DFU image driver instance to use.

Returns
The version

inline unsigned `rtos_dfu_image_get_upgrade_addr`(*rtos_dfu_image_t* *dfu_image_ctx)

Get the starting address of the upgrade image

Parameters

- `ctx` – A pointer to the DFU image driver instance to use.

Returns
The byte address

inline unsigned `rtos_dfu_image_get_upgrade_size`(*rtos_dfu_image_t* *dfu_image_ctx)

Get the size of the upgrade image

Parameters

- `ctx` – A pointer to the DFU image driver instance to use.

Returns
The size in bytes

inline unsigned `rtos_dfu_image_get_upgrade_version`(*rtos_dfu_image_t* *dfu_image_ctx)

Get the version of the upgrade image

Parameters

- `ctx` – A pointer to the DFU image driver instance to use.

Returns
The version

void `rtos_dfu_image_print_debug`(*rtos_dfu_image_t* *dfu_image_ctx)

Print debug information

Parameters

- `ctx` – A pointer to the DFU image driver instance to use.

Intertile RTOS Driver

This driver allows for communication between AMP RTOS instances running on different xcore tiles.

Initialization API The following structures and functions are used to initialize and start an intertile driver instance.

void `rtos_intertile_start`(*rtos_intertile_t* *intertile_ctx)

Starts an RTOS intertile driver instance. It may be called either before or after starting the RTOS, but must be called before any of the core intertile driver functions are called with this instance.

rtos_intertile_init() must be called on this intertile driver instance prior to calling this.

Parameters

- `intertile_ctx` – A pointer to the intertile driver instance to start.

void `rtos_intertile_init`(*rtos_intertile_t* *intertile_ctx, chanend_t c)

Initializes an RTOS intertile driver instance. This must be called simultaneously on the two tiles establishing an intertile link. It may be called either before or after starting the RTOS, but must be called before calling *rtos_intertile_start()* or any of the core RTOS intertile functions with this instance.

This establishes a new streaming channel between the two tiles, using the provided non-streaming channel to bootstrap this.

Parameters

- `intertile_ctx` – A pointer to the intertile driver instance to initialize.
- `c` – A channel end that is already allocated and connected to channel end on the tile with which to establish an intertile link. After this function returns, this channel end is no longer needed and may be deallocated or used for other purposes.

struct `rtos_intertile_t`

#include <rtos_intertile.h> Struct representing an RTOS intertile driver instance.

The members in this struct should not be accessed directly.

struct `rtos_intertile_address_t`

#include <rtos_intertile.h> Struct to hold an address to a remote function, consisting of both an intertile instance and a port number. Primarily used by the RPC mechanism in the RTOS drivers.

Core API The following functions are the core intertile driver functions that are used after it has been initialized and started.

void `rtos_intertile_tx_len`(*rtos_intertile_t* *ctx, uint8_t port, size_t len)

size_t `rtos_intertile_tx_data`(*rtos_intertile_t* *ctx, void *data, size_t len)

void `rtos_intertile_tx`(*rtos_intertile_t* *ctx, uint8_t port, void *msg, size_t len)

Transmits data to an intertile link.

Parameters

- `ctx` – A pointer to the intertile driver instance to use.
- `port` – The number of the port to send the data to. Only the thread listening on this particular port on the remote tile will receive this data.
- `msg` – A pointer to the data buffer to transmit.
- `len` – The number of bytes from the buffer to transmit.

size_t `rtos_intertile_rx_len`(*rtos_intertile_t* *ctx, uint8_t port, unsigned timeout)

size_t `rtos_intertile_rx_data`(*rtos_intertile_t* *ctx, void *data, size_t len)

size_t `rtos_intertile_rx`(*rtos_intertile_t* *ctx, uint8_t port, void **msg, unsigned timeout)

Receives data from an intertile link.

Note: the buffer returned via `msg` must be freed by the application using rtos_osal_free().

Note: It is important that no other thread listen on this port simultaneously. If this happens, it is undefined which one will receive the data, and it is possible for a resource exception to occur.

Parameters

- `ctx` – A pointer to the intertile driver instance to use.
- `port` – The number of the port to listen for data on. Only data sent to this port by the remote tile will be received.
- `msg` – A pointer to the received data is written to this pointer variable. This buffer is obtained from the heap and must be freed by the application using rtos_osal_free().
- `timeout` – The amount of time to wait before data become available.

Returns
the number of bytes received.

L2 Cache RTOS Driver

This driver can be used to instantiate a software defined L2 Cache for code and data.

Initialization API The following structures and functions are used to initialize and start an L2 cache driver instance.

typedef struct *rtos_l2_cache_struct* `rtos_l2_cache_t`

Typedef to the RTOS l2 cache driver instance struct.

void `rtos_l2_cache_start`(*rtos_l2_cache_t* *ctx)

Starts the RTOS l2 cache memory driver.

void `rtos_l2_cache_init`(*rtos_l2_cache_t* *ctx, l2_cache_setup_fn setup_fn, l2_cache_thread_fn thread_fn, l2_cache_swmem_read_fn read_func, uint32_t io_core_mask, void *cache_buffer)

Initializes the l2 cache for use by the RTOS l2 cache memory driver.

Cache buffer must be dword aligned

`RTOS_L2_CACHE_DIRECT_MAP`

Convenience macro that may be used to specify the direct map cache to *rtos_l2_cache_init()* in place of setup_fn and thread_fn.

`RTOS_L2_CACHE_TWO_WAY_ASSOCIATIVE`

Convenience macro that may be used to specify the two way associative cache to *rtos_l2_cache_init()* in place of setup_fn and thread_fn.

`RTOS_L2_CACHE_BUFFER_WORDS_DIRECT_MAP`

Convenience macro that may be used to specify the size of the cache buffer for a direct map cache. A pointer to the buffer of size RTOS_L2_CACHE_BUFFER_WORDS_DIRECT_MAP should be passed to the cache_buffer argument of *rtos_l2_cache_init()*.

