Hardware APIs

This group of libraries provides a thin and efficient C API / abstractions to access the RP2040 hardware without having to read and write hardware registers directly.

hardware_adc

Part of: Hardware APIs

Functions

- **void adc_init (void)**

  Initialise the ADC HW.

- **static void adc_gpio_init (uint gpio)**

  Initialise the gpio for use as an ADC pin.

- **static void adc_select_input (uint input)**

  ADC input select.

- **static uint adc_get_selected_input (void)**

  Get the currently selected ADC input channel.

- **static void adc_set_round_robin (uint input_mask)**

  Round Robin sampling selector.

- **static void adc_set_temp_sensor_enabled (bool enable)**

  Enable the onboard temperature sensor.

- **static uint16_t adc_read (void)**

  Perform a single conversion.

- **static void adc_run (bool run)**

  Enable or disable free-running sampling mode.

- **static void adc_set_clkdiv (float clkdiv)**

  Set the ADC Clock divisor.

- **static void adc_fifo_setup (bool en, bool dreq_en, uint16_t dreq_thresh, bool err_in_fifo, bool byte_shift)**

  Setup the ADC FIFO.

- **static bool adc_fifo_is_empty (void)**

  Check FIFO empty state.

- **static uint8_t adc_fifo_get_level (void)**

  Get number of entries in the ADC FIFO.

- **static uint16_t adc_fifo_get (void)**

  Get ADC result from FIFO.

- **static uint16_t adc_fifo_get_blocking (void)**

  Wait for the ADC FIFO to have data.

- **static void adc_fifo_drain (void)**

  Drain the ADC FIFO.

- **static void adc_irq_set_enabled (bool enabled)**

  Enable/Disable ADC interrupts.

Detailed Description

Analog to Digital Converter (ADC) API

The RP2040 has an internal analogue-digital converter (ADC) with the following features:

- SAR ADC
- 500 kS/s (Using an independent 48MHz clock)
- 12 bit (8.7 ENOB)
- 5 input mux:
  - 4 inputs that are available on package pins shared with GPIO[29:26]
  - 1 input is dedicated to the internal temperature sensor
- 4 element receive sample FIFO
- Interrupt generation
- DMA interface

Although there is only one ADC you can specify the input to it using the adc_select_input() function. In round robin mode (adc_set_round_robin()), the ADC will use that input and move to the next one after a read.

User ADC inputs are on 0-3 (GPIO 26-29), the temperature sensor is on input 4.

Temperature sensor values can be approximated in centigrade as:

T = 27 - (ADC_Voltage - 0.706)/0.001721

The FIFO, if used, can contain up to 4 entries.

Example

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/gpio.h"
#include "hardware/adc.h"


int main() {
    stdio_init_all();
    printf("ADC Example, measuring GPIO26\n");

    adc_init();

    // Make sure GPIO is high-impedance, no pullups etc
    adc_gpio_init(26);
    // Select ADC input 0 (GPIO26)
    adc_select_input(0);

    while (1) {
        // 12-bit conversion, assume max value == ADC_VREF == 3.3 V
        const float conversion_factor = 3.3f / (1 << 12);
        uint16_t result = adc_read();
        printf("Raw value: 0x%03x, voltage: %f V\n", result, result * conversion_factor);
        sleep_ms(500);
```

```
    }
}
```

Function Documentation

◆ adc_fifo_drain()

```
static void adc_fifo_drain ( void )
```

inline static

Drain the ADC FIFO.

Will wait for any conversion to complete then drain the FIFO, discarding any results.

◆ adc_fifo_get()

```
static uint16_t adc_fifo_get ( void )
```

inline static

Get ADC result from FIFO.

Pops the latest result from the ADC FIFO.

◆ adc_fifo_get_blocking()

```
static uint16_t adc_fifo_get_blocking ( void )
```

inline static

Wait for the ADC FIFO to have data.

Blocks until data is present in the FIFO

◆ adc_fifo_get_level()

```
static uint8_t adc_fifo_get_level ( void )
```

inline static

Get number of entries in the ADC FIFO.

The ADC FIFO is 4 entries long. This function will return how many samples are currently present.

◆ adc_fifo_is_empty()

```
static bool adc_fifo_is_empty ( void )
```

inline static

Check FIFO empty state.

Returns

Returns true if the FIFO is empty

◆ adc_fifo_setup()

```
static void adc_fifo_setup ( bool     en,
                             bool     dreq_en,
                             uint16_t dreq_thresh,
```

bool *err_in_fifo*,
bool *byte_shift*
)

inline static

Setup the ADC FIFO.

FIFO is 4 samples long, if a conversion is completed and the FIFO is full, the result is dropped.

Parameters

en	Enables write each conversion result to the FIFO
dreq_en	Enable DMA requests when FIFO contains data
dreq_thresh	Threshold for DMA requests/FIFO IRQ if enabled.
err_in_fifo	If enabled, bit 15 of the FIFO contains error flag for each sample
byte_shift	Shift FIFO contents to be one byte in size (for byte DMA) - enables DMA to byte buffers.

◆ adc_get_selected_input()

static uint adc_get_selected_input (void)

inline static

Get the currently selected ADC input channel.

Returns

The currently selected input channel. 0...3 are GPIOs 26...29 respectively. Input 4 is the onboard temperature sensor.

◆ adc_gpio_init()

static void adc_gpio_init (uint *gpio*)

inline static

Initialise the gpio for use as an ADC pin.

Prepare a GPIO for use with ADC by disabling all digital functions.

Parameters

gpio The GPIO number to use. Allowable GPIO numbers are 26 to 29 inclusive.

◆ adc_init()

void adc_init (void)

Initialise the ADC HW.

◆ adc_irq_set_enabled()

static void adc_irq_set_enabled (bool *enabled*)

inline static

Enable/Disable ADC interrupts.

Parameters

enabled Set to true to enable the ADC interrupts, false to disable

◆ adc_read()

static uint16_t adc_read (void) inline static

Perform a single conversion.

Performs an ADC conversion, waits for the result, and then returns it.

Returns

Result of the conversion.

◆ adc_run()

static void adc_run (bool ***run***) inline static

Enable or disable free-running sampling mode.

Parameters

run false to disable, true to enable free running conversion mode.

◆ adc_select_input()

static void adc_select_input (uint ***input***) inline static

ADC input select.

Select an ADC input. 0...3 are GPIOs 26...29 respectively. Input 4 is the onboard temperature sensor.

Parameters

input Input to select.

◆ adc_set_clkdiv()

static void adc_set_clkdiv (float ***clkdiv***) inline static

Set the ADC Clock divisor.

Period of samples will be (1 + div) cycles on average. Note it takes 96 cycles to perform a conversion, so any period less than that will be clamped to 96.

Parameters

clkdiv If non-zero, conversion will be started at intervals rather than back to back.

◆ adc_set_round_robin()

static void adc_set_round_robin (uint ***input_mask***) inline static

Round Robin sampling selector.

This function sets which inputs are to be run through in round robin mode. Value between 0 and 0x1f (bit 0 to bit 4 for GPIO 26 to 29 and temperature sensor input respectively)

Parameters

input_mask A bit pattern indicating which of the 5 inputs are to be sampled. Write a value of 0 to disable round robin sampling.

◆ **adc_set_temp_sensor_enabled()**

static void adc_set_temp_sensor_enabled (bool ***enable***) inline static

Enable the onboard temperature sensor.

Parameters

enable Set true to power on the onboard temperature sensor, false to power off.

hardware_base

Part of: Hardware APIs

Functions

- **`static __force_inline void hw_set_bits (io_rw_32 *addr, uint32_t mask)`**

  Atomically set the specified bits to 1 in a HW register.

- **`static __force_inline void hw_clear_bits (io_rw_32 *addr, uint32_t mask)`**

  Atomically clear the specified bits to 0 in a HW register.

- **`static __force_inline void hw_xor_bits (io_rw_32 *addr, uint32_t mask)`**

  Atomically flip the specified bits in a HW register.

- **`static __force_inline void hw_write_masked (io_rw_32 *addr, uint32_t values, uint32_t write_mask)`**

  Set new values for a sub-set of the bits in a HW register.

Detailed Description

Low-level types and (atomic) accessors for memory-mapped hardware registers

`hardware_base` defines the low level types and access functions for memory mapped hardware registers. It is included by default by all other hardware libraries.

The following register access typedefs codify the access type (read/write) and the bus size (8/16/32) of the hardware register. The register type names are formed by concatenating one from each of the 3 parts A, B, C

A	B	C	Meaning
io_			A Memory mapped IO register
	ro_		read-only access
	rw_		read-write access
	wo_		write-only access (can't actually be enforced via C API)
		8	8-bit wide access

		16	16-bit wide access
		32	32-bit wide access

When dealing with these types, you will always use a pointer, i.e. `io_rw_32 *some_reg` is a pointer to a read/write 32 bit register that you can write with `*some_reg = value`, or read with `value = *some_reg`.

RP2040 hardware is also aliased to provide atomic setting, clear or flipping of a subset of the bits within a hardware register so that concurrent access by two cores is always consistent with one atomic operation being performed first, followed by the second.

See hw_set_bits(), hw_clear_bits() and hw_xor_bits() provide for atomic access via a pointer to a 32 bit register

Additionally given a pointer to a structure representing a piece of hardware (e.g. `dma_hw_t *dma_hw` for the DMA controller), you can get an alias to the entire structure such that writing any member (register) within the structure is equivalent to an atomic operation via hw_set_alias(), hw_clear_alias() or hw_xor_alias()...

For example `hw_set_alias(dma_hw)->inte1 = 0x80;` will set bit 7 of the INTE1 register of the DMA controller, leaving the other bits unchanged.

Function Documentation

◆ hw_clear_bits()

```
static __force_inline void hw_clear_bits ( io_rw_32 *  addr,
                                           uint32_t    mask
                                         )                                    static
```

Atomically clear the specified bits to 0 in a HW register.

Parameters

addr Address of writable register
mask Bit-mask specifying bits to clear

◆ hw_set_bits()

```
static __force_inline void hw_set_bits ( io_rw_32 *  addr,
                                         uint32_t    mask
                                       )                                      static
```

Atomically set the specified bits to 1 in a HW register.

Parameters

addr Address of writable register
mask Bit-mask specifying bits to set

◆ hw_write_masked()

```
static __force_inline void hw_write_masked ( io_rw_32 *  addr,
                                             uint32_t    values,
                                             uint32_t    write_mask
                                           )                                  static
```

Set new values for a sub-set of the bits in a HW register.

Sets destination bits to values specified in `values`, if and only if corresponding bit in `write_mask` is set

Note: this method allows safe concurrent modification of *different* bits of a register, but multiple concurrent access to the same bits is still unsafe.

Parameters

addr	Address of writable register
values	Bits values
write_mask	Mask of bits to change

◆ hw_xor_bits()

```
static __force_inline void hw_xor_bits ( io_rw_32 *  addr,
                                         uint32_t    mask
                                       )                                  static
```

Atomically flip the specified bits in a HW register.

Parameters

addr	Address of writable register
mask	Bit-mask specifying bits to invert

hardware_claim

Part of: Hardware APIs

Functions

- **void hw_claim_or_assert (uint8_t *bits, uint bit_index, const char *message)**

  Atomically claim a resource, panicking if it is already in use.

- **int hw_claim_unused_from_range (uint8_t *bits, bool required, uint bit_lsb, uint bit_msb, const char *message)**

  Atomically claim one resource out of a range of resources, optionally asserting if none are free.

- **bool hw_is_claimed (const uint8_t *bits, uint bit_index)**

  Determine if a resource is claimed at the time of the call.

- **void hw_claim_clear (uint8_t *bits, uint bit_index)**

  Atomically unclaim a resource.

- **uint32_t hw_claim_lock (void)**

  Acquire the runtime mutual exclusion lock provided by the `hardware_claim` library.

- **void hw_claim_unlock (uint32_t token)**

  Release the runtime mutual exclusion lock provided by the `hardware_claim` library.

Detailed Description

Lightweight hardware resource management

`hardware_claim` provides a simple API for management of hardware resources at runtime.

This API is usually called by other hardware specific *claiming* APIs and provides simple multi-core safe methods to manipulate compact bit-sets representing hardware resources.

This API allows any other library to cooperatively participate in a scheme by which both compile time and runtime allocation of resources can co-exist, and conflicts can be avoided or detected (depending on the use case) without the libraries having any other knowledge of each other.

Facilities are providing for:

1. Claiming resources (and asserting if they are already claimed)
2. Freeing (unclaiming) resources
3. Finding unused resources

Function Documentation

◆ hw_claim_clear()

```
void hw_claim_clear ( uint8_t *  bits,
                      uint       bit_index
                    )
```

Atomically unclaim a resource.

The resource ownership is indicated by the bit_index bit in an array of bits.

Parameters

bits	pointer to an array of bits (8 bits per byte)
bit_index	resource to unclaim (bit index into array of bits)

◆ hw_claim_lock()

```
uint32_t hw_claim_lock ( void  )
```

Acquire the runtime mutual exclusion lock provided by the `hardware_claim` library.

This method is called automatically by the other `hw_claim_` methods, however it is provided as a convenience to code that might want to protect other hardware initialization code from concurrent use.

NOTE

hw_claim_lock() uses a spin lock internally, so disables interrupts on the calling core, and will deadlock if the calling core already owns the lock.

Returns

a token to pass to hw_claim_unlock()

◆ hw_claim_or_assert()

```
void hw_claim_or_assert ( uint8_t *       bits,
                          uint            bit_index,
                          const char *    message
                        )
```

Atomically claim a resource, panicking if it is already in use.

The resource ownership is indicated by the bit_index bit in an array of bits.

Parameters

bits	pointer to an array of bits (8 bits per byte)
bit_index	resource to claim (bit index into array of bits)
message	string to display if the bit cannot be claimed; note this may have a single printf format "%d" for the bit

◆ hw_claim_unlock()

```
void hw_claim_unlock ( uint32_t  token )
```

Release the runtime mutual exclusion lock provided by the `hardware_claim` library.

NOTE

This method MUST be called from the same core that call hw_claim_lock()

Parameters

token	the token returned by the corresponding call to hw_claim_lock()

◆ hw_claim_unused_from_range()

```
int hw_claim_unused_from_range ( uint8_t *       bits,
                                 bool            required,
                                 uint            bit_lsb,
                                 uint            bit_msb,
                                 const char *    message
                               )
```

Atomically claim one resource out of a range of resources, optionally asserting if none are free.

Parameters

bits	pointer to an array of bits (8 bits per byte)
required	true if this method should panic if the resource is not free
bit_lsb	the lower bound (inclusive) of the resource range to claim from
bit_msb	the upper bound (inclusive) of the resource range to claim from
message	string to display if the bit cannot be claimed

Returns

the bit index representing the claimed or -1 if none are available in the range, and required = false

◆ hw_is_claimed()

```
bool hw_is_claimed ( const uint8_t *  bits,
                     uint             bit_index
                   )                                                    inline
```

Determine if a resource is claimed at the time of the call.

The resource ownership is indicated by the bit_index bit in an array of bits.

Parameters

bits	pointer to an array of bits (8 bits per byte)
bit_index	resource to check (bit index into array of bits)

Returns

true if the resource is claimed

hardware_clocks

Part of: Hardware APIs

Typedefs

- **typedef void(* resus_callback_t) (void)**

  Resus callback function type.

Enumerations

- **enum clock_index {**
  clk_gpout0 = 0 , clk_gpout1 , clk_gpout2 , clk_gpout3 ,
  clk_ref , clk_sys , clk_peri , clk_usb ,
  clk_adc , clk_rtc , CLK_COUNT
  }

  Enumeration identifying a hardware clock. More...

Functions

- **void clocks_init (void)**

  Initialise the clock hardware.

- **bool clock_configure (enum clock_index clk_index, uint32_t src, uint32_t auxsrc, uint32_t src_freq, uint32_t freq)**

  Configure the specified clock.

- **void clock_stop (enum clock_index clk_index)**

  Stop the specified clock.

- **uint32_t clock_get_hz (enum clock_index clk_index)**

  Get the current frequency of the specified clock.

- **uint32_t frequency_count_khz (uint src)**

  Measure a clocks frequency using the Frequency counter.

- **void clock_set_reported_hz (enum clock_index clk_index, uint hz)**

  Set the "current frequency" of the clock as reported by clock_get_hz without actually changing the clock.

- **void clocks_enable_resus (resus_callback_t resus_callback)**

  Enable the resus function. Restarts clk_sys if it is accidentally stopped.

- **void clock_gpio_init_int_frac (uint gpio, uint src, uint32_t div_int, uint8_t div_frac)**

  Output an optionally divided clock to the specified gpio pin.

- **static void clock_gpio_init (uint gpio, uint src, float div)**

  Output an optionally divided clock to the specified gpio pin.

- **bool clock_configure_gpin (enum clock_index clk_index, uint gpio, uint32_t src_freq, uint32_t freq)**

Configure a clock to come from a gpio input.

Detailed Description

Clock Management API

This API provides a high level interface to the clock functions.

The clocks block provides independent clocks to on-chip and external components. It takes inputs from a variety of clock sources allowing the user to trade off performance against cost, board area and power consumption. From these sources it uses multiple clock generators to provide the required clocks. This architecture allows the user flexibility to start and stop clocks independently and to vary some clock frequencies whilst maintaining others at their optimum frequencies

Please refer to the datasheet for more details on the RP2040 clocks.

The clock source depends on which clock you are attempting to configure. The first table below shows main clock sources. If you are not setting the Reference clock or the System clock, or you are specifying that one of those two will be using an auxiliary clock source, then you will need to use one of the entries from the subsequent tables.

Main Clock Sources

Source	Reference Clock	System Clock
ROSC	CLOCKS_CLK_REF_CTRL_SRC_VALUE_ROSC_CLKSRC_PH	
Auxiliary	CLOCKS_CLK_REF_CTRL_SRC_VALUE_CLKSRC_CLK_REF_AUX	CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_
XOSC	CLOCKS_CLK_REF_CTRL_SRC_VALUE_XOSC_CLKSRC	
Reference		CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLK_REF

Auxiliary Clock Sources

The auxiliary clock sources available for use in the configure function depend on which clock is being configured. The following table describes the available values that can be used. Note that for clk_gpout[x], x can be 0-3.

Aux Source	clk_gpout[x]	clk_ref
System PLL	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS	
GPIO in 0	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLKSRC_GPIN0	CLOCKS_CLK_REF_CTRL_AUXSRC_VALUE_CLKSRC_GPIN
GPIO in 1	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLKSRC_GPIN1	CLOCKS_CLK_REF_CTRL_AUXSRC_VALUE_CLKSRC_GPIN
USB PLL	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB	CLOCKS_CLK_REF_CTRL_AUXSRC_VALUE_CLKSRC_PLL_
ROSC	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_ROSC_CLKSRC	
XOSC	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_XOSC_CLKSRC	
System clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_SYS	
USB Clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_USB	
ADC clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_ADC	
RTC Clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_RTC	
Ref clock	CLOCKS_CLK_GPOUTx_CTRL_AUXSRC_VALUE_CLK_REF	

Aux Source	clk_peri	clk_usb
System PLL	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS	CLOCKS_CLK_USB_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SY
GPIO in 0	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLKSRC_GPIN0	CLOCKS_CLK_USB_CTRL_AUXSRC_VALUE_CLKSRC_GPIN0
GPIO in 1	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLKSRC_GPIN1	CLOCKS_CLK_USB_CTRL_AUXSRC_VALUE_CLKSRC_GPIN1

USB PLL	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB	CLOCKS_CLK_USB_CTRL_AUXSRC_VALUE_CLKSRC_PLL_US
ROSC	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_ROSC_CLKSRC_PH	CLOCKS_CLK_USB_CTRL_AUXSRC_VALUE_ROSC_CLKSRC_F
XOSC	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_XOSC_CLKSRC	CLOCKS_CLK_USB_CTRL_AUXSRC_VALUE_XOSC_CLKSRC
System clock	CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLK_SYS	

Aux Source	clk_rtc
System PLL	CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS
GPIO in 0	CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_CLKSRC_GPIN0
GPIO in 1	CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_CLKSRC_GPIN1
USB PLL	CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB
ROSC	CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_ROSC_CLKSRC_PH
XOSC	CLOCKS_CLK_RTC_CTRL_AUXSRC_VALUE_XOSC_CLKSRC

Example

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/pll.h"
#include "hardware/clocks.h"
#include "hardware/structs/pll.h"
#include "hardware/structs/clocks.h"


void measure_freqs(void) {
    uint f_pll_sys = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_PLL_SYS_CLKSRC_PRIMARY);
    uint f_pll_usb = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_PLL_USB_CLKSRC_PRIMARY);
    uint f_rosc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_ROSC_CLKSRC);
    uint f_clk_sys = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_SYS);
    uint f_clk_peri = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_PERI);
    uint f_clk_usb = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_USB);
    uint f_clk_adc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_ADC);
    uint f_clk_rtc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_RTC);

    printf("pll_sys  = %dkHz\n", f_pll_sys);
    printf("pll_usb  = %dkHz\n", f_pll_usb);
    printf("rosc     = %dkHz\n", f_rosc);
    printf("clk_sys  = %dkHz\n", f_clk_sys);
    printf("clk_peri = %dkHz\n", f_clk_peri);
    printf("clk_usb  = %dkHz\n", f_clk_usb);
    printf("clk_adc  = %dkHz\n", f_clk_adc);
    printf("clk_rtc  = %dkHz\n", f_clk_rtc);

    // Can't measure clk_ref / xosc as it is the ref
}


int main() {
    stdio_init_all();

    printf("Hello, world!\n");

    measure_freqs();

    // Change clk_sys to be 48MHz. The simplest way is to take this from PLL_USB
    // which has a source frequency of 48MHz
    clock_configure(clk_sys,
                    CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX,
                    CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_USB,
                    48 * MHZ,
                    48 * MHZ);

    // Turn off PLL sys for good measure
    pll_deinit(pll_sys);

    // CLK peri is clocked from clk_sys so need to change clk_peri's freq
    clock_configure(clk_peri,
                    0,
                    CLOCKS_CLK_PERI_CTRL_AUXSRC_VALUE_CLK_SYS,
                    48 * MHZ,
                    48 * MHZ);

    // Re init uart now that clk_peri has changed
    stdio_init_all();

    measure_freqs();
    printf("Hello, 48MHz");

    return 0;
}
```

Typedef Documentation

◆ resus_callback_t

typedef void(* resus_callback_t) (void)

Resus callback function type.

User provided callback for a resus event (when clk_sys is stopped by the programmer and is restarted for them).

Enumeration Type Documentation

◆ clock_index

enum clock_index

Enumeration identifying a hardware clock.

Enumerator	
clk_gpout0	GPIO Muxing 0.
clk_gpout1	GPIO Muxing 1.
clk_gpout2	GPIO Muxing 2.
clk_gpout3	GPIO Muxing 3.
clk_ref	Watchdog and timers reference clock.
clk_sys	Processors, bus fabric, memory, memory mapped registers.
clk_peri	Peripheral clock for UART and SPI.
clk_usb	USB clock.
clk_adc	ADC clock.
clk_rtc	Real time clock.

Function Documentation

◆ clock_configure()

```
bool clock_configure ( enum clock_index  clk_index,
                       uint32_t          src,
                       uint32_t          auxsrc,
                       uint32_t          src_freq,
                       uint32_t          freq
                     )
```

Configure the specified clock.

See the tables in the description for details on the possible values for clock sources.

Parameters

clk_index	The clock to configure
src	The main clock source, can be 0.
auxsrc	The auxiliary clock source, which depends on which clock is being set. Can be 0
src_freq	Frequency of the input clock source

freq	Requested frequency

◆ clock_configure_gpin()

```
bool clock_configure_gpin ( enum clock_index  clk_index,
                            uint              gpio,
                            uint32_t          src_freq,
                            uint32_t          freq
                          )
```

Configure a clock to come from a gpio input.

Parameters

clk_index	The clock to configure
gpio	The GPIO pin to run the clock from. Valid GPIOs are: 20 and 22.
src_freq	Frequency of the input clock source
freq	Requested frequency

◆ clock_get_hz()

```
uint32_t clock_get_hz ( enum clock_index  clk_index )
```

Get the current frequency of the specified clock.

Parameters

clk_index	Clock

Returns

Clock frequency in Hz

◆ clock_gpio_init()

```
static void clock_gpio_init ( uint   gpio,
                              uint   src,
                              float  div
                            )                                    inline  static
```

Output an optionally divided clock to the specified gpio pin.

Parameters

gpio	The GPIO pin to output the clock to. Valid GPIOs are: 21, 23, 24, 25. These GPIOs are connected to the GPOUT0-3 clock generators.
src	The source clock. See the register field CLOCKS_CLK_GPOUT0_CTRL_AUXSRC for a full list. The list is the same for each GPOUT clock generator.
div	The float amount to divide the source clock by. This is useful to not overwhelm the GPIO pin with a fast clock.

◆ clock_gpio_init_int_frac()

```
void clock_gpio_init_int_frac ( uint      gpio,
                                uint      src,
                                uint32_t  div_int,
```

uint8_t *div_frac*
)

Output an optionally divided clock to the specified gpio pin.

Parameters

gpio	The GPIO pin to output the clock to. Valid GPIOs are: 21, 23, 24, 25. These GPIOs are connected to the GPOUT0-3 clock generators.
src	The source clock. See the register field CLOCKS_CLK_GPOUT0_CTRL_AUXSRC for a full list. The list is the same for each GPOUT clock generator.
div_int	The integer part of the value to divide the source clock by. This is useful to not overwhelm the GPIO pin with a fast clock. this is in range of 1..2^24-1.
div_frac	The fractional part of the value to divide the source clock by. This is in range of 0..255 (/256).

◆ clock_set_reported_hz()

void clock_set_reported_hz (enum clock_index *clk_index*,
uint *hz*
)

Set the "current frequency" of the clock as reported by clock_get_hz without actually changing the clock.

See also clock_get_hz()

◆ clock_stop()

void clock_stop (enum clock_index *clk_index*)

Stop the specified clock.

Parameters

clk_index The clock to stop

◆ clocks_enable_resus()

void clocks_enable_resus (resus_callback_t *resus_callback*)

Enable the resus function. Restarts clk_sys if it is accidentally stopped.

The resuscitate function will restart the system clock if it falls below a certain speed (or stops). This could happen if the clock source the system clock is running from stops. For example if a PLL is stopped.

Parameters

resus_callback a function pointer provided by the user to call if a resus event happens.

◆ clocks_init()

void clocks_init (void)

Initialise the clock hardware.

Must be called before any other clock function.

◆ **frequency_count_khz()**

uint32_t frequency_count_khz (uint ***src***)

Measure a clocks frequency using the Frequency counter.

Uses the inbuilt frequency counter to measure the specified clocks frequency. Currently, this function is accurate to +-1KHz. See the datasheet for more details.

hardware_divider

Part of: Hardware APIs

Functions

- **static void hw_divider_divmod_s32_start (int32_t a, int32_t b)**

  Start a signed asynchronous divide.

- **static void hw_divider_divmod_u32_start (uint32_t a, uint32_t b)**

  Start an unsigned asynchronous divide.

- **static void hw_divider_wait_ready (void)**

  Wait for a divide to complete.

- **static divmod_result_t hw_divider_result_nowait (void)**

  Return result of HW divide, nowait.

- **static divmod_result_t hw_divider_result_wait (void)**

  Return result of last asynchronous HW divide.

- **static uint32_t hw_divider_u32_quotient_wait (void)**

  Return result of last asynchronous HW divide, unsigned quotient only.

- **static int32_t hw_divider_s32_quotient_wait (void)**

  Return result of last asynchronous HW divide, signed quotient only.

- **static uint32_t hw_divider_u32_remainder_wait (void)**

  Return result of last asynchronous HW divide, unsigned remainder only.

- **static int32_t hw_divider_s32_remainder_wait (void)**

  Return result of last asynchronous HW divide, signed remainder only.

- **divmod_result_t hw_divider_divmod_s32 (int32_t a, int32_t b)**

  Do a signed HW divide and wait for result.

- **divmod_result_t hw_divider_divmod_u32 (uint32_t a, uint32_t b)**

  Do an unsigned HW divide and wait for result.