`RTOS_L2_CACHE_BUFFER_WORDS_TWO_WAY`

Convenience macro that may be used to specify the size of the cache buffer for a two way associative cache. A pointer to the buffer of size RTOS_L2_CACHE_BUFFER_WORDS_TWO_WAY should be passed to the cache_buffer argument of *rtos_l2_cache_init()*.

struct `rtos_l2_cache_struct`

#include <rtos_l2_cache.h> Struct representing an RTOS l2 cache driver instance.

The members in this struct should not be accessed directly.

Software Memory RTOS Driver

This driver allows for implementing application defined software memory in an RTOS.

bool `rtos_swmem_read_request_isr`(unsigned offset, uint32_t *buf)

Services a software memory read request from within the software memory fill interrupt handler. This function may be provided by the application when the software memory driver is initialized with the RTOS_SWMEM_READ_FLAG flag. If the application code to satisfy a fill request requires being run from within an RTOS thread, then *rtos_swmem_read_request()* should be used instead. Both this handler and *rtos_swmem_read_request()* may be used together. If the ISR handler is able to satisfy the request it should return true. If it is not, but the request can be satisfied from within *rtos_swmem_read_request()*, then it should return false.

Parameters

- `offset` – The byte offset into the software memory of the cache line that has had a cache miss.
- `buf` – This function must fill this with SWMEM_EVICT_SIZE_WORDS words of data. Where this data comes from is up to the application. One example is from a flash memory.

Return values

- `true` – if the fill request was satisfied.
- `false` – if the fill request was not satisfied. This requires that *rtos_swmem_read_request()* also be provided.

bool `rtos_swmem_write_request_isr`(unsigned offset, uint32_t dirty_mask, const uint32_t *buf)

Services a software memory write request from within the software memory fill interrupt handler. This function may be provided by the application when the software memory driver is initialized with the RTOS_SWMEM_WRITE_FLAG flag. If the application code to satisfy an evict request requires being run from within an RTOS thread, then *rtos_swmem_write_request()* should be used instead. Both this handler and *rtos_swmem_write_request()* may be used together. If the ISR handler is able to satisfy the request it should return true. If it is not, but the request can be satisfied from within *rtos_swmem_write_request()*, then it should return false.

Parameters

- `offset` – The byte offset into the software memory of the cache line that is being evicted.
- `dirty_mask` – A bytewise dirty mask for the data in `buf`. The least significant bit corresponds to the lowest byte address in `buf` and each subsequent byte address corresponds to the next least significant bit.

- `buf` – A pointer to a buffer containing SWMEM_EVICT_SIZE_WORDS words of data from the cache line being evicted. It is up to the application what it does with this data. One example is to write it to flash memory.

Return values

- `true` – if the evict request was satisifed.
- `false` – if the evict request was not satisfied. This requires that *rtos_swmem_write_request()* also be provided.

void `rtos_swmem_read_request`(unsigned offset, uint32_t *buf)

Services a software memory read request from within the software memory RTOS thread. This function may be provided by the application when the software memory driver is initialized with the RTOS_SWMEM_READ_FLAG flag. If *rtos_swmem_read_request_isr()* is also implemented, then it will be called first. If it is unable to satisfy the request, then this handler will be called. See the description for *rtos_swmem_read_request_isr()*.

Parameters

- `offset` – The byte offset into the software memory of the cache line that has had a cache miss.
- `buf` – This function must fill this with SWMEM_EVICT_SIZE_WORDS words of data. Where this data comes from is up to the application. One example is from a flash memory.

void `rtos_swmem_write_request`(unsigned offset, uint32_t dirty_mask, const uint32_t *buf)

Services a software memory write request from within the software memory RTOS thread. This function may be provided by the application when the software memory driver is initialized with the RTOS_SWMEM_WRITE_FLAG flag. If *rtos_swmem_write_request_isr()* is also implemented, then it will be called first. If it is unable to satisfy the request, then this handler will be called. See the description for *rtos_swmem_write_request_isr()*.

Parameters

- `offset` – The byte offset into the software memory of the cache line that is being evicted.
- `dirty_mask` – A bytewise dirty mask for the data in `buf`. The least significant bit corresponds to the lowest byte address in `buf` and each subsequent byte address corresponds to the next least significant bit.
- `buf` – A pointer to a buffer containing SWMEM_EVICT_SIZE_WORDS words of data from the cache line being evicted. It is up to the application what it does with this data. One example is to write it to flash memory.

void `rtos_swmem_start`(unsigned priority)

Starts the RTOS software memory driver.

Parameters

- `priority` – The priority of the task that gets created by the driver to service the software memory.

void `rtos_swmem_init`(uint32_t init_flags)

Initializes the software memory for use by the RTOS software memory driver.

Parameters

- `init_flags` – A bitfield consisting of initialization flags.
  - RTOS_SWMEM_READ_FLAG enables swmem reads.

- RTOS_SWMEM_WRITE_FLAG enables swmem writes.

unsigned int `rtos_swmem_offset_get()`

Return the offset from XS1_SWMEM_BASE to the start of the software memory.

`RTOS_SWMEM_READ_FLAG`

Flag indicating that software memory reads should be enabled. This should probably always be set when using software memory.

`RTOS_SWMEM_WRITE_FLAG`

Flag indicating that software memory writes should be enabled. This will not always need to be set, especially if flash is backing the software memory and intended to be read only.

3.2 RTOS Services

Several RTOS software services are included to accelerate development of new applications.

3.2.1 Device Control

The Device Control Service provides the ability to configure and control an XMOS device from a host over a number of transport layers. Features of the service include:

- Simple read/write API
- Fully acknowledged protocol
- Includes different transports including I2C and USB.

The table below shows combinations of host and transport mechanisms that are currently supported. Adding new transport layers and/or hosts is straightforward where the hardware supports it.