- **static uint32_t to_quotient_u32 (divmod_result_t r)**

  Efficient extraction of unsigned quotient from 32p32 fixed point.

- **static int32_t to_quotient_s32 (divmod_result_t r)**

  Efficient extraction of signed quotient from 32p32 fixed point.

- **static uint32_t to_remainder_u32 (divmod_result_t r)**

  Efficient extraction of unsigned remainder from 32p32 fixed point.

- **static int32_t to_remainder_s32 (divmod_result_t r)**

  Efficient extraction of signed remainder from 32p32 fixed point.

- **static uint32_t hw_divider_u32_quotient (uint32_t a, uint32_t b)**

  Do an unsigned HW divide, wait for result, return quotient.

- **static uint32_t hw_divider_u32_remainder (uint32_t a, uint32_t b)**

  Do an unsigned HW divide, wait for result, return remainder.

- **static int32_t hw_divider_quotient_s32 (int32_t a, int32_t b)**

  Do a signed HW divide, wait for result, return quotient.

- **static int32_t hw_divider_remainder_s32 (int32_t a, int32_t b)**

  Do a signed HW divide, wait for result, return remainder.

- **static void hw_divider_pause (void)**

  Pause for exact amount of time needed for a asynchronous divide to complete.

- **static uint32_t hw_divider_u32_quotient_inlined (uint32_t a, uint32_t b)**

  Do a hardware unsigned HW divide, wait for result, return quotient.

- **static uint32_t hw_divider_u32_remainder_inlined (uint32_t a, uint32_t b)**

  Do a hardware unsigned HW divide, wait for result, return remainder.

- **static int32_t hw_divider_s32_quotient_inlined (int32_t a, int32_t b)**

  Do a hardware signed HW divide, wait for result, return quotient.

- **static int32_t hw_divider_s32_remainder_inlined (int32_t a, int32_t b)**

  Do a hardware signed HW divide, wait for result, return remainder.

- **void hw_divider_save_state (hw_divider_state_t *dest)**

  Save the calling cores hardware divider state.

- **void hw_divider_restore_state (hw_divider_state_t *src)**

  Load a saved hardware divider state into the current core's hardware divider.

Detailed Description

Low-level hardware-divider access

The SIO contains an 8-cycle signed/unsigned divide/modulo circuit, per core. Calculation is started by writing a dividend and divisor to the two argument registers, DIVIDEND and DIVISOR. The divider calculates the quotient / and remainder % of this division over the next 8 cycles, and on the 9th cycle the results can be read from the two result registers DIV_QUOTIENT and DIV_REMAINDER. A 'ready' bit in register DIV_CSR can be polled to wait for the calculation to complete, or software can insert a fixed 8-cycle delay

This header provides low level macros and inline functions for accessing the hardware dividers directly, and perhaps most usefully performing asynchronous divides. These functions however do not follow the regular SDK conventions

for saving/restoring the divider state, so are not generally safe to call from interrupt handlers

The pico_divider library provides a more user friendly set of APIs over the divider (and support for 64 bit divides), and of course by default regular C language integer divisions are redirected through that library, meaning you can just use C level `/` and `%` operators and gain the benefits of the fast hardware divider.

See also pico_divider

Example

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/divider.h"


int main() {
    stdio_init_all();
    printf("Hello, divider!\n");

    // This is the basic hardware divider function
    int32_t dividend = 123456;
    int32_t divisor = -321;
    divmod_result_t result = hw_divider_divmod_s32(dividend, divisor);

    printf("%d/%d = %d remainder %d\n", dividend, divisor, to_quotient_s32(result), to_remainde
r_s32(result));

    // Is it right?

    printf("Working backwards! Result %d should equal %d!\n\n",
           to_quotient_s32(result) * divisor + to_remainder_s32(result), dividend);

    // This is the recommended unsigned fast divider for general use.
    int32_t udividend = 123456;
    int32_t udivisor = 321;
    divmod_result_t uresult = hw_divider_divmod_u32(udividend, udivisor);

    printf("%d/%d = %d remainder %d\n", udividend, udivisor, to_quotient_u32(uresult), to_remai
nder_u32(uresult));

    // Is it right?

    printf("Working backwards! Result %d should equal %d!\n\n",
           to_quotient_u32(result) * divisor + to_remainder_u32(result), dividend);

    // You can also do divides asynchronously. Divides will be complete after 8 cycles.

    hw_divider_divmod_s32_start(dividend, divisor);

    // Do something for 8 cycles!

    // In this example, our results function will wait for completion.
    // Use hw_divider_result_nowait() if you don't want to wait, but are sure you have delayed
at least 8 cycles

    result = hw_divider_result_wait();

    printf("Async result %d/%d = %d remainder %d\n", dividend, divisor, to_quotient_s32(resul
t),
           to_remainder_s32(result));

    // For a really fast divide, you can use the inlined versions... the / involves a function
call as / always does
    // when using the ARM AEABI, so if you really want the best performance use the inlined ver
sions.
    // Note that the / operator function DOES use the hardware divider by default, although you
can change
    // that behavior by calling pico_set_divider_implementation in the cmake build for your tar
get.
    printf("%d / %d = (by operator %d) (inlined %d)\n", dividend, divisor,
           dividend / divisor, hw_divider_s32_quotient_inlined(dividend, divisor));

    // Note however you must manually save/restore the divider state if you call the inlined me
thods from within an IRQ
    // handler.
    hw_divider_state_t state;
    hw_divider_divmod_s32_start(dividend, divisor);
    hw_divider_save_state(&state);

    hw_divider_divmod_s32_start(123, 7);
    printf("inner %d / %d = %d\n", 123, 7, hw_divider_s32_quotient_wait());

    hw_divider_restore_state(&state);
    int32_t tmp = hw_divider_s32_quotient_wait();
    printf("outer divide %d / %d = %d\n", dividend, divisor, tmp);
    return 0;
}
```

Function Documentation

◆ hw_divider_divmod_s32()

```
divmod_result_t hw_divider_divmod_s32 ( int32_t a,
                                        int32_t b
                                      )
```

Do a signed HW divide and wait for result.

Divide **a** by **b**, wait for calculation to complete, return result as a pair of 32-bit quotient/remainder values.

Parameters

a The dividend
b The divisor

Returns

Results of divide as a pair of 32-bit quotient/remainder values.

◆ hw_divider_divmod_s32_start()

```
static void hw_divider_divmod_s32_start ( int32_t a,
                                          int32_t b
                                        )                                inline static
```

Start a signed asynchronous divide.

Start a divide of the specified signed parameters. You should wait for 8 cycles (__div_pause()) or wait for the ready bit to be set (hw_divider_wait_ready()) prior to reading the results.

Parameters

a The dividend
b The divisor

◆ hw_divider_divmod_u32()

```
divmod_result_t hw_divider_divmod_u32 ( uint32_t a,
                                        uint32_t b
                                      )
```

Do an unsigned HW divide and wait for result.

Divide **a** by **b**, wait for calculation to complete, return result as a pair of 32-bit quotient/remainder values.

Parameters

a The dividend
b The divisor

Returns

Results of divide as a pair of 32-bit quotient/remainder values.

◆ hw_divider_divmod_u32_start()

```
static void hw_divider_divmod_u32_start ( uint32_t a,
                                          uint32_t b
                                        )                                    inline static
```

Start an unsigned asynchronous divide.

Start a divide of the specified unsigned parameters. You should wait for 8 cycles (__div_pause()) or wait for the ready bit to be set (hw_divider_wait_ready()) prior to reading the results.

Parameters

a The dividend
b The divisor

◆ hw_divider_quotient_s32()

```
static int32_t hw_divider_quotient_s32 ( int32_t a,
                                         int32_t b
                                       )                                     inline static
```

Do a signed HW divide, wait for result, return quotient.

Divide a by b, wait for calculation to complete, return quotient.

Parameters

a The dividend
b The divisor

Returns

Quotient results of the divide

◆ hw_divider_remainder_s32()

```
static int32_t hw_divider_remainder_s32 ( int32_t a,
                                          int32_t b
                                        )                                    inline static
```

Do a signed HW divide, wait for result, return remainder.

Divide a by b, wait for calculation to complete, return remainder.

Parameters

a The dividend
b The divisor

Returns

Remainder results of the divide

◆ hw_divider_restore_state()

void hw_divider_restore_state (hw_divider_state_t * ***src***)

Load a saved hardware divider state into the current core's hardware divider.

Copy the passed hardware divider state into the hardware divider.

Parameters

src the location to load the divider state from

◆ hw_divider_result_nowait()

static divmod_result_t hw_divider_result_nowait (void) inline static

Return result of HW divide, nowait.

NOTE

This is UNSAFE in that the calculation may not have been completed.

Returns

Current result. Most significant 32 bits are the remainder, lower 32 bits are the quotient.

◆ hw_divider_result_wait()

static divmod_result_t hw_divider_result_wait (void) inline static

Return result of last asynchronous HW divide.

This function waits for the result to be ready by calling hw_divider_wait_ready().

Returns

Current result. Most significant 32 bits are the remainder, lower 32 bits are the quotient.

◆ hw_divider_s32_quotient_inlined()

static int32_t hw_divider_s32_quotient_inlined (int32_t ***a***,
int32_t ***b***
) inline static

Do a hardware signed HW divide, wait for result, return quotient.

Divide **a** by **b**, wait for calculation to complete, return quotient.

Parameters

a The dividend
b The divisor

Returns

Quotient result of the divide

◆ hw_divider_s32_quotient_wait()

```
static int32_t hw_divider_s32_quotient_wait ( void )
```

inline static

Return result of last asynchronous HW divide, signed quotient only.

This function waits for the result to be ready by calling hw_divider_wait_ready().

Returns

Current signed quotient result.

◆ hw_divider_s32_remainder_inlined()

```
static int32_t hw_divider_s32_remainder_inlined ( int32_t a,
                                                  int32_t b
                                                )
```

inline static

Do a hardware signed HW divide, wait for result, return remainder.

Divide **a** by **b**, wait for calculation to complete, return remainder.

Parameters

a The dividend
b The divisor

Returns

Remainder result of the divide

◆ hw_divider_s32_remainder_wait()

```
static int32_t hw_divider_s32_remainder_wait ( void )
```

inline static

Return result of last asynchronous HW divide, signed remainder only.

This function waits for the result to be ready by calling hw_divider_wait_ready().

Returns

Current remainder results.

◆ hw_divider_save_state()

```
void hw_divider_save_state ( hw_divider_state_t * dest )
```

Save the calling cores hardware divider state.

Copy the current core's hardware divider state into the provided structure. This method waits for the divider results to be stable, then copies them to memory. They can be restored via hw_divider_restore_state()

Parameters

dest the location to store the divider state

◆ hw_divider_u32_quotient()

```
static uint32_t hw_divider_u32_quotient ( uint32_t a,
                                          uint32_t b
                                        )
```
inline static

Do an unsigned HW divide, wait for result, return quotient.

Divide **a** by **b**, wait for calculation to complete, return quotient.

Parameters

a The dividend
b The divisor

Returns

Quotient results of the divide

◆ hw_divider_u32_quotient_inlined()

```
static uint32_t hw_divider_u32_quotient_inlined ( uint32_t a,
                                                  uint32_t b
                                                )
```
inline static

Do a hardware unsigned HW divide, wait for result, return quotient.

Divide **a** by **b**, wait for calculation to complete, return quotient.

Parameters

a The dividend
b The divisor

Returns

Quotient result of the divide

◆ hw_divider_u32_quotient_wait()

```
static uint32_t hw_divider_u32_quotient_wait ( void )
```
inline static

Return result of last asynchronous HW divide, unsigned quotient only.

This function waits for the result to be ready by calling hw_divider_wait_ready().

Returns

Current unsigned quotient result.

◆ hw_divider_u32_remainder()