Table 3.1: Supported Device Control Library Transports

Host	I2C	USB
PC / Windows		Yes
PC / OSX		Yes
Raspberry Pi / Linux	Yes	Yes
xCORE	Yes	

Device Control Shared API

The following structures and functions are common to the control instance on the xcore device and the host.

typedef uint8_t `control_resid_t`

These types are used in control functions to identify the resource id, command, version, and status.

typedef uint8_t `control_cmd_t`

typedef uint8_t `control_version_t`

typedef uint8_t `control_status_t`

enum `control_ret_t`

This type enumerates the possible outcomes from a control transaction.

Values:

enumerator `CONTROL_SUCCESS`

enumerator `CONTROL_REGISTRATION_FAILED`

enumerator `CONTROL_BAD_COMMAND`

enumerator `CONTROL_DATA_LENGTH_ERROR`

enumerator `CONTROL_OTHER_TRANSPORT_ERROR`

enumerator `CONTROL_BAD_RESOURCE`

enumerator `CONTROL_MALFORMED_PACKET`

enumerator `CONTROL_COMMAND_IGNORED_IN_DEVICE`

enumerator `CONTROL_ERROR`

enumerator `SERVICER_COMMAND_RETRY`

enumerator `SERVICER_WRONG_COMMAND_ID`

enumerator `SERVICER_WRONG_COMMAND_LEN`

enumerator `SERVICER_WRONG_PAYLOAD`

enumerator `SERVICER_QUEUE_FULL`

enumerator `SERVICER_SPECIAL_COMMAND_ALREADY_ONGOING`

enumerator `SERVICER_SPECIAL_COMMAND_BUFFER_OVERFLOW`

enumerator `SERVICER_RESOURCE_ERROR`

enumerator SERVICER_SPECIAL_COMMAND_WRONG_ORDER

enumerator SERVICER_SPECIAL_COMMAND_BUF_SIZE_ERROR

enum control_direction_t

This type is used to inform the control library the direction of a control transfer from the transport layer.

Values:

enumerator CONTROL_HOST_TO_DEVICE

enumerator CONTROL_DEVICE_TO_HOST

CONTROL_VERSION

This is the version of control protocol. Used to check compatibility

IS_CONTROL_CMD_READ(c)

Checks if the read bit is set in a command code.

Parameters

- c – **[in]** The command code to check

Returns

true if the read bit in the command is set

Returns

false if the read bit is not set

CONTROL_CMD_SET_READ(c)

Sets the read bit on a command code

Parameters

- c – **[inout]** The command code to set the read bit on.

CONTROL_CMD_SET_WRITE(c)

Clears the read bit on a command code

Parameters

- c – **[inout]** The command code to clear the read bit on.

CONTROL_SPECIAL_RESID

This is the special resource ID owned by the control library. It can be used to check the version of the control protocol. Servicers may not register this resource ID.

CONTROL_MAX_RESOURCE_ID

The maximum resource ID. IDs greater than this cannot be registered.

CONTROL_GET_VERSION

The command to read the version of the control protocol. It must be sent to resource ID CONTROL_SPECIAL_RESID.

`CONTROL_GET_LAST_COMMAND_STATUS`

The command to read the return status of the last command. It must be sent to resource ID CONTROL_SPECIAL_RESID.

`DEVICE_CONTROL_HOST_MODE`

The mode value to use when initializing a device control instance that is on the same tile as its associated transport layer. These may be connected to device control instances on other tiles that have been initialized with DEVICE_CONTROL_CLIENT_MODE.

`DEVICE_CONTROL_CLIENT_MODE`

The mode value to use when initializing a device control instance that is not on the same tile as its associated transport layer. These must be connected to a device control instance on another tile that has been initialized with DEVICE_CONTROL_HOST_MODE.

`DEVICE_CONTROL_CALLBACK_ATTR`

This attribute must be specified on all device control command handler callback functions provided by the application.

Device Control XCORE API

The following structures and functions are used to initialize and start a control instance on the xcore device.

typedef *control_ret_t* (*`device_control_read_cmd_cb_t`)(*control_resid_t* resid, *control_cmd_t* cmd, uint8_t *payload, size_t payload_len, void *app_data)

Function pointer type for application provided device control read command handler callback functions.

Called by *device_control_servicer_cmd_recv()* when a read command is received from the transport layer. The command consists of a resource ID, command value, and a payload_len. This handler must respond with a payload of the requested length.

Param resid
[in] Resource ID. Indicates which resource the command is intended for.

Param cmd
[in] Command code. Note that this will be in the range 0x80 to 0xFF because bit 7 set indicates a read command.

Param payload
[out] Payload bytes of length `payload_len` that will be sent back over the transport layer in response to this read command.

Param payload_len
[in] Requested size of the payload in bytes.

Param app_data
[inout] A pointer to application specific data provided to *device_control_servicer_cmd_recv()*. How and if this is used is entirely up to the application.

Return
CONTROL_SUCCESS if the handling of the read data by the device was successful. An error code otherwise.

typedef *control_ret_t* (*`device_control_write_cmd_cb_t`)(*control_resid_t* resid, *control_cmd_t* cmd, const uint8_t *payload, size_t payload_len, void *app_data)

Function pointer type for application provided device control write command handler callback functions.

Called by *device_control_servicer_cmd_recv()* when a write command is received from the transport layer. The command consists of a resource ID, command value, payload, and the payload's length.

Param resid
[in] Resource ID. Indicates which resource the command is intended for.

Param cmd
[in] Command code. Note that this will be in the range 0x80 to 0xFF because bit 7 set indicates a read command.

Param payload
[in] Payload bytes of length `payload_len`.

Param payload_len
[in] The number of bytes in `payload`.

Param app_data
[inout] A pointer to application specific data provided to *device_control_servicer_cmd_recv()*. How and if this is used is entirely up to the application.

Return
CONTROL_SUCCESS if the handling of the read data by the device was successful. An error code otherwise.

control_ret_t `device_control_request(`*device_control_t* *ctx, *control_resid_t* resid, *control_cmd_t* cmd, size_t payload_len`)`

Must be called by the transport layer when a new request is received.

Precisely how each of the three command parameters resid, cmd, and payload_len are received is specific to the transport layer and not defined by this library.

Parameters

- `ctx` – A pointer to the associated device control instance.
- `resid` – The received resource ID.
- `cmd` – The received command value.
- `payload_len` – The length in bytes of the payload that will follow.

Return values

- `CONTROL_SUCCESS` – if `resid` has been registered by a servicer.
- `CONTROL_BAD_COMMAND` – if `resid` has not been registered by a servicer.

control_ret_t `device_control_payload_transfer(`*device_control_t* *ctx, uint8_t *payload_buf, size_t *buf_size, *control_direction_t* direction`)`

Must be called by the transport layer either when it receives a payload, or when it requires a payload to transmit.

Parameters

- `ctx` – A pointer to the associated device control instance.
- `payload_buf` – A pointer to the payload buffer.
- `buf_size` – A pointer to a variable containing the size of `payload_buf`.

```
                When \p direction is CONTROL_HOST_TO_DEVICE, no more␣
↪than this
                number of bytes will be read from it.

                When \p direction is CONTROL_DEVICE_TO_HOST, this will␣
↪be updated
                to the number of bytes actually written to \p payload_
↪buf.
```

- `direction` – The direction of the payload transfer.

```
                This must be CONTROL_HOST_TO_DEVICE when a payload has␣
↪already
                been received and is inside \p payload_buf.

                This must be CONTROL_DEVICE_TO_HOST when a payload needs␣
↪to be
                written into \p payload_buf by device_control_payload_
↪transfer()
                before sending it.
```

Returns

CONTROL_SUCCESS if everything works and the command is successfully handled by a registered servicer. An error code otherwise.

void `device_control_payload_transfer_bidir`(*device_control_t* *ctx, uint8_t *rx_buf, const size_t rx_size, uint8_t *tx_buf, size_t *tx_size)

Must be called by the transport layer when it receives a payload and requires a payload to transmit, for example, in a SPI transfer. The error status returned by the servicer handling the command is updated in the first byte of the tx_buf.

Parameters

- `ctx` – A pointer to the associated device control instance.
- `rx_buf` – A pointer to the receive payload buffer.
- `rx_size` – A variable containing the size of `rx_buf`.

```
                No more than this
                number of bytes will be read from it.
```

- `tx_buf` – A pointer to the transmitr payload buffer.
- `tx_size` – A pointer variable containing the size of `tx_buf`.

```
                This will be updated
                to the number of bytes actually written to \p tx_buf.
```

control_ret_t `device_control_servicer_cmd_recv`(*device_control_servicer_t* *ctx, *device_control_read_cmd_cb_t* read_cmd_cb, *device_control_write_cmd_cb_t* write_cmd_cb, void *app_data, unsigned timeout)

This is called by servicers to wait for and receive any commands received by the transport layer contain one of the resource IDs registered by the servicer. This is also responsible for responding to read commands.

Parameters

- `ctx` – A pointer to the device control servicer context to receive commands for.
- `read_cmd_cb` – The callback function to handle read commands for all resource IDs associated with the given servicer.
- `write_cmd_cb` – The callback function to handle write commands for all resource IDs associated with the given servicer.
- `app_data` – A pointer to application specific data to pass along to the provided callback functions. How and if this is used is entirely up to the application.
- `timeout` – The number of RTOS ticks to wait before returning if no command is received.

Return values

- `CONTROL_SUCCESS` – if a command successfully received and responded to.
- `CONTROL_ERROR` – if no command is received before the function times out, or if there was a problem communicating back to the transport layer thread.

control_ret_t `device_control_resources_register(`*device_control_t* *ctx, unsigned timeout)

This must be called on the tile that runs the transport layer for the device control instance, and has initialized it with DEVICE_CONTROL_HOST_MODE. This must be called after calling *device_control_start()* and before the transport layer is started. It is to be run simultaneously with *device_control_servicer_register()* from other threads on any tiles associated with the device control instance. The number of servicers that must register is specified by the servicer_count parameter of *device_control_init()*.

Parameters

- `ctx` – A pointer to the device control instance to register resources for.
- `timeout` – The amount of time in RTOS ticks to wait before all servicers register their resource IDs with *device_control_servicer_register()*.

Return values

- `CONTROL_SUCCESS` – if all servicers successfully register their resource IDs before the timeout.
- `CONTROL_REGISTRATION_FAILED` – otherwise.

control_ret_t `device_control_servicer_register(`*device_control_servicer_t* *ctx, *device_control_t* *device_control_ctx[], size_t device_control_ctx_count, const *control_resid_t* resources[], size_t num_resources)

Registers a servicer for a device control instance. Each servicer is responsible for handling any number of resource IDs. All commands received from the transport layer will be forwarded to the servicer that has registered the resource ID that is found in the command.

Servicers may be registered on any tile that has initialized a device control instance. This must be called after calling *device_control_start()*.

Parameters

- `ctx` – A pointer to the device control servicer context to initialize.
- `device_control_ctx` – An array of pointers to the device control instance to register the servicer with.
- `device_control_ctx_count` – The number of device control instances to register the servicer with.
- `resources` – Array of resource IDs to associate with this servicer.
- `num_resources` – The number of resource IDs within `resources`.

control_ret_t `device_control_start(`*device_control_t* *ctx, uint8_t intertile_port, unsigned priority`)`

Starts a device control instance. This must be called by all tiles that have called *device_control_init()*. It may be called either before or after starting the RTOS, but must be called before registering the resources and servicers for this instance.

device_control_init() must be called on this device control instance prior to calling this.