```
static uint32_t hw_divider_u32_remainder ( uint32_t a,
```

uint32_t *b*
)

inline static

Do an unsigned HW divide, wait for result, return remainder.

Divide **a** by **b**, wait for calculation to complete, return remainder.

Parameters

a The dividend
b The divisor

Returns

Remainder results of the divide

◆ hw_divider_u32_remainder_inlined()

static uint32_t hw_divider_u32_remainder_inlined (uint32_t *a*,
uint32_t *b*
)

inline static

Do a hardware unsigned HW divide, wait for result, return remainder.

Divide **a** by **b**, wait for calculation to complete, return remainder.

Parameters

a The dividend
b The divisor

Returns

Remainder result of the divide

◆ hw_divider_u32_remainder_wait()

static uint32_t hw_divider_u32_remainder_wait (void)

inline static

Return result of last asynchronous HW divide, unsigned remainder only.

This function waits for the result to be ready by calling hw_divider_wait_ready().

Returns

Current unsigned remainder result.

◆ hw_divider_wait_ready()

static void hw_divider_wait_ready (void)

inline static

Wait for a divide to complete.

Wait for a divide to complete

◆ to_quotient_s32()

static int32_t to_quotient_s32 (divmod_result_t *r*) inline static

Efficient extraction of signed quotient from 32p32 fixed point.

Parameters

r A pair of 32-bit quotient/remainder values.

Returns

Unsigned quotient

◆ to_quotient_u32()

static uint32_t to_quotient_u32 (divmod_result_t *r*) inline static

Efficient extraction of unsigned quotient from 32p32 fixed point.

Parameters

r A pair of 32-bit quotient/remainder values.

Returns

Unsigned quotient

◆ to_remainder_s32()

static int32_t to_remainder_s32 (divmod_result_t *r*) inline static

Efficient extraction of signed remainder from 32p32 fixed point.

Parameters

r A pair of 32-bit quotient/remainder values.

Returns

Signed remainder

NOTE

On arm this is just a 32 bit register move or a nop

◆ to_remainder_u32()

static uint32_t to_remainder_u32 (divmod_result_t *r*) inline static

Efficient extraction of unsigned remainder from 32p32 fixed point.

Parameters

r A pair of 32-bit quotient/remainder values.

Returns

Unsigned remainder

NOTE

On Arm this is just a 32 bit register move or a nop

hardware_dma

Part of: Hardware APIs

Modules

- **channel_config**

  DMA channel configuration.

Enumerations

- **enum dma_channel_transfer_size { DMA_SIZE_8 = 0 , DMA_SIZE_16 = 1 , DMA_SIZE_32 = 2 }**

  Enumeration of available DMA channel transfer sizes. More...

Functions

- **void dma_channel_claim (uint channel)**

  Mark a dma channel as used.

- **void dma_claim_mask (uint32_t channel_mask)**

  Mark multiple dma channels as used.

- **void dma_channel_unclaim (uint channel)**

  Mark a dma channel as no longer used.

- **void dma_unclaim_mask (uint32_t channel_mask)**

  Mark multiple dma channels as no longer used.

- **int dma_claim_unused_channel (bool required)**

  Claim a free dma channel.

- **bool dma_channel_is_claimed (uint channel)**

  Determine if a dma channel is claimed.

- **static void dma_channel_set_config (uint channel, const dma_channel_config *config, bool trigger)**

  Set a channel configuration.

- **static void dma_channel_set_read_addr (uint channel, const volatile void *read_addr, bool trigger)**

  Set the DMA initial read address.

- **static void dma_channel_set_write_addr (uint channel, volatile void *write_addr, bool trigger)**

  Set the DMA initial write address.

- **static void dma_channel_set_trans_count (uint channel, uint32_t trans_count, bool trigger)**

  Set the number of bus transfers the channel will do.

- **static void dma_channel_configure (uint channel, const dma_channel_config *config, volatile void *write_addr, const volatile void *read_addr, uint transfer_count, bool trigger)**

  Configure all DMA parameters and optionally start transfer.

- **static void dma_channel_transfer_from_buffer_now (uint channel, const volatile void *read_addr, uint32_t transfer_count)**

  Start a DMA transfer from a buffer immediately.

- **static void dma_channel_transfer_to_buffer_now (uint channel, volatile void *write_addr, uint32_t transfer_count)**

  Start a DMA transfer to a buffer immediately.

- **static void dma_start_channel_mask (uint32_t chan_mask)**

  Start one or more channels simultaneously.

- **static void dma_channel_start (uint channel)**

  Start a single DMA channel.

- **static void dma_channel_abort (uint channel)**

  Stop a DMA transfer.

- **static void dma_channel_set_irq0_enabled (uint channel, bool enabled)**

  Enable single DMA channel's interrupt via DMA_IRQ_0.

- **static void dma_set_irq0_channel_mask_enabled (uint32_t channel_mask, bool enabled)**

  Enable multiple DMA channels' interrupts via DMA_IRQ_0.

- **static void dma_channel_set_irq1_enabled (uint channel, bool enabled)**

  Enable single DMA channel's interrupt via DMA_IRQ_1.

- **static void dma_set_irq1_channel_mask_enabled (uint32_t channel_mask, bool enabled)**

  Enable multiple DMA channels' interrupts via DMA_IRQ_1.

- **static void dma_irqn_set_channel_enabled (uint irq_index, uint channel, bool enabled)**

  Enable single DMA channel interrupt on either DMA_IRQ_0 or DMA_IRQ_1.

- **static void dma_irqn_set_channel_mask_enabled (uint irq_index, uint32_t channel_mask, bool enabled)**

  Enable multiple DMA channels' interrupt via either DMA_IRQ_0 or DMA_IRQ_1.

- **static bool dma_channel_get_irq0_status (uint channel)**

  Determine if a particular channel is a cause of DMA_IRQ_0.

- **static bool dma_channel_get_irq1_status (uint channel)**

Determine if a particular channel is a cause of DMA_IRQ_1.

- **static bool dma_irqn_get_channel_status (uint irq_index, uint channel)**

  Determine if a particular channel is a cause of DMA_IRQ_N.

- **static void dma_channel_acknowledge_irq0 (uint channel)**

  Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_0.

- **static void dma_channel_acknowledge_irq1 (uint channel)**

  Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_1.

- **static void dma_irqn_acknowledge_channel (uint irq_index, uint channel)**

  Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_N.

- **static bool dma_channel_is_busy (uint channel)**

  Check if DMA channel is busy.

- **static void dma_channel_wait_for_finish_blocking (uint channel)**

  Wait for a DMA channel transfer to complete.

- **static void dma_sniffer_enable (uint channel, uint mode, bool force_channel_enable)**

  Enable the DMA sniffing targeting the specified channel.

- **static void dma_sniffer_set_byte_swap_enabled (bool swap)**

  Enable the Sniffer byte swap function.

- **static void dma_sniffer_set_output_invert_enabled (bool invert)**

  Enable the Sniffer output invert function.

- **static void dma_sniffer_set_output_reverse_enabled (bool reverse)**

  Enable the Sniffer output bit reversal function.

- **static void dma_sniffer_disable (void)**

  Disable the DMA sniffer.

- **static void dma_sniffer_set_data_accumulator (uint32_t seed_value)**

  Set the sniffer's data accumulator with initial value.

- **static uint32_t dma_sniffer_get_data_accumulator (void)**

  Get the sniffer's data accumulator value.

- **void dma_timer_claim (uint timer)**

  Mark a dma timer as used.

- **void dma_timer_unclaim (uint timer)**

  Mark a dma timer as no longer used.

- **int dma_claim_unused_timer (bool required)**

  Claim a free dma timer.

- **bool dma_timer_is_claimed (uint timer)**

  Determine if a dma timer is claimed.

- **`static void dma_timer_set_fraction (uint timer, uint16_t numerator, uint16_t denominator)`**

  Set the divider for the given DMA timer.

- **`static uint dma_get_timer_dreq (uint timer_num)`**

  Return the DREQ number for a given DMA timer.

- **`void dma_channel_cleanup (uint channel)`**

  Performs DMA channel cleanup after use.

Detailed Description

DMA Controller API

The RP2040 Direct Memory Access (DMA) master performs bulk data transfers on a processor's behalf. This leaves processors free to attend to other tasks, or enter low-power sleep states. The data throughput of the DMA is also significantly higher than one of RP2040's processors.

The DMA can perform one read access and one write access, up to 32 bits in size, every clock cycle. There are 12 independent channels, which each supervise a sequence of bus transfers, usually in one of the following scenarios:

- Memory to peripheral
- Peripheral to memory
- Memory to memory

Enumeration Type Documentation

◆ dma_channel_transfer_size

enum dma_channel_transfer_size

Enumeration of available DMA channel transfer sizes.

Names indicate the number of bits.

Enumerator	
DMA_SIZE_8	Byte transfer (8 bits)
DMA_SIZE_16	Half word transfer (16 bits)
DMA_SIZE_32	Word transfer (32 bits)

Function Documentation

◆ dma_channel_abort()

static void dma_channel_abort (uint ***channel***) inline static

Stop a DMA transfer.

Function will only return once the DMA has stopped.

Note that due to errata RP2040-E13, aborting a channel which has transfers in-flight (i.e. an individual read has taken place but the corresponding write has not), the ABORT status bit will clear prematurely, and subsequently the in-flight transfers will trigger a completion interrupt once they complete.

The effect of this is that you *may* see a spurious completion interrupt on the channel as a result of calling this method.

The calling code should be sure to ignore a completion IRQ as a result of this method. This may not require any additional work, as aborting a channel which may be about to complete, when you have a completion IRQ handler registered, is inherently race-prone, and so code is likely needed to disambiguate the two occurrences.

If that is not the case, but you do have a channel completion IRQ handler registered, you can simply disable/re-enable the IRQ around the call to this method as shown by this code fragment (using DMA IRQ0).

```
// disable the channel on IRQ0

dma_channel_set_irq0_enabled(channel, false);
// abort the channel

dma_channel_abort(channel);
// clear the spurious IRQ (if there was one)

dma_channel_acknowledge_irq0(channel);
// re-enable the channel on IRQ0

dma_channel_set_irq0_enabled(channel, true);
```

Parameters

channel DMA channel

◆ dma_channel_acknowledge_irq0()

static void dma_channel_acknowledge_irq0 (uint ***channel***) inline static

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_0.

Parameters

channel DMA channel

◆ dma_channel_acknowledge_irq1()

static void dma_channel_acknowledge_irq1 (uint ***channel***) inline static

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_1.

Parameters

channel DMA channel

◆ dma_channel_claim()

void dma_channel_claim (uint ***channel***)

Mark a dma channel as used.

Method for cooperative claiming of hardware. Will cause a panic if the channel is already claimed. Use of this method by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters

channel the dma channel

◆ dma_channel_cleanup()

void dma_channel_cleanup (uint ***channel***)

Performs DMA channel cleanup after use.

This can be used to cleanup dma channels when they're no longer needed, such that they are in a clean state for reuse. IRQ's for the channel are disabled, any in flight-transfer is aborted and any outstanding interrupts are cleared. The channel is then clear to be reused for other purposes.

```
if (dma_channel >= 0) {
    dma_channel_cleanup(dma_channel);
    dma_channel_unclaim(dma_channel);
    dma_channel = -1;
}
```

Parameters

channel DMA channel

◆ dma_channel_configure()

static void dma_channel_configure (uint ***channel***,
const dma_channel_config * ***config***,
volatile void * ***write_addr***,
const volatile void * ***read_addr***,
uint ***transfer_count***,
bool ***trigger***
)

inline static

Configure all DMA parameters and optionally start transfer.

Parameters

channel	DMA channel
config	Pointer to DMA config structure
write_addr	Initial write address
read_addr	Initial read address
transfer_count	Number of transfers to perform
trigger	True to start the transfer immediately

◆ dma_channel_get_irq0_status()

static bool dma_channel_get_irq0_status (uint ***channel***)

inline static

Determine if a particular channel is a cause of DMA_IRQ_0.

Parameters

channel DMA channel

Returns

true if the channel is a cause of DMA_IRQ_0, false otherwise

◆ dma_channel_get_irq1_status()

static bool dma_channel_get_irq1_status (uint ***channel***) inline static

Determine if a particular channel is a cause of DMA_IRQ_1.

Parameters

channel DMA channel

Returns

true if the channel is a cause of DMA_IRQ_1, false otherwise

◆ dma_channel_is_busy()

static bool dma_channel_is_busy (uint ***channel***) inline static

Check if DMA channel is busy.

Parameters

channel DMA channel

Returns

true if the channel is currently busy

◆ dma_channel_is_claimed()

bool dma_channel_is_claimed (uint ***channel***)

Determine if a dma channel is claimed.

Parameters

channel the dma channel

Returns

true if the channel is claimed, false otherwise

See also dma_channel_claim dma_channel_claim_mask

◆ dma_channel_set_config()

```
static void dma_channel_set_config ( uint                          channel,
                                     const dma_channel_config *    config,
                                     bool                          trigger
                                   )                                          inline static
```

Set a channel configuration.

Parameters

channel DMA channel
config Pointer to a config structure with required configuration

trigger True to trigger the transfer immediately

◆ dma_channel_set_irq0_enabled()

```
static void dma_channel_set_irq0_enabled ( uint channel,
                                           bool enabled
                                         )                                  inline static
```

Enable single DMA channel's interrupt via DMA_IRQ_0.

Parameters

channel DMA channel
enabled true to enable interrupt 0 on specified channel, false to disable.

◆ dma_channel_set_irq1_enabled()

```
static void dma_channel_set_irq1_enabled ( uint channel,
                                           bool enabled
                                         )                                  inline static
```

Enable single DMA channel's interrupt via DMA_IRQ_1.

Parameters

channel DMA channel
enabled true to enable interrupt 1 on specified channel, false to disable.

◆ dma_channel_set_read_addr()

```
static void dma_channel_set_read_addr ( uint                  channel,
                                        const volatile void * read_addr,
                                        bool                  trigger
                                      )                                     inline static
```

Set the DMA initial read address.

Parameters

channel DMA channel
read_addr Initial read address of transfer.
trigger True to start the transfer immediately

◆ dma_channel_set_trans_count()

```
static void dma_channel_set_trans_count ( uint     channel,
                                          uint32_t trans_count,
                                          bool     trigger
                                        )                                   inline static
```

Set the number of bus transfers the channel will do.

Parameters

channel DMA channel

trans_count	The number of transfers (not NOT bytes, see channel_config_set_transfer_data_size)
trigger	True to start the transfer immediately

◆ dma_channel_set_write_addr()

```
static void dma_channel_set_write_addr ( uint              channel,
                                         volatile void *   write_addr,
                                         bool              trigger
                                       )                                  inline  static
```

Set the DMA initial write address.

Parameters

channel	DMA channel
write_addr	Initial write address of transfer.
trigger	True to start the transfer immediately

◆ dma_channel_start()

```
static void dma_channel_start ( uint  channel )                           inline  static
```

Start a single DMA channel.

Parameters

channel	DMA channel

◆ dma_channel_transfer_from_buffer_now()

```
static void dma_channel_transfer_from_buffer_now ( uint                     channel,
                                                   const volatile void *    read_addr,
                                                   uint32_t                 transfer_count
                                                 )                          inline  static
```

Start a DMA transfer from a buffer immediately.

Parameters

channel	DMA channel
read_addr	Sets the initial read address
transfer_count	Number of transfers to make. Not bytes, but the number of transfers of channel_config_set_transfer_data_size() to be sent.

◆ dma_channel_transfer_to_buffer_now()

```
static void dma_channel_transfer_to_buffer_now ( uint              channel,
                                                 volatile void *   write_addr,
                                                 uint32_t          transfer_count
                                               )                            inline  static
```

Start a DMA transfer to a buffer immediately.

Parameters

channel	DMA channel
write_addr	Sets the initial write address
transfer_count	Number of transfers to make. Not bytes, but the number of transfers of channel_config_set_transfer_data_size() to be sent.

◆ dma_channel_unclaim()

void dma_channel_unclaim (uint ***channel***)

Mark a dma channel as no longer used.

Parameters

channel the dma channel to release

◆ dma_channel_wait_for_finish_blocking()

static void dma_channel_wait_for_finish_blocking (uint ***channel***) inline static

Wait for a DMA channel transfer to complete.

Parameters

channel DMA channel

◆ dma_claim_mask()

void dma_claim_mask (uint32_t ***channel_mask***)

Mark multiple dma channels as used.

Method for cooperative claiming of hardware. Will cause a panic if any of the channels are already claimed. Use of this method by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters

channel_mask Bitfield of all required channels to claim (bit 0 == channel 0, bit 1 == channel 1 etc)

◆ dma_claim_unused_channel()

int dma_claim_unused_channel (bool ***required***)

Claim a free dma channel.

Parameters

required if true the function will panic if none are available

Returns

the dma channel number or -1 if required was false, and none were free

◆ dma_claim_unused_timer()

int dma_claim_unused_timer (bool ***required***)

Claim a free dma timer.

Parameters

required if true the function will panic if none are available

Returns

the dma timer number or -1 if required was false, and none were free

◆ dma_get_timer_dreq()

static uint dma_get_timer_dreq (uint ***timer_num***) inline static

Return the DREQ number for a given DMA timer.

Parameters

timer_num DMA timer number 0-3

◆ dma_irqn_acknowledge_channel()

static void dma_irqn_acknowledge_channel (uint ***irq_index***,
uint ***channel***
) inline static

Acknowledge a channel IRQ, resetting it as the cause of DMA_IRQ_N.

Parameters

irq_index the IRQ index; either 0 or 1 for DMA_IRQ_0 or DMA_IRQ_1
channel DMA channel

◆ dma_irqn_get_channel_status()

static bool dma_irqn_get_channel_status (uint ***irq_index***,
uint ***channel***
) inline static

Determine if a particular channel is a cause of DMA_IRQ_N.

Parameters

irq_index the IRQ index; either 0 or 1 for DMA_IRQ_0 or DMA_IRQ_1
channel DMA channel

Returns

true if the channel is a cause of the DMA_IRQ_N, false otherwise

◆ dma_irqn_set_channel_enabled()

```
static void dma_irqn_set_channel_enabled ( uint irq_index,
                                           uint channel,
                                           bool enabled
                                         )                                    inline static
```

Enable single DMA channel interrupt on either DMA_IRQ_0 or DMA_IRQ_1.

Parameters

irq_index	the IRQ index; either 0 or 1 for DMA_IRQ_0 or DMA_IRQ_1
channel	DMA channel
enabled	true to enable interrupt via irq_index for specified channel, false to disable.

◆ dma_irqn_set_channel_mask_enabled()

```
static void dma_irqn_set_channel_mask_enabled ( uint     irq_index,
                                                uint32_t channel_mask,
                                                bool     enabled
                                              )                              inline static
```

Enable multiple DMA channels' interrupt via either DMA_IRQ_0 or DMA_IRQ_1.

Parameters

irq_index	the IRQ index; either 0 or 1 for DMA_IRQ_0 or DMA_IRQ_1
channel_mask	Bitmask of all the channels to enable/disable. Channel 0 = bit 0, channel 1 = bit 1 etc.
enabled	true to enable all the interrupts specified in the mask, false to disable all the interrupts specified in the mask.

◆ dma_set_irq0_channel_mask_enabled()

```
static void dma_set_irq0_channel_mask_enabled ( uint32_t channel_mask,
                                                bool     enabled
                                              )                              inline static
```

Enable multiple DMA channels' interrupts via DMA_IRQ_0.

Parameters

channel_mask	Bitmask of all the channels to enable/disable. Channel 0 = bit 0, channel 1 = bit 1 etc.
enabled	true to enable all the interrupts specified in the mask, false to disable all the interrupts specified in the mask.

◆ dma_set_irq1_channel_mask_enabled()

```
static void dma_set_irq1_channel_mask_enabled ( uint32_t channel_mask,
                                                bool     enabled
                                              )                              inline static
```

Enable multiple DMA channels' interrupts via DMA_IRQ_1.

Parameters

channel_mask	Bitmask of all the channels to enable/disable. Channel 0 = bit 0, channel 1 = bit 1 etc.

enabled	true to enable all the interrupts specified in the mask, false to disable all the interrupts specified in the mask.

◆ dma_sniffer_disable()

```
static void dma_sniffer_disable ( void )
```

inline static

Disable the DMA sniffer.

◆ dma_sniffer_enable()

```
static void dma_sniffer_enable ( uint channel,
                                 uint mode,
                                 bool force_channel_enable
                               )
```

inline static

Enable the DMA sniffing targeting the specified channel.

The mode can be one of the following:

Mode	Function
0x0	Calculate a CRC-32 (IEEE802.3 polynomial)
0x1	Calculate a CRC-32 (IEEE802.3 polynomial) with bit reversed data
0x2	Calculate a CRC-16-CCITT
0x3	Calculate a CRC-16-CCITT with bit reversed data
0xe	XOR reduction over all data. == 1 if the total 1 population count is odd.
0xf	Calculate a simple 32-bit checksum (addition with a 32 bit accumulator)

Parameters

channel	DMA channel
mode	See description
force_channel_enable	Set true to also turn on sniffing in the channel configuration (this is usually what you want, but sometimes you might have a chain DMA with only certain segments of the chain sniffed, in which case you might pass false).

◆ dma_sniffer_get_data_accumulator()

```
static uint32_t dma_sniffer_get_data_accumulator ( void )
```

inline static

Get the sniffer's data accumulator value.

Read value calculated by the hardware from sniffing the DMA stream

◆ dma_sniffer_set_byte_swap_enabled()

```
static void dma_sniffer_set_byte_swap_enabled ( bool swap )
```

inline static

Enable the Sniffer byte swap function.

Locally perform a byte reverse on the sniffed data, before feeding into checksum.

Note that the sniff hardware is downstream of the DMA channel byteswap performed in the read master: if channel_config_set_bswap() and dma_sniffer_set_byte_swap_enabled() are both enabled, their effects cancel from the sniffer's point of view.

Parameters

swap Set true to enable byte swapping

◆ dma_sniffer_set_data_accumulator()

static void dma_sniffer_set_data_accumulator (uint32_t ***seed_value***) inline static

Set the sniffer's data accumulator with initial value.

Generally, CRC algorithms are used with the data accumulator initially seeded with 0xFFFF or 0xFFFFFFFF (for crc16 and crc32 algorithms)

Parameters

seed_value value to set data accumulator

◆ dma_sniffer_set_output_invert_enabled()

static void dma_sniffer_set_output_invert_enabled (bool ***invert***) inline static

Enable the Sniffer output invert function.

If enabled, the sniff data result appears bit-inverted when read. This does not affect the way the checksum is calculated.

Parameters

invert Set true to enable output bit inversion

◆ dma_sniffer_set_output_reverse_enabled()

static void dma_sniffer_set_output_reverse_enabled (bool ***reverse***) inline static

Enable the Sniffer output bit reversal function.

If enabled, the sniff data result appears bit-reversed when read. This does not affect the way the checksum is calculated.

Parameters

reverse Set true to enable output bit reversal

◆ dma_start_channel_mask()

static void dma_start_channel_mask (uint32_t ***chan_mask***) inline static

Start one or more channels simultaneously.

Parameters

chan_mask Bitmask of all the channels requiring starting. Channel 0 = bit 0, channel 1 = bit 1 etc.

◆ dma_timer_claim()

void dma_timer_claim (uint ***timer***)

Mark a dma timer as used.

Method for cooperative claiming of hardware. Will cause a panic if the timer is already claimed. Use of this method by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters

timer the dma timer

◆ dma_timer_is_claimed()

bool dma_timer_is_claimed (uint ***timer***)

Determine if a dma timer is claimed.

Parameters

timer the dma timer

Returns

true if the timer is claimed, false otherwise

See also dma_timer_claim

◆ dma_timer_set_fraction()

```
static void dma_timer_set_fraction ( uint     timer,
                                     uint16_t numerator,
                                     uint16_t denominator
                                   )                                  inline  static
```

Set the divider for the given DMA timer.

The timer will run at the system_clock_freq * numerator / denominator, so this is the speed that data elements will be transferred at via a DMA channel using this timer as a DREQ

Parameters

timer	the dma timer
numerator	the fraction's numerator
denominator	the fraction's denominator

◆ dma_timer_unclaim()

void dma_timer_unclaim (uint ***timer***)

Mark a dma timer as no longer used.

Method for cooperative claiming of hardware.

Parameters

timer the dma timer to release

◆ **dma_unclaim_mask()**

void dma_unclaim_mask (uint32_t ***channel_mask***)

Mark multiple dma channels as no longer used.

Parameters

channel_mask Bitfield of all channels to unclaim (bit 0 == channel 0, bit 1 == channel 1 etc)

channel_config

Part of: Hardware APIs » hardware_dma
DMA channel configuration. More...

Functions

- **static void channel_config_set_read_increment (dma_channel_config *c, bool incr)**

  Set DMA channel read increment in a channel configuration object.

- **static void channel_config_set_write_increment (dma_channel_config *c, bool incr)**

  Set DMA channel write increment in a channel configuration object.

- **static void channel_config_set_dreq (dma_channel_config *c, uint dreq)**

  Select a transfer request signal in a channel configuration object.

- **static void channel_config_set_chain_to (dma_channel_config *c, uint chain_to)**

  Set DMA channel chain_to channel in a channel configuration object.

- **static void channel_config_set_transfer_data_size (dma_channel_config *c, enum dma_channel_transfer_size size)**

  Set the size of each DMA bus transfer in a channel configuration object.

- **static void channel_config_set_ring (dma_channel_config *c, bool write, uint size_bits)**

  Set address wrapping parameters in a channel configuration object.

- **static void channel_config_set_bswap (dma_channel_config *c, bool bswap)**

  Set DMA byte swapping config in a channel configuration object.

- **static void channel_config_set_irq_quiet (dma_channel_config *c, bool irq_quiet)**

  Set IRQ quiet mode in a channel configuration object.

- **static void channel_config_set_high_priority (dma_channel_config *c, bool high_priority)**

  Set the channel priority in a channel configuration object.

- **static void channel_config_set_enable (dma_channel_config *c, bool enable)**

  Enable/Disable the DMA channel in a channel configuration object.

- **static void channel_config_set_sniff_enable (dma_channel_config *c, bool sniff_enable)**

  Enable access to channel by sniff hardware in a channel configuration object.

- **static dma_channel_config dma_channel_get_default_config (uint channel)**

  Get the default channel configuration for a given channel.

- **static dma_channel_config dma_get_channel_config (uint channel)**

  Get the current configuration for the specified channel.

- **static uint32_t channel_config_get_ctrl_value (const dma_channel_config *config)**

  Get the raw configuration register from a channel configuration.

Detailed Description

DMA channel configuration.

A DMA channel needs to be configured, these functions provide handy helpers to set up configuration structures. See dma_channel_config

Function Documentation

◆ channel_config_get_ctrl_value()

static uint32_t channel_config_get_ctrl_value (const dma_channel_config * ***config***) inline static

Get the raw configuration register from a channel configuration.

Parameters

config Pointer to a config structure.

Returns

Register content

◆ channel_config_set_bswap()

static void channel_config_set_bswap (dma_channel_config * ***c***,
bool ***bswap***
) inline static

Set DMA byte swapping config in a channel configuration object.

No effect for byte data, for halfword data, the two bytes of each halfword are swapped. For word data, the four bytes of each word are swapped to reverse their order.

Parameters

c Pointer to channel configuration object
bswap True to enable byte swapping

◆ channel_config_set_chain_to()

static void channel_config_set_chain_to (dma_channel_config * ***c***,
uint ***chain_to***
) inline static

Set DMA channel chain_to channel in a channel configuration object.

When this channel completes, it will trigger the channel indicated by chain_to. Disable by setting chain_to to itself (the same channel)

Parameters

c	Pointer to channel configuration object
chain_to	Channel to trigger when this channel completes.

◆ channel_config_set_dreq()

```
static void channel_config_set_dreq ( dma_channel_config * c,
                                      uint                 dreq
                                    )                                  inline  static
```

Select a transfer request signal in a channel configuration object.

The channel uses the transfer request signal to pace its data transfer rate. Sources for TREQ signals are internal (TIMERS) or external (DREQ, a Data Request from the system). 0x0 to 0x3a -> select DREQ n as TREQ 0x3b -> Select Timer 0 as TREQ 0x3c -> Select Timer 1 as TREQ 0x3d -> Select Timer 2 as TREQ (Optional) 0x3e -> Select Timer 3 as TREQ (Optional) 0x3f -> Permanent request, for unpaced transfers.

Parameters

c	Pointer to channel configuration data
dreq	Source (see description)

◆ channel_config_set_enable()

```
static void channel_config_set_enable ( dma_channel_config * c,
                                        bool                 enable
                                      )                                inline  static
```

Enable/Disable the DMA channel in a channel configuration object.

When false, the channel will ignore triggers, stop issuing transfers, and pause the current transfer sequence (i.e. BUSY will remain high if already high)

Parameters

c	Pointer to channel configuration object
enable	True to enable the DMA channel. When enabled, the channel will respond to triggering events, and start transferring data.

◆ channel_config_set_high_priority()

```
static void channel_config_set_high_priority ( dma_channel_config * c,
                                               bool                 high_priority
                                             )                           inline  static
```

Set the channel priority in a channel configuration object.

When true, gives a channel preferential treatment in issue scheduling: in each scheduling round, all high priority channels are considered first, and then only a single low priority channel, before returning to the high priority channels.

This only affects the order in which the DMA schedules channels. The DMA's bus priority is not changed. If the DMA is not saturated then a low priority channel will see no loss of throughput.

Parameters

c	Pointer to channel configuration object
high_priority	True to enable high priority

◆ **channel_config_set_irq_quiet()**

```
static void channel_config_set_irq_quiet ( dma_channel_config * c,
                                           bool                 irq_quiet
                                         )                                  inline  static
```

Set IRQ quiet mode in a channel configuration object.

In QUIET mode, the channel does not generate IRQs at the end of every transfer block. Instead, an IRQ is raised when NULL is written to a trigger register, indicating the end of a control block chain.

Parameters

c	Pointer to channel configuration object
irq_quiet	True to enable quiet mode, false to disable.

◆ **channel_config_set_read_increment()**

```
static void channel_config_set_read_increment ( dma_channel_config * c,
                                                bool                 incr
                                              )                             inline  static
```

Set DMA channel read increment in a channel configuration object.

Parameters

c	Pointer to channel configuration object
incr	True to enable read address increments, if false, each read will be from the same address Usually disabled for peripheral to memory transfers

◆ **channel_config_set_ring()**