Parameters

- `ctx` – A pointer to the device control instance to start.
- `intertile_port` – The port to use with any and all associated intertile instances associated with this device control instance. If this device control instance is only used by one tile then this is unused.
- `priority` – The priority of the task that will be created if the device control instance was initialized with DEVICE_CONTROL_CLIENT_MODE. This is unused on the tiles where this has been initialized with DEVICE_CONTROL_HOST_MODE. This task is used to listen for commands for a resource ID registered by a servicer running on this tile, but received by the transport layer that is running on another.

control_ret_t `device_control_init(`*device_control_t* *ctx, int mode, size_t servicer_count, *rtos_intertile_t* *intertile_ctx[], size_t intertile_count`)`

Initializes a device control instance.

This must be called by the tile that runs the transport layer (I2C, USB, etc) for the device control instance, as well as all tiles that will register device control servicers for it. It may be called either before or after starting the RTOS, but must be called before calling *device_control_start()*.

Parameters

- `ctx` – A pointer to the device control context to initialize.
- `mode` – Set to DEVICE_CONTROL_HOST_MODE if the command transport layer is on the same tile. Set to DEVICE_CONTROL_CLIENT_MODE if the command transport layer is on another tile.
- `servicer_count` – The number of servicers that will be associated with this device control instance.
- `intertile_ctx` – An array of intertile contexts used to communicate with other tiles.
- `intertile_count` – The number of intertile contexts in the `intertile_ctx` array.

```
                    When \p mode is DEVICE_CONTROL_HOST_MODE, this may␣
↪be 0 if there are
                    no servicers on other tiles, up to one per device␣
↪control instance that
                    has been initialized with DEVICE_CONTROL_CLIENT_MODE␣
↪on other tiles.

                    When \p mode is DEVICE_CONTROL_CLIENT_MODE then this␣
↪must be 1,
                    and the intertile context must connect to a device␣
↪control instance
                    on another tile that has been initialized with␣
↪DEVICE_CONTROL_HOST_MODE.
```

Returns

CONTROL_SUCCESS if the initialization was successful. An error status otherwise.

struct `device_control_t`

#include <device_control.h> Struct representing a device control instance.

The members in this struct should not be accessed directly.

struct `device_control_client_t`

#include <device_control.h> A *device_control_t* pointer may be cast to a pointer to this structure type and used with the device control API, provided it is initialized with DEVICE_CONTROL_CLIENT_MODE. This is not necessary to do, but will save a small amount of memory.

struct `device_control_servicer_t`

#include <device_control.h> Struct representing a device control servicer instance.

The members in this struct should not be accessed directly.

Device Control Host API

The following structures and functions are used to initialize and call a control instance on the host.

control_ret_t `control_init_i2c`(unsigned char i2c_slave_address)

Initialize the I2C host (master) interface

Parameters

- `i2c_slave_address` – I2C address of the slave (controlled device)

Returns

Whether the initialization was successful or not

control_ret_t `control_cleanup_i2c`(void)

Shutdown the I2C host (master) interface connection

Returns

Whether the shutdown was successful or not

control_ret_t `control_init_usb`(int vendor_id, int product_id, int interface_num)

Initialize the USB host interface

Parameters

- `vendor_id` – Vendor ID of controlled USB device
- `product_id` – Product ID of controlled USB device
- `interface_num` – USB Control interface number of controlled device

Returns

Whether the initialization was successful or not

control_ret_t `control_cleanup_usb`(void)

Shutdown the USB host interface connection

Returns

Whether the shutdown was successful or not

control_ret_t `control_init_spi_pi`(spi_mode_t spi_mode, bcm2835SPIClockDivider clock_divider, long intertransation_delay_ns)

Initialize the SPI host (master) interface for the Raspberry Pi

Parameters

- `spi_mode` – Mode that the SPI will run in
- `clock_divider` – The amount to divide the Raspberry Pi's clock by, e.g. BCM2835_SPI_CLOCK_DIVIDER_1024 gives a clock of ~122kHz on the RPI 2.
- `intertransaction_delay` – Delay in nanoseconds that will be applied between each spi transaction. This is implemented with nanosleep() from time.h.

Returns

Whether the initialization was successful or not

control_ret_t `control_cleanup_spi`(void)

Shutdown the SPI host (master) interface connection

Returns

Whether the shutdown was successful or not

control_ret_t `control_query_version`(*control_version_t* *version)

Checks to see that the version of control library in the device is the same as the host

Parameters

- `version` – Reference to control version variable that is set on this call

Returns

Whether the checking of control library version was successful or not

control_ret_t `control_write_command`(*control_resid_t* resid, *control_cmd_t* cmd, const uint8_t payload[], size_t payload_len)

Request to write to controllable resource inside the device. The command consists of a resource ID, command and a byte payload of length payload_len.

Parameters

- `resid` – Resource ID. Indicates which resource the command is intended for
- `cmd` – Command code. Note that this will be in the range 0x80 to 0xFF because bit 7 set indiciates a write command
- `payload` – Array of bytes which constitutes the data payload
- `payload_len` – Size of the payload in bytes

Returns

Whether the write to the device was successful or not

control_ret_t `control_read_command`(*control_resid_t* resid, *control_cmd_t* cmd, uint8_t payload[], size_t payload_len)

Request to read from controllable resource inside the device. The command consists of a resource ID, command and a byte payload of length payload_len.

Parameters

- `resid` – Resource ID. Indicates which resource the command is intended for
- `cmd` – Command code. Note that this will be in the range 0x80 to 0xFF because bit 7 set indiciates a write command
- `payload` – Array of bytes which constitutes the data payload
- `payload_len` – Size of the payload in bytes

Returns
Whether the read from the device was successful or not

Command Transport Protocol

Transport protocol for control parameters Control parameters are converted to an array of bytes in network byte order (big endian) before they're sent over the transport protocol. For example, to set a control parameter to integer value 305419896 which corresponds to hex 0x12345678, the array of bytes sent over the transport protocol would be {0x12, 0x34, 0x56, 0x78}. Similarly, a 4 byte payload {0x00, 0x01, 0x23, 0x22} read over the transport protocol is interpreted as an integer value 0x00012322.