```
static void channel_config_set_ring ( dma_channel_config * c,
                                      bool                 write,
                                      uint                 size_bits
                                    )                                       inline  static
```

Set address wrapping parameters in a channel configuration object.

Size of address wrap region. If 0, don't wrap. For values n > 0, only the lower n bits of the address will change. This wraps the address on a (1 << n) byte boundary, facilitating access to naturally-aligned ring buffers. Ring sizes between 2 and 32768 bytes are possible (size_bits from 1 - 15)

0x0 -> No wrapping.

Parameters

c	Pointer to channel configuration object
write	True to apply to write addresses, false to apply to read addresses
size_bits	0 to disable wrapping. Otherwise the size in bits of the changing part of the address. Effectively wraps the address on a (1 << size_bits) byte boundary.

◆ **channel_config_set_sniff_enable()**

```
static void channel_config_set_sniff_enable ( dma_channel_config * c,
                                              bool                 sniff_enable
```

)

inline static

Enable access to channel by sniff hardware in a channel configuration object.

Sniff HW must be enabled and have this channel selected.

Parameters

c	Pointer to channel configuration object
sniff_enable	True to enable the Sniff HW access to this DMA channel.

◆ channel_config_set_transfer_data_size()

static void channel_config_set_transfer_data_size (dma_channel_config * ***c***,
enum dma_channel_transfer_size ***size***
)

inline static

Set the size of each DMA bus transfer in a channel configuration object.

Set the size of each bus transfer (byte/halfword/word). The read and write addresses advance by the specific amount (1/2/4 bytes) with each transfer.

Parameters

c	Pointer to channel configuration object
size	See enum for possible values.

◆ channel_config_set_write_increment()

static void channel_config_set_write_increment (dma_channel_config * ***c***,
bool ***incr***
)

inline static

Set DMA channel write increment in a channel configuration object.

Parameters

c	Pointer to channel configuration object
incr	True to enable write address increments, if false, each write will be to the same address Usually disabled for memory to peripheral transfers

◆ dma_channel_get_default_config()

static dma_channel_config dma_channel_get_default_config (uint ***channel***)

inline static

Get the default channel configuration for a given channel.

Setting	Default
Read Increment	true
Write Increment	false
DReq	DREQ_FORCE
Chain to	self
Data size	DMA_SIZE_32
Ring	write=false, size=0 (i.e. off)
Byte Swap	false
Quiet IRQs	false
High Priority	false

Channel Enable	true
Sniff Enable	false

Parameters

channel DMA channel

Returns

the default configuration which can then be modified.

◆ dma_get_channel_config()

static dma_channel_config dma_get_channel_config (uint ***channel***) inline static

Get the current configuration for the specified channel.

Parameters

channel DMA channel

Returns

The current configuration as read from the HW register (not cached)

hardware_exception

Part of: Hardware APIs

Typedefs

- **typedef void(* exception_handler_t) (void)**

  Exception handler function type.

Enumerations

- **enum exception_number {**
  NMI_EXCEPTION = -14 , HARDFAULT_EXCEPTION = -13 , SVCALL_EXCEPTION = -5 , PENDSV_EXCEPTION = -2 ,
  SYSTICK_EXCEPTION = -1
  }

  Exception number definitions. More...

Functions

- **exception_handler_t exception_set_exclusive_handler (enum exception_number num, exception_handler_t handler)**

  Set the exception handler for an exception on the executing core.

- **void exception_restore_handler (enum exception_number num, exception_handler_t original_handler)**

  Restore the original exception handler for an exception on this core.

- **exception_handler_t exception_get_vtable_handler (enum exception_number num)**

Get the current exception handler for the specified exception from the currently installed vector table of the execution core.

Detailed Description

Methods for setting processor exception handlers

Exceptions are identified by a exception_number which is a number from -15 to -1; these are the numbers relative to the index of the first IRQ vector in the vector table. (i.e. vector table index is exception_num plus 16)

There is one set of exception handlers per core, so the exception handlers for each core as set by these methods are independent.

NOTE

That all exception APIs affect the executing core only (i.e. the core calling the function).

Typedef Documentation

◆ exception_handler_t

typedef void(* exception_handler_t) (void)

Exception handler function type.

All exception handlers should be of this type, and follow normal ARM EABI register saving conventions

Enumeration Type Documentation

◆ exception_number

enum exception_number

Exception number definitions.

Note for consistency with irq numbers, these numbers are defined to be negative. The VTABLE index is the number here plus 16.

Name	Value	Exception
NMI_EXCEPTION	-14	Non Maskable Interrupt
HARDFAULT_EXCEPTION	-13	HardFault
SVCALL_EXCEPTION	-5	SV Call
PENDSV_EXCEPTION	-2	Pend SV
SYSTICK_EXCEPTION	-1	System Tick

Function Documentation

◆ exception_get_vtable_handler()

exception_handler_t exception_get_vtable_handler (enum exception_number ***num***)

Get the current exception handler for the specified exception from the currently installed vector table of the execution core.

Parameters

num Exception number

Returns

the address stored in the VTABLE for the given exception number

◆ **exception_restore_handler()**

void exception_restore_handler (enum exception_number ***num***,
exception_handler_t ***original_handler***
)

Restore the original exception handler for an exception on this core.

This method may be used to restore the exception handler for an exception on this core to the state prior to the call to exception_set_exclusive_handler(), so that exception_set_exclusive_handler() may be called again in the future.

Parameters

num	Exception number exception_number
original_handler	The original handler returned from exception_set_exclusive_handler

See also exception_set_exclusive_handler()

◆ **exception_set_exclusive_handler()**

exception_handler_t exception_set_exclusive_handler (enum exception_number ***num***,
exception_handler_t ***handler***
)

Set the exception handler for an exception on the executing core.

This method will assert if an exception handler has been set for this exception number on this core via this method, without an intervening restore via exception_restore_handler.

NOTE

this method may not be used to override an exception handler that was specified at link time by providing a strong replacement for the weakly defined stub exception handlers. It will assert in this case too.

Parameters

num	Exception number
handler	The handler to set

See also exception_number

hardware_flash

Part of: Hardware APIs

Functions

- **`void flash_range_erase (uint32_t flash_offs, size_t count)`**

  Erase areas of flash.

- **void flash_range_program (uint32_t flash_offs, const uint8_t *data, size_t count)**

  Program flash.

- **void flash_get_unique_id (uint8_t *id_out)**

  Get flash unique 64 bit identifier.

- **void flash_do_cmd (const uint8_t *txbuf, uint8_t *rxbuf, size_t count)**

  Execute bidirectional flash command.

Detailed Description

Low level flash programming and erase API

Note these functions are *unsafe* if you are using both cores, and the other is executing from flash concurrently with the operation. In this could be the case, you must perform your own synchronisation to make sure that no XIP accesses take place during flash programming. One option is to use the lockout functions.

Likewise they are *unsafe* if you have interrupt handlers or an interrupt vector table in flash, so you must disable interrupts before calling in this case.

If PICO_NO_FLASH=1 is not defined (i.e. if the program is built to run from flash) then these functions will make a static copy of the second stage bootloader in SRAM, and use this to reenter execute-in-place mode after programming or erasing flash, so that they can safely be called from flash-resident code.

Example

```
#include <stdio.h>
#include <stdlib.h>

#include "pico/stdlib.h"
#include "hardware/flash.h"

// We're going to erase and reprogram a region 256k from the start of flash.
// Once done, we can access this at XIP_BASE + 256k.
#define FLASH_TARGET_OFFSET (256 * 1024)


const uint8_t *flash_target_contents = (const uint8_t *) (XIP_BASE + FLASH_TARGET_OFFSET);


void print_buf(const uint8_t *buf, size_t len) {
    for (size_t i = 0; i < len; ++i) {
        printf("%02x", buf[i]);
        if (i % 16 == 15)
            printf("\n");
        else
            printf(" ");
    }
}


int main() {
    stdio_init_all();
    uint8_t random_data[FLASH_PAGE_SIZE];
    for (int i = 0; i < FLASH_PAGE_SIZE; ++i)
        random_data[i] = rand() >> 16;

    printf("Generated random data:\n");
    print_buf(random_data, FLASH_PAGE_SIZE);

    // Note that a whole number of sectors must be erased at a time.
    printf("\nErasing target region...\n");
    flash_range_erase(FLASH_TARGET_OFFSET, FLASH_SECTOR_SIZE);
    printf("Done. Read back target region:\n");
    print_buf(flash_target_contents, FLASH_PAGE_SIZE);

    printf("\nProgramming target region...\n");
    flash_range_program(FLASH_TARGET_OFFSET, random_data, FLASH_PAGE_SIZE);
    printf("Done. Read back target region:\n");
    print_buf(flash_target_contents, FLASH_PAGE_SIZE);

    bool mismatch = false;
    for (int i = 0; i < FLASH_PAGE_SIZE; ++i) {
        if (random_data[i] != flash_target_contents[i])
            mismatch = true;
    }
    if (mismatch)
        printf("Programming failed!\n");
```

```
    else
        printf("Programming successful!\n");
}
```

Function Documentation

◆ flash_do_cmd()

```
void flash_do_cmd ( const uint8_t *  txbuf,
                    uint8_t *        rxbuf,
                    size_t           count
                  )
```

Execute bidirectional flash command.

Low-level function to execute a serial command on a flash device attached to the QSPI interface. Bytes are simultaneously transmitted and received from txbuf and to rxbuf. Therefore, both buffers must be the same length, count, which is the length of the overall transaction. This is useful for reading metadata from the flash chip, such as device ID or SFDP parameters.

The XIP cache is flushed following each command, in case flash state has been modified. Like other hardware_flash functions, the flash is not accessible for execute-in-place transfers whilst the command is in progress, so entering a flash-resident interrupt handler or executing flash code on the second core concurrently will be fatal. To avoid these pitfalls it is recommended that this function only be used to extract flash metadata during startup, before the main application begins to run: see the implementation of pico_get_unique_id() for an example of this.

Parameters

txbuf	Pointer to a byte buffer which will be transmitted to the flash
rxbuf	Pointer to a byte buffer where data received from the flash will be written. txbuf and rxbuf may be the same buffer.
count	Length in bytes of txbuf and of rxbuf

◆ flash_get_unique_id()

```
void flash_get_unique_id ( uint8_t *  id_out )
```

Get flash unique 64 bit identifier.

Use a standard 4Bh RUID instruction to retrieve the 64 bit unique identifier from a flash device attached to the QSPI interface. Since there is a 1:1 association between the MCU and this flash, this also serves as a unique identifier for the board.

Parameters

id_out	Pointer to an 8-byte buffer to which the ID will be written

◆ flash_range_erase()

```
void flash_range_erase ( uint32_t  flash_offs,
                         size_t    count
                       )
```

Erase areas of flash.

Parameters

flash_offs	Offset into flash, in bytes, to start the erase. Must be aligned to a 4096-byte flash sector.

count	Number of bytes to be erased. Must be a multiple of 4096 bytes (one sector).

◆ **flash_range_program()**