In addition to the control parameters values, commands include Resource ID, the Command ID and Payload Length fields that must be communicated from the host to the device. The Resource ID is an 8-bit identifier that identifies the resource within the device that the command is for. The Command ID is an 8-bit identifier used to identify a command for a resource in the device. Payload length is the length of the data in bytes that the host wants to write to the device or read from the device.

The payload length is interpreted differently for GET_ and SET_ commands. For SET_commands, the payload length is simply the number of bytes worth of control parameters to write to the device. For example, the payload length for a SET_ command to set a control parameter of type int32 to a certain value, would be set to 4. For GET_ commands the payload length is 1 more than the number of bytes of control parameters to read from the device. For example, a GET_ command to read a parameter of type int32, payload length would be set to 5. The one extra byte is used for status and is the first byte (payload[0]) of the payload received from the device. In the example above, payload[0] would be the status byte and payload[1]..payload[4] would be the 4 bytes that make up the value of the control parameter.

The table below lists the different values of the status byte and the action the user is expected to take for each status:

Table 3.2: Values for returned status byte

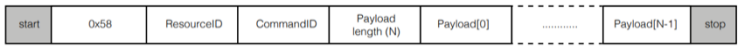
Return code	Values	Description
ctrl_done	0	Read command successful. The payload bytes contain valid payload returned from the device
ctrl_wait	1	Read command not serviced. Retry until ctrl_done status returned
ctrl_invalid	3	Error in read command. Abort and debug

The GET_commands need the extra status byte since the device might not return the control parameter value immediately due to timing constraints. If that is the case the status byte would indicate the status as ctrl_wait and the user would need to retry the command. When returned a ctrl_wait, the user is expected to retry the GET_ command until the status is returned as ctrl_done. The first GET_command is placed in a queue and it will be serviced by the end of each 15ms audio frame. Once the status byte indicates ctrl_done, the rest of the bytes in the payload indicate the control parameter value.

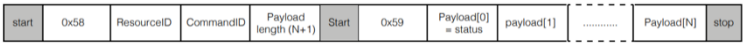
Transporting control parameters over I^2C This section describes the I^2C command sequence when issuing read and write commands to the device.

The first byte sent over I^2C after start contains the device address and information about whether this is an I^2C read transaction or a write transaction. This byte is 0x58 for a write command or 0x59 for a read command. These values are derived by left shifting the device address (0x2c) by 1 and doing a logical OR of the resulting value with 0 for an I^2C write and 1 for an I^2C read.

The bytes sequence sent between I^2C start and stop for SET_ commands is shown in the figure below.



For GET_ commands, the I^{2}C commands sequence consists of a write command followed by a read command with a repeated start between the 2 commands. The write command writes the resource ID, command ID and the expected data length to the device and the read command reads the status byte followed by the rest of the payload that makes up the control parameter value. The figure below shows the I^{2}C bytes sequence sent and received for a GET_ command.



Transporting control parameters over USB Use the vendor_id 0x20B1, product_id 0x0020 and interface number 0 to initialize for USB.

Floating point to fixed point (Q format) conversion Numbers with fractional parts can be represented as floating-point or fixed-point numbers. Floating point formats are widely used but carry performance overheads. Fixed point formats can improve system efficiency and are used extensively within the XVF3610. Fixed point numbers have the position of the decimal point fixed and this is indicated as a part of the format description.

In this document, Q format is used to describe fixed point number formats, with the representation given as Q*m.n* format where *m* is the number of bits reserved for the sign and integer part of the number and *n* is the number of bits reserved for the fractional part of the number. The position of the decimal point is a trade-off between the range of values supported and the resolution provided by the fractional bits.

The dynamic range of Q*m.n* format is -2^{m-1} and $2^{m-1}-2^{-n}$ with a resolution of 2^{-n}

To convert a floating-point format number to Q*m.n* format fixed-point number:

- Multiply the floating-point number by 2^m
- Round the result to the nearest integer
- The resulting integer number is the Q*m.n* fixed-point representation of the initial floating-point number

To convert a Q*m.n* fixed-point number to floating-point:

- Divide the fixed-point number by 2^m
- The resulting decimal number is a floating-point representation of the fixed-point number.

Converting a number into fixed point format and then back to a floating point number may introduce an error of up to $\pm 2^{-(n+1)}$

Example:

To represent a floating-point number 14.765467 in Q8.24 format, the equivalent fixed-point number would be 14.765467 x 2^{24} = 247723429.2 which rounds to 247723429.

To get back the floating-point number given the Q8.24 number 247723429, calculate 247723429 ÷ 2^{24} and get back the floating-point number as 14.76546699. The difference of 0.00000001 is correct to with the error bounds of $\pm 2^{-25}$ which is ±0.00000003

3.2.2 Concurrency Support

The concurrency support sw_service contains a multiple reader single writer lock to support multitheaded applications that need to safely support shared access to a single hardware or software resource. This implementation supports either reader preferred or writer preferred locks.

Concurrency Support API

The following structures and functions are used to initialize a multiple reader single writer lock instance.

enum `mrsw_lock_type_t`

Values:

enumerator `MRSW_READER_PREFERRED`

enumerator `MRSW_WRITER_PREFERRED`

enumerator `MRSW_COUNT`

typedef struct *mrsw_lock* `mrsw_lock_t`

Struct representing an MRSW instance.

The members in this struct should not be accessed directly.

typedef struct *read_pref_mrsw_lock* `read_pref_mrsw_lock_t`

Struct representing an reader preferred MRSW

The members in this struct should not be accessed directly.

typedef struct *write_pref_mrsw_lock* `write_pref_mrsw_lock_t`

Struct representing an writer preferred MRSW

The members in this struct should not be accessed directly.

rtos_osal_status_t `mrsw_lock_create`(*mrsw_lock_t* *ctx, char *name, *mrsw_lock_type_t* type)

Create a MRSW lock

Parameters

- `ctx` – A pointer to an uninitialized lock context
- `name` – An optional ASCII name
- `type` – The type of lock

Returns

RTOS_OSAL_SUCCESS on success

rtos_osal_status_t `mrsw_lock_delete`(*mrsw_lock_t* *ctx)

Destroy a MRSW lock

Note: This does not check if it is safe to delete locks

Parameters

- `ctx` – A pointer to the associated lock context

Returns

RTOS_OSAL_SUCCESS on success RTOS_OSAL_ERROR otherwise

struct `mrsw_lock`

#include <mrsw_lock.h> Struct representing an MRSW instance.