```
void flash_range_program ( uint32_t          flash_offs,
                           const uint8_t *   data,
                           size_t            count
                         )
```

Program flash.

Parameters

flash_offs	Flash address of the first byte to be programmed. Must be aligned to a 256-byte flash page.
data	Pointer to the data to program into flash
count	Number of bytes to program. Must be a multiple of 256 bytes (one page).

hardware_gpio

Part of: Hardware APIs

Typedefs

- **typedef void(* gpio_irq_callback_t) (uint gpio, uint32_t event_mask)**

Enumerations

- **enum gpio_function {**
  GPIO_FUNC_XIP = 0 , GPIO_FUNC_SPI = 1 , GPIO_FUNC_UART = 2 , GPIO_FUNC_I2C = 3 ,
  GPIO_FUNC_PWM = 4 , GPIO_FUNC_SIO = 5 , GPIO_FUNC_PIO0 = 6 , GPIO_FUNC_PIO1 = 7 ,
  GPIO_FUNC_GPCK = 8 , GPIO_FUNC_USB = 9 , GPIO_FUNC_NULL = 0x1f
  }

  GPIO function definitions for use with function select. More...

- **enum gpio_irq_level { GPIO_IRQ_LEVEL_LOW = 0x1u , GPIO_IRQ_LEVEL_HIGH = 0x2u , GPIO_IRQ_EDGE_FALL = 0x4u , GPIO_IRQ_EDGE_RISE = 0x8u }**

  GPIO Interrupt level definitions (GPIO events) More...

- **enum gpio_slew_rate { GPIO_SLEW_RATE_SLOW = 0 , GPIO_SLEW_RATE_FAST = 1 }**

  Slew rate limiting levels for GPIO outputs. More...

- **enum gpio_drive_strength { GPIO_DRIVE_STRENGTH_2MA = 0 , GPIO_DRIVE_STRENGTH_4MA = 1 , GPIO_DRIVE_STRENGTH_8MA = 2 , GPIO_DRIVE_STRENGTH_12MA = 3 }**

  Drive strength levels for GPIO outputs. More...

Functions

- **void gpio_set_function (uint gpio, enum gpio_function fn)**

  Select GPIO function.

- **enum gpio_function gpio_get_function (uint gpio)**

  Determine current GPIO function.

- **void gpio_set_pulls (uint gpio, bool up, bool down)**

  Select up and down pulls on specific GPIO.

- **static void gpio_pull_up (uint gpio)**

  Set specified GPIO to be pulled up.

- **static bool gpio_is_pulled_up (uint gpio)**

  Determine if the specified GPIO is pulled up.

- **static void gpio_pull_down (uint gpio)**

  Set specified GPIO to be pulled down.

- **static bool gpio_is_pulled_down (uint gpio)**

  Determine if the specified GPIO is pulled down.

- **static void gpio_disable_pulls (uint gpio)**

  Disable pulls on specified GPIO.

- **void gpio_set_irqover (uint gpio, uint value)**

  Set GPIO IRQ override.

- **void gpio_set_outover (uint gpio, uint value)**

  Set GPIO output override.

- **void gpio_set_inover (uint gpio, uint value)**

  Select GPIO input override.

- **void gpio_set_oeover (uint gpio, uint value)**

  Select GPIO output enable override.

- **void gpio_set_input_enabled (uint gpio, bool enabled)**

  Enable GPIO input.

- **void gpio_set_input_hysteresis_enabled (uint gpio, bool enabled)**

  Enable/disable GPIO input hysteresis (Schmitt trigger)

- **bool gpio_is_input_hysteresis_enabled (uint gpio)**

  Determine whether input hysteresis is enabled on a specified GPIO.

- **void gpio_set_slew_rate (uint gpio, enum gpio_slew_rate slew)**

  Set slew rate for a specified GPIO.

- **enum gpio_slew_rate gpio_get_slew_rate (uint gpio)**

  Determine current slew rate for a specified GPIO.

- **void gpio_set_drive_strength (uint gpio, enum gpio_drive_strength drive)**

  Set drive strength for a specified GPIO.

- **enum gpio_drive_strength gpio_get_drive_strength (uint gpio)**

  Determine current slew rate for a specified GPIO.

- **void gpio_set_irq_enabled (uint gpio, uint32_t event_mask, bool enabled)**

Enable or disable specific interrupt events for specified GPIO.

- **void gpio_set_irq_callback (gpio_irq_callback_t callback)**

  Set the generic callback used for GPIO IRQ events for the current core.

- **void gpio_set_irq_enabled_with_callback (uint gpio, uint32_t event_mask, bool enabled, gpio_irq_callback_t callback)**

  Convenience function which performs multiple GPIO IRQ related initializations.

- **void gpio_set_dormant_irq_enabled (uint gpio, uint32_t event_mask, bool enabled)**

  Enable dormant wake up interrupt for specified GPIO and events.

- **static uint32_t gpio_get_irq_event_mask (uint gpio)**

  Return the current interrupt status (pending events) for the given GPIO.

- **void gpio_acknowledge_irq (uint gpio, uint32_t event_mask)**

  Acknowledge a GPIO interrupt for the specified events on the calling core.

- **void gpio_add_raw_irq_handler_with_order_priority_masked (uint gpio_mask, irq_handler_t handler, uint8_t order_priority)**

  Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

- **static void gpio_add_raw_irq_handler_with_order_priority (uint gpio, irq_handler_t handler, uint8_t order_priority)**

  Adds a raw GPIO IRQ handler for a specific GPIO on the current core.

- **void gpio_add_raw_irq_handler_masked (uint gpio_mask, irq_handler_t handler)**

  Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

- **static void gpio_add_raw_irq_handler (uint gpio, irq_handler_t handler)**

  Adds a raw GPIO IRQ handler for a specific GPIO on the current core.

- **void gpio_remove_raw_irq_handler_masked (uint gpio_mask, irq_handler_t handler)**

  Removes a raw GPIO IRQ handler for the specified GPIOs on the current core.

- **static void gpio_remove_raw_irq_handler (uint gpio, irq_handler_t handler)**

  Removes a raw GPIO IRQ handler for the specified GPIO on the current core.

- **void gpio_init (uint gpio)**

  Initialise a GPIO for (enabled I/O and set func to GPIO_FUNC_SIO)

- **void gpio_deinit (uint gpio)**

  Resets a GPIO back to the NULL function, i.e. disables it.

- **void gpio_init_mask (uint gpio_mask)**

  Initialise multiple GPIOs (enabled I/O and set func to GPIO_FUNC_SIO)

- **static bool gpio_get (uint gpio)**

  Get state of a single specified GPIO.

- **static uint32_t gpio_get_all (void)**

  Get raw value of all GPIOs.

- **static void gpio_set_mask (uint32_t mask)**

  Drive high every GPIO appearing in mask.

- **static void gpio_clr_mask (uint32_t mask)**

  Drive low every GPIO appearing in mask.

- **static void gpio_xor_mask (uint32_t mask)**

  Toggle every GPIO appearing in mask.

- **static void gpio_put_masked (uint32_t mask, uint32_t value)**

  Drive GPIO high/low depending on parameters.

- **static void gpio_put_all (uint32_t value)**

  Drive all pins simultaneously.

- **static void gpio_put (uint gpio, bool value)**

  Drive a single GPIO high/low.

- **static bool gpio_get_out_level (uint gpio)**

  Determine whether a GPIO is currently driven high or low.

- **static void gpio_set_dir_out_masked (uint32_t mask)**

  Set a number of GPIOs to output.

- **static void gpio_set_dir_in_masked (uint32_t mask)**

  Set a number of GPIOs to input.

- **static void gpio_set_dir_masked (uint32_t mask, uint32_t value)**

  Set multiple GPIO directions.

- **static void gpio_set_dir_all_bits (uint32_t values)**

  Set direction of all pins simultaneously.

- **static void gpio_set_dir (uint gpio, bool out)**

  Set a single GPIO direction.

- **static bool gpio_is_dir_out (uint gpio)**

  Check if a specific GPIO direction is OUT.

- **static uint gpio_get_dir (uint gpio)**

  Get a specific GPIO direction.

Detailed Description

General Purpose Input/Output (GPIO) API

RP2040 has 36 multi-functional General Purpose Input / Output (GPIO) pins, divided into two banks. In a typical use case, the pins in the QSPI bank (QSPI_SS, QSPI_SCLK and QSPI_SD0 to QSPI_SD3) are used to execute code from an external flash device, leaving the User bank (GPIO0 to GPIO29) for the programmer to use. All GPIOs support digital input and output, but GPIO26 to GPIO29 can also be used as inputs to the chip's Analogue to Digital Converter (ADC). Each GPIO can be controlled directly by software running on the processors, or by a number of other functional blocks.

The function allocated to each GPIO is selected by calling the gpio_set_function function.

NOTE

Not all functions are available on all pins.

Each GPIO can have one function selected at a time. Likewise, each peripheral input (e.g. UART0 RX) should only be selected on one *GPIO* at a time. If the same peripheral input is connected to multiple GPIOs, the peripheral sees the logical OR of these GPIO inputs. Please refer to the datasheet for more information on GPIO function select.

Function Select Table

GPIO	F1	F2	F3	F4	F5	F6	F7	F8	F9
0	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1		USB OVCUR DET
1	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1		USB VBUS DET
2	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1		USB VBUS EN
3	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1		USB OVCUR DET
4	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1		USB VBUS DET
5	SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1		USB VBUS EN
6	SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1		USB OVCUR DET
7	SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PIO0	PIO1		USB VBUS DET
8	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1		USB VBUS EN
9	SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PIO0	PIO1		USB OVCUR DET
10	SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PIO0	PIO1		USB VBUS DET
11	SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PIO0	PIO1		USB VBUS EN
12	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB OVCUR DET
13	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PIO0	PIO1		USB VBUS DET
14	SPI1 SCK	UART0 CTS	I2C1 SDA	PWM7 A	SIO	PIO0	PIO1		USB VBUS EN
15	SPI1 TX	UART0 RTS	I2C1 SCL	PWM7 B	SIO	PIO0	PIO1		USB OVCUR DET
16	SPI0 RX	UART0 TX	I2C0 SDA	PWM0 A	SIO	PIO0	PIO1		USB VBUS DET
17	SPI0 CSn	UART0 RX	I2C0 SCL	PWM0 B	SIO	PIO0	PIO1		USB VBUS EN
18	SPI0 SCK	UART0 CTS	I2C1 SDA	PWM1 A	SIO	PIO0	PIO1		USB OVCUR DET
19	SPI0 TX	UART0 RTS	I2C1 SCL	PWM1 B	SIO	PIO0	PIO1		USB VBUS DET
20	SPI0 RX	UART1 TX	I2C0 SDA	PWM2 A	SIO	PIO0	PIO1	CLOCK GPIN0	USB VBUS EN
21	SPI0 CSn	UART1 RX	I2C0 SCL	PWM2 B	SIO	PIO0	PIO1	CLOCK GPOUT0	USB OVCUR DET
22	SPI0 SCK	UART1 CTS	I2C1 SDA	PWM3 A	SIO	PIO0	PIO1	CLOCK GPIN1	USB VBUS DET
23	SPI0 TX	UART1 RTS	I2C1 SCL	PWM3 B	SIO	PIO0	PIO1	CLOCK GPOUT1	USB VBUS EN
24	SPI1 RX	UART1 TX	I2C0 SDA	PWM4 A	SIO	PIO0	PIO1	CLOCK GPOUT2	USB OVCUR DET
25	SPI1 CSn	UART1 RX	I2C0 SCL	PWM4 B	SIO	PIO0	PIO1	CLOCK GPOUT3	USB VBUS DET
26	SPI1 SCK	UART1 CTS	I2C1 SDA	PWM5 A	SIO	PIO0	PIO1		USB VBUS EN
27	SPI1 TX	UART1 RTS	I2C1 SCL	PWM5 B	SIO	PIO0	PIO1		USB OVCUR DET
28	SPI1 RX	UART0 TX	I2C0 SDA	PWM6 A	SIO	PIO0	PIO1		USB VBUS DET
29	SPI1 CSn	UART0 RX	I2C0 SCL	PWM6 B	SIO	PIO0	PIO1		USB VBUS EN

Typedef Documentation

◆ gpio_irq_callback_t

typedef void(* gpio_irq_callback_t) (uint gpio, uint32_t event_mask)

Callback function type for GPIO events

Parameters

gpio Which GPIO caused this interrupt

event_mask Which events caused this interrupt. See gpio_irq_level for details.

See also gpio_set_irq_enabled_with_callback() gpio_set_irq_callback()

Enumeration Type Documentation

◆ gpio_drive_strength

enum gpio_drive_strength

Drive strength levels for GPIO outputs.

Drive strength levels for GPIO outputs.

See also gpio_set_drive_strength

Enumerator	
GPIO_DRIVE_STRENGTH_2MA	2 mA nominal drive strength
GPIO_DRIVE_STRENGTH_4MA	4 mA nominal drive strength
GPIO_DRIVE_STRENGTH_8MA	8 mA nominal drive strength
GPIO_DRIVE_STRENGTH_12MA	12 mA nominal drive strength

◆ gpio_function

enum gpio_function

GPIO function definitions for use with function select.

GPIO function selectors

Each GPIO can have one function selected at a time. Likewise, each peripheral input (e.g. UART0 RX) should only be selected on one GPIO at a time. If the same peripheral input is connected to multiple GPIOs, the peripheral sees the logical OR of these GPIO inputs.

Please refer to the datasheet for more information on GPIO function selection.

◆ gpio_irq_level

enum gpio_irq_level

GPIO Interrupt level definitions (GPIO events)

GPIO Interrupt levels

An interrupt can be generated for every GPIO pin in 4 scenarios:

- Level High: the GPIO pin is a logical 1
- Level Low: the GPIO pin is a logical 0
- Edge High: the GPIO has transitioned from a logical 0 to a logical 1
- Edge Low: the GPIO has transitioned from a logical 1 to a logical 0

The level interrupts are not latched. This means that if the pin is a logical 1 and the level high interrupt is active, it will become inactive as soon as the pin changes to a logical 0. The edge interrupts are stored in the INTR register and can be cleared by writing to the INTR register.

◆ gpio_slew_rate

enum gpio_slew_rate

Slew rate limiting levels for GPIO outputs.

Slew rate limiting increases the minimum rise/fall time when a GPIO output is lightly loaded, which can help to reduce electromagnetic emissions.

See also gpio_set_slew_rate

Enumerator	
GPIO_SLEW_RATE_SLOW	Slew rate limiting enabled.
GPIO_SLEW_RATE_FAST	Slew rate limiting disabled.

Function Documentation

◆ gpio_acknowledge_irq()

```
void gpio_acknowledge_irq ( uint     gpio,
                            uint32_t event_mask
                          )
```

Acknowledge a GPIO interrupt for the specified events on the calling core.

NOTE

This may be called with a mask of any of valid bits specified in gpio_irq_level, however it has no effect on *level* sensitive interrupts which remain pending while the GPIO is at the specified level. When handling *level* sensitive interrupts, you should generally disable the interrupt (see gpio_set_irq_enabled) and then set it up again later once the GPIO level has changed (or to catch the opposite level).

Parameters

gpio GPIO number

NOTE

For callbacks set with gpio_set_irq_enabled_with_callback, or gpio_set_irq_callback, this function is called automatically.

Parameters

event_mask Bitmask of events to clear. See gpio_irq_level for details.

◆ gpio_add_raw_irq_handler()

```
static void gpio_add_raw_irq_handler ( uint          gpio,
                                       irq_handler_t handler
                                     )                                  inline static
```

Adds a raw GPIO IRQ handler for a specific GPIO on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method adds such a callback, and disables the "default" callback for the specified GPIO.

NOTE

Multiple raw handlers should not be added for the same GPIO, and this method will assert if you attempt to.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```
void my_irq_handler(void) {
    if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
       gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
      // handle the IRQ
    }
}
```

Parameters

gpio	the GPIO number that will no longer be passed to the default callback for this core
handler	the handler to add to the list of GPIO IRQ handlers for this core

◆ gpio_add_raw_irq_handler_masked()

void gpio_add_raw_irq_handler_masked (uint ***gpio_mask***,
irq_handler_t ***handler***
)

Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method adds such a callback, and disables the "default" callback for the specified GPIOs.

NOTE

Multiple raw handlers should not be added for the same GPIOs, and this method will assert if you attempt to.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```
void my_irq_handler(void) {
    if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
       gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
      // handle the IRQ
    }
    if (gpio_get_irq_event_mask(my_gpio_num2) & my_gpio_event_mask2) {
       gpio_acknowledge_irq(my_gpio_num2, my_gpio_event_mask2);
      // handle the IRQ
    }
}
```

Parameters

gpio_mask	a bit mask of the GPIO numbers that will no longer be passed to the default callback for this core
handler	the handler to add to the list of GPIO IRQ handlers for this core

◆ gpio_add_raw_irq_handler_with_order_priority()

static void gpio_add_raw_irq_handler_with_order_priority (uint ***gpio***,
irq_handler_t ***handler***,
uint8_t ***order_priority***
)

inline static

Adds a raw GPIO IRQ handler for a specific GPIO on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default callback. The order relative to the default callback can be controlled via the order_priority parameter(the default callback has the priority GPIO_IRQ_CALLBACK_ORDER_PRIORITY which defaults to the lowest priority with the intention of it running last).

This method adds such a callback, and disables the "default" callback for the specified GPIO.

NOTE

Multiple raw handlers should not be added for the same GPIO, and this method will assert if you attempt to.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```
void my_irq_handler(void) {
    if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
       gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
      // handle the IRQ
    }
}
```

Parameters

gpio	the GPIO number that will no longer be passed to the default callback for this core
handler	the handler to add to the list of GPIO IRQ handlers for this core
order_priority	the priority order to determine the relative position of the handler in the list of GPIO IRQ handlers for this core.