The members in this struct should not be accessed directly.

struct `read_pref_mrsw_lock`

#include <mrsw_lock.h> Struct representing an reader preferred MRSW

The members in this struct should not be accessed directly.

struct `write_pref_mrsw_lock`

#include <mrsw_lock.h> Struct representing an writer preferred MRSW

The members in this struct should not be accessed directly.

The following functions are used to use a multiple reader single writer lock instance as a reader.

rtos_osal_status_t `mrsw_lock_reader_get`(*mrsw_lock_t* *ctx, unsigned timeout)

Attempt to acquire a lock as a reader.

Parameters

- `ctx` – A pointer to the associated lock context
- `timeout` – A timeout before giving up

Returns

RTOS_OSAL_SUCCESS on success RTOS_OSAL_TIMEOUT on timeout RTOS_OSAL_ERROR otherwise

rtos_osal_status_t `mrsw_lock_reader_put`(*mrsw_lock_t* *ctx)

Give an acquired lock as a reader.

Note: User must not give a lock they do not own.

Parameters

- `ctx` – A pointer to the associated lock context

Returns

RTOS_OSAL_SUCCESS on success RTOS_OSAL_ERROR otherwise

The following functions are used to use a multiple reader single writer lock instance as a writer.

rtos_osal_status_t `mrsw_lock_writer_get`(*mrsw_lock_t* *ctx, unsigned timeout)

Attempt to acquire a lock as a writer.

Parameters

- `ctx` – A pointer to the associated lock context
- `timeout` – A timeout before giving up

Returns

RTOS_OSAL_SUCCESS on success RTOS_OSAL_TIMEOUT on timeout RTOS_OSAL_ERROR otherwise

rtos_osal_status_t `mrsw_lock_writer_put`(*mrsw_lock_t* *ctx)

Give an acquired lock as a writer.

Note: User must not give a lock they do not own.

Parameters

- `ctx` – A pointer to the associated lock context

Returns

RTOS_OSAL_SUCCESS on success RTOS_OSAL_ERROR otherwise

3.2.3 Generic Pipeline

The generic pipeline service provides a generic construct to create multithreaded pipelines. This can be used to create a variety of sequential operations on data, such as an audio processing pipeline.

The *generic_pipeline_init()* creates *stage_count* tasks. In the first stage the application provided *input_data* function pointer is called. The data then is passed to the first *stage_function*. After the first state function the data is passed by an RTOS queue to the subsequent stage function. Middle stage functions receive from the previous stage queue, call the stage function, and output to the next stage queue. The last stage function will receive from the previous stage queue, call the stage function, and then call the *output_data* function pointer.

Generic Pipeline Example

This code snippet is an example of creating a pipeline to consume a buffer.

Listing 3.1: Example generic pipeline use

```
static void *input_func(void *input_app_data)
{
    uint32_t* data = pvPortMalloc(100 * sizeof(uint32_t));

    /* Populate some dummy data */
    for(int i=0; i<100; i++)
    {
        data[i] = i;
    }

    return data;
}

static void *output_func(void *data, void *output_app_data)
{
    /* Use data here */
    for(int i=0; i<100; i++)
    {
        rtos_printf("val[%d] = %d\n", i, (uint32_t*)data[i]);
    }

    return 1;   /* Return nonzero value for generic pipeline to implicitly free the packet␣
↪*/
}

static void stage0(void *data)
{
    /* Perform operation on data here*/
    ;
```

(continues on next page)

(continued from previous page)

```
}

static void stage1(void *data)
{
    /* Perform operation on data here*/
    ;
}

static void stage2(void *data)
{
    /* Perform operation on data here*/
    ;
}
```

Listing 3.2: Example generic pipeline use

```
const pipeline_stage_t stages[] = {
    (pipeline_stage_t)stage0,
    (pipeline_stage_t)stage1,
    (pipeline_stage_t)stage2,
};

const configSTACK_DEPTH_TYPE stage_stack_sizes[] = {
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage0) + RTOS_THREAD_STACK_
↪SIZE(input_func),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage1),
    configMINIMAL_STACK_SIZE + RTOS_THREAD_STACK_SIZE(stage2) + RTOS_THREAD_STACK_
↪SIZE(output_func),
};

generic_pipeline_init((pipeline_input_t)input_func,
                      (pipeline_output_t)output_func,
                      NULL,
                      NULL,
                      stages,
                      (const size_t*) stage_stack_sizes,
                      configMAX_PRIORITIES,
                      stage_count);
```

Generic Pipeline API

The following structures and functions are used to initialize and start a generic pipeline instance.

typedef void *(*`pipeline_input_t`)(void *input_data)

Function pointer type for application provided generic pipeline input callback functions.

Called by the first generic_pipeline_stage() when the stage wants input data. This data pointer is provided to the first stage function to be processed.

Param input_data
A pointer to application specific data

Return
A frame pointer to be used by the pipeline stages

typedef int (*`pipeline_output_t`)(void *data, void *output_data)

Function pointer type for application provided generic pipeline output callback functions.

Called by the last generic_pipeline_stage() when the stage wants is done processing the data.

Param data
A pointer to the processed data

Param output_data
A pointer to application specific data

Return
0, to take ownership of data pointer otherwise, request the generic pipeline to free data internally

typedef void (*`pipeline_stage_t`)(void *data)

Function pointer type for application provided generic pipeline stage callback functions.

Called by each generic_pipeline_stage() after input data is received.

Param data
A pointer to the data. This buffer is used for both input and output.

void `generic_pipeline_init`(const *pipeline_input_t* input, const *pipeline_output_t* output, void *const input_data, void *const output_data, const *pipeline_stage_t* *const stage_functions, const size_t *const stage_stack_word_sizes, const int pipeline_priority, const int stage_count)

Create a multistage generic pipeline.

This function will create a multistage pipeline, creating a task per stage and connecting them via queues. Each stage task follows the convention:

- Get input data
- Process data
- Push output data

For the first stage, the input data are the provided by the input callback. For the final stage, the output data are provided to the output callback.

Parameters

- `input` – A function pointer called to get input data
- `output` – A function pointer called to give output data
- `input_data` – A pointer to application specific data to pass to the input callback function
- `output_data` – A pointer to application specific data to pass to the output callback function
- `stage_functions` – An array of stage function pointers
- `stage_stack_word_sizes` – The stack size of each stage. Note: For the first stage must contain enough stack for the stage function + input function. Likewise, the last stage must contain enough stack for the stage function + output function.
- `pipeline_priority` – The priority of all pipeline tasks

- `stage_count` – The number of stages. The limit is 10 stages.

4 FAQs

4.1 What is the memory overhead of the FreeRTOS kernel?

The FreeRTOS kernel can be configured to require as little as 9kB of RAM (per tile). In a typical applicaiton, expect the requirement to be closer to 16kB of RAM (per tile).

4.2 How do I determine the number of words to allocate for use as a task's stack?

Since tasks run within FreeRTOS, the RTOS stack requirement must be known at compile time. In FreeRTOS applications on most other microcontrollers, the general practice is to create a task with a large amount of stack, use the FreeRTOS stack debug functions to determine the worst case runtime usage of stack, and then adjust the stack memory value accordingly. The problem with this method is that the stack of any given thread varies greatly based on the functions that are called within, and thus a code or compiler optimization change result in the optimal task stack usage to have to be redetermined. This issue results in many FreeRTOS applications being written in such a way that wastes memory, by providing task with way more stack than they should need. Additionally, stack overflow bugs can remain hidden for a long time and even when bugs do manifest, the source can be difficult to pinpoint.

The XTC Tools address this issue by creating a symbol that represents the maximum stack requirement of any function at compile time. By using the *RTOS_THREAD_STACK_SIZE()* macro, for the stack words argument for creating a FreeRTOS task, it is guaranteed that the optimal stack requirement is used, provided that the function does not call function pointers nor can infinitely recurse.

```
xTaskCreate((TaskFunction_t) example_task,
            "example_task",
            RTOS_THREAD_STACK_SIZE(example_task),
            NULL,
            EXAMPLE_TASK_PRIORITY,
            NULL);
```

If function pointers are used within a thread, then the application programmer must annotate the code with the appropriate function pointer group attribute. For recursive functions, the only option is to specify the stack manually. See Appendix A - Guiding Stack Size Calculation in the XTC Tools documentation for more information.

4.3 Can I use xcore resources like channels, timers and hw_locks?

You are free to use channels, ports, timers, etc… in your FreeRTOS applications. However, some considerations need to be made. The RTOS kernel knows about RTOS primitives. For example, if RTOS thread A attempts to take a semaphore, the kernel is free to schedule other tasks in thread A's place while thread A is waiting for some other task to give the semaphore. The RTOS kernel does not know anything about xcore resources. For example, if RTOS thread A attempts to *recv* on a channel, the kernel is **not** free to schedule other tasks in its place while thread A is waiting for some other task to send to the other end of the channel. A developer should be aware that blocking calls on xcore resources will block a FreeRTOS thread. This may be OK as long as it is carefully considered in the application design. There are a variety of methods to handle the decoupling of xcore and RTOS

resources. These can be best seen in the various RTOS drivers, which wrap the realtime IO hardware imitation layer.

5 Common Issues

5.1 Task Stack Space

One easy to make mistake in FreeRTOS, is not providing enough stack space for a created task. A vast amount of questions exist online around how to select the FreeRTOS stack size, which the most common answer being to create the task with more than enough stack, force the worst case stack condition (not always trivial), and then use the FreeRTOS debug function *uxTaskGetStackHighWaterMark()* to determine how much you can decrease the stack. This method leaves plenty of room for error and must be done during runtime, and therefore on a build by build basis. The static analysis tools provided by The XTC Tools greatly simplify this process since they calculate the exact stack required for a given function call. The macro *RTOS_THREAD_STACK_SIZE* will return the *nstackwords* symbol for a given thread plus the additional space required for the kernel ISRs. Using this macro for every task create will ensure that there is appropriate stack space for each thread, and thus no stack overflow.

```
xTaskCreate((TaskFunction_t) task_foo,
            "foo",
            RTOS_THREAD_STACK_SIZE(task_foo),
            NULL,
            configMAX_PRIORITIES-1,
            NULL);
```

6 Copyright & Disclaimer

7 Licenses

7.1 XMOS

All original source code is licensed under the XMOS License.

7.2 Third-Party

Additional third party code is included under the following copyrights and licenses:

Table 7.1: Third Party Module Copyrights & Licenses

Module	Copyright & License
Argtable3	Copyright (C) 1998-2001,2003-2011,2013 Stewart Heitmann, licensed under LICENSE
FatFS	Copyright (C) 2017 ChaN, licensed under a BSD-style license
FreeRTOS	Copyright (c) 2017 Amazon.com, Inc., licensed under the MIT License
HTTP Parser	Copyright (c) Joyent, Inc. and other Node contributors, licensed under the MIT license
JSMN JSON Parser	Copyright (c) 2010 Serge A. Zaitsev, licensed under the MIT license
Mbed TLS library	Copyright (c) 2006-2018 ARM Limited, licensed under the Apache License 2.0
Paho MQTT C/C++ client for Embedded platforms	Copyright (c) 2020 The TensorFlow Authors, licensed under the Apache License
TinyUSB	Copyright (c) 2018 hathach (tinyusb.org), licensed under the MIT license

7 Index

P

R

S

U

W

X