◆ gpio_add_raw_irq_handler_with_order_priority_masked()

```
void gpio_add_raw_irq_handler_with_order_priority_masked ( uint           gpio_mask,
                                                           irq_handler_t  handler,
                                                           uint8_t        order_priority
                                                         )
```

Adds a raw GPIO IRQ handler for the specified GPIOs on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default callback. The order relative to the default callback can be controlled via the order_priority parameter (the default callback has the priority GPIO_IRQ_CALLBACK_ORDER_PRIORITY which defaults to the lowest priority with the intention of it running last).

This method adds such an explicit GPIO IRQ handler, and disables the "default" callback for the specified GPIOs.

NOTE

Multiple raw handlers should not be added for the same GPIOs, and this method will assert if you attempt to.

A raw handler should check for whichever GPIOs and events it handles, and acknowledge them itself; it might look something like:

```
void my_irq_handler(void) {
    if (gpio_get_irq_event_mask(my_gpio_num) & my_gpio_event_mask) {
       gpio_acknowledge_irq(my_gpio_num, my_gpio_event_mask);
      // handle the IRQ
    }
    if (gpio_get_irq_event_mask(my_gpio_num2) & my_gpio_event_mask2) {
       gpio_acknowledge_irq(my_gpio_num2, my_gpio_event_mask2);
      // handle the IRQ
    }
}
```

Parameters

gpio_mask	a bit mask of the GPIO numbers that will no longer be passed to the default callback for this core
handler	the handler to add to the list of GPIO IRQ handlers for this core

order_priority the priority order to determine the relative position of the handler in the list of GPIO IRQ handlers for this core.

◆ gpio_clr_mask()

static void gpio_clr_mask (uint32_t ***mask***) inline static

Drive low every GPIO appearing in mask.

Parameters

mask Bitmask of GPIO values to clear, as bits 0-29

◆ gpio_deinit()

void gpio_deinit (uint ***gpio***)

Resets a GPIO back to the NULL function, i.e. disables it.

Parameters

gpio GPIO number

◆ gpio_disable_pulls()

static void gpio_disable_pulls (uint ***gpio***) inline static

Disable pulls on specified GPIO.

Parameters

gpio GPIO number

◆ gpio_get()

static bool gpio_get (uint ***gpio***) inline static

Get state of a single specified GPIO.

Parameters

gpio GPIO number

Returns

Current state of the GPIO. 0 for low, non-zero for high

◆ gpio_get_all()

static uint32_t gpio_get_all (void) inline static

Get raw value of all GPIOs.

Returns

Bitmask of raw GPIO values, as bits 0-29

◆ gpio_get_dir()

static uint gpio_get_dir (uint ***gpio***) inline static

Get a specific GPIO direction.

Parameters

gpio GPIO number

Returns

1 for out, 0 for in

◆ gpio_get_drive_strength()

enum gpio_drive_strength gpio_get_drive_strength (uint ***gpio***)

Determine current slew rate for a specified GPIO.

See also gpio_set_drive_strength

Parameters

gpio GPIO number

Returns

Current drive strength of that GPIO

◆ gpio_get_function()

enum gpio_function gpio_get_function (uint ***gpio***)

Determine current GPIO function.

Parameters

gpio GPIO number

Returns

Which GPIO function is currently selected from list gpio_function

◆ gpio_get_irq_event_mask()

static uint32_t gpio_get_irq_event_mask (uint ***gpio***) inline static

Return the current interrupt status (pending events) for the given GPIO.

Parameters

gpio GPIO number

Returns

Bitmask of events that are currently pending for the GPIO. See gpio_irq_level for details.

See also gpio_acknowledge_irq

◆ gpio_get_out_level()

static bool gpio_get_out_level (uint ***gpio***) inline static

Determine whether a GPIO is currently driven high or low.

This function returns the high/low output level most recently assigned to a GPIO via gpio_put() or similar. This is the value that is presented outward to the IO muxing, *not* the input level back from the pad (which can be read using gpio_get()).

To avoid races, this function must not be used for read-modify-write sequences when driving GPIOs – instead functions like gpio_put() should be used to atomically update GPIOs. This accessor is intended for debug use only.

Parameters

gpio GPIO number

Returns

true if the GPIO output level is high, false if low.

◆ gpio_get_slew_rate()

enum gpio_slew_rate gpio_get_slew_rate (uint ***gpio***)

Determine current slew rate for a specified GPIO.

See also gpio_set_slew_rate

Parameters

gpio GPIO number

Returns

Current slew rate of that GPIO

◆ gpio_init()

void gpio_init (uint ***gpio***)

Initialise a GPIO for (enabled I/O and set func to GPIO_FUNC_SIO)

Clear the output enable (i.e. set to input). Clear any output value.

Parameters

gpio GPIO number

◆ gpio_init_mask()

void gpio_init_mask (uint ***gpio_mask***)

Initialise multiple GPIOs (enabled I/O and set func to GPIO_FUNC_SIO)

Clear the output enable (i.e. set to input). Clear any output value.

Parameters

gpio_mask Mask with 1 bit per GPIO number to initialize

◆ gpio_is_dir_out()

static bool gpio_is_dir_out (uint ***gpio***) inline static

Check if a specific GPIO direction is OUT.

Parameters

gpio GPIO number

Returns

true if the direction for the pin is OUT

◆ gpio_is_input_hysteresis_enabled()

bool gpio_is_input_hysteresis_enabled (uint ***gpio***)

Determine whether input hysteresis is enabled on a specified GPIO.

See also gpio_set_input_hysteresis_enabled

Parameters

gpio GPIO number

◆ gpio_is_pulled_down()

static bool gpio_is_pulled_down (uint ***gpio***) inline static

Determine if the specified GPIO is pulled down.

Parameters

gpio GPIO number

Returns

true if the GPIO is pulled down

◆ gpio_is_pulled_up()

static bool gpio_is_pulled_up (uint ***gpio***) inline static

Determine if the specified GPIO is pulled up.

Parameters

gpio GPIO number

Returns

true if the GPIO is pulled up

◆ gpio_pull_down()

static void gpio_pull_down (uint ***gpio***) inline static

Set specified GPIO to be pulled down.

Parameters

gpio GPIO number

◆ gpio_pull_up()

static void gpio_pull_up (uint ***gpio***) inline static

Set specified GPIO to be pulled up.

Parameters

gpio GPIO number

◆ gpio_put()

static void gpio_put (uint ***gpio,***
bool ***value***
) inline static

Drive a single GPIO high/low.

Parameters

gpio GPIO number
value If false clear the GPIO, otherwise set it.

◆ gpio_put_all()

static void gpio_put_all (uint32_t ***value***) inline static

Drive all pins simultaneously.

Parameters

value Bitmask of GPIO values to change, as bits 0-29

◆ gpio_put_masked()

```
static void gpio_put_masked ( uint32_t mask,
                              uint32_t value
                            )                                    inline static
```

Drive GPIO high/low depending on parameters.

Parameters

mask Bitmask of GPIO values to change, as bits 0-29
value Value to set

For each 1 bit in `mask`, drive that pin to the value given by corresponding bit in `value`, leaving other pins unchanged. Since this uses the TOGL alias, it is concurrency-safe with e.g. an IRQ bashing different pins from the same core.

◆ gpio_remove_raw_irq_handler()

```
static void gpio_remove_raw_irq_handler ( uint           gpio,
                                          irq_handler_t  handler
                                        )                        inline static
```

Removes a raw GPIO IRQ handler for the specified GPIO on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method removes such a callback, and enables the "default" callback for the specified GPIO.

Parameters

gpio the GPIO number that will now be passed to the default callback for this core
handler the handler to remove from the list of GPIO IRQ handlers for this core

◆ gpio_remove_raw_irq_handler_masked()

```
void gpio_remove_raw_irq_handler_masked ( uint           gpio_mask,
                                          irq_handler_t  handler
                                        )
```

Removes a raw GPIO IRQ handler for the specified GPIOs on the current core.

In addition to the default mechanism of a single GPIO IRQ event callback per core (see gpio_set_irq_callback), it is possible to add explicit GPIO IRQ handlers which are called independent of the default event callback.

This method removes such a callback, and enables the "default" callback for the specified GPIOs.

Parameters

gpio_mask a bit mask of the GPIO numbers that will now be passed to the default callback for this core

handler	the handler to remove from the list of GPIO IRQ handlers for this core

◆ gpio_set_dir()

```
static void gpio_set_dir ( uint gpio,
                           bool out
                         )                                    inline static
```

Set a single GPIO direction.

Parameters

gpio	GPIO number
out	true for out, false for in

◆ gpio_set_dir_all_bits()

```
static void gpio_set_dir_all_bits ( uint32_t values )         inline static
```

Set direction of all pins simultaneously.

Parameters

values	individual settings for each gpio; for GPIO N, bit N is 1 for out, 0 for in

◆ gpio_set_dir_in_masked()

```
static void gpio_set_dir_in_masked ( uint32_t mask )          inline static
```

Set a number of GPIOs to input.

Parameters

mask	Bitmask of GPIO to set to input, as bits 0-29

◆ gpio_set_dir_masked()

```
static void gpio_set_dir_masked ( uint32_t mask,
                                  uint32_t value
                                )                              inline static
```

Set multiple GPIO directions.

Parameters

mask	Bitmask of GPIO to set to input, as bits 0-29
value	Values to set

For each 1 bit in "mask", switch that pin to the direction given by corresponding bit in "value", leaving other pins unchanged. E.g. gpio_set_dir_masked(0x3, 0x2); -> set pin 0 to input, pin 1 to output, simultaneously.

◆ gpio_set_dir_out_masked()

static void gpio_set_dir_out_masked (uint32_t ***mask***) inline static

Set a number of GPIOs to output.

Switch all GPIOs in "mask" to output

Parameters

mask Bitmask of GPIO to set to output, as bits 0-29

◆ gpio_set_dormant_irq_enabled()

```
void gpio_set_dormant_irq_enabled ( uint     gpio,
                                    uint32_t event_mask,
                                    bool     enabled
                                  )
```

Enable dormant wake up interrupt for specified GPIO and events.

This configures IRQs to restart the XOSC or ROSC when they are disabled in dormant mode

Parameters

gpio	GPIO number
event_mask	Which events will cause an interrupt. See gpio_irq_level for details.
enabled	Enable/disable flag

◆ gpio_set_drive_strength()

```
void gpio_set_drive_strength ( uint                          gpio,
                               enum gpio_drive_strength drive
                             )
```

Set drive strength for a specified GPIO.

See also gpio_get_drive_strength

Parameters

gpio	GPIO number
drive	GPIO output drive strength

◆ gpio_set_function()

```
void gpio_set_function ( uint                 gpio,
                         enum gpio_function fn
                       )
```

Select GPIO function.

Parameters

gpio	GPIO number
fn	Which GPIO function select to use from list gpio_function

◆ **gpio_set_inover()**

```
void gpio_set_inover ( uint gpio,
                       uint value
                     )
```

Select GPIO input override.

Parameters

gpio	GPIO number
value	See gpio_override

◆ **gpio_set_input_enabled()**

```
void gpio_set_input_enabled ( uint gpio,
                              bool enabled
                            )
```

Enable GPIO input.

Parameters

gpio	GPIO number
enabled	true to enable input on specified GPIO

◆ **gpio_set_input_hysteresis_enabled()**

```
void gpio_set_input_hysteresis_enabled ( uint gpio,
                                         bool enabled
                                       )
```

Enable/disable GPIO input hysteresis (Schmitt trigger)

Enable or disable the Schmitt trigger hysteresis on a given GPIO. This is enabled on all GPIOs by default. Disabling input hysteresis can lead to inconsistent readings when the input signal has very long rise or fall times, but slightly reduces the GPIO's input delay.

See also gpio_is_input_hysteresis_enabled

Parameters

gpio	GPIO number
enabled	true to enable input hysteresis on specified GPIO

◆ **gpio_set_irq_callback()**

```
void gpio_set_irq_callback ( gpio_irq_callback_t callback )
```

Set the generic callback used for GPIO IRQ events for the current core.

This function sets the callback used for all GPIO IRQs on the current core that are not explicitly hooked via gpio_add_raw_irq_handler or other gpio_add_raw_irq_handler_ functions.

This function is called with the GPIO number and event mask for each of the (not explicitly hooked) GPIOs that have events enabled and that are pending (see gpio_get_irq_event_mask).

NOTE

The IO IRQs are independent per-processor. This function affects the processor that calls the function.

Parameters

callback default user function to call on GPIO irq. Note only one of these can be set per processor.

◆ gpio_set_irq_enabled()

```
void gpio_set_irq_enabled ( uint     gpio,
                            uint32_t event_mask,
                            bool     enabled
                          )
```

Enable or disable specific interrupt events for specified GPIO.

This function sets which GPIO events cause a GPIO interrupt on the calling core. See gpio_set_irq_callback, gpio_set_irq_enabled_with_callback and gpio_add_raw_irq_handler to set up a GPIO interrupt handler to handle the events.

NOTE

The IO IRQs are independent per-processor. This configures the interrupt events for the processor that calls the function.

Parameters

gpio	GPIO number
event_mask	Which events will cause an interrupt
enabled	Enable or disable flag

Events is a bitmask of the following gpio_irq_level values:

bit | constant | interrupt -—|---------- — 0 | GPIO_IRQ_LEVEL_LOW | Continuously while level is low 1 | GPIO_IRQ_LEVEL_HIGH | Continuously while level is high 2 | GPIO_IRQ_EDGE_FALL | On each transition from high to low 3 | GPIO_IRQ_EDGE_RISE | On each transition from low to high

which are specified in gpio_irq_level

◆ gpio_set_irq_enabled_with_callback()

```
void gpio_set_irq_enabled_with_callback ( uint                gpio,
                                          uint32_t            event_mask,
                                          bool                enabled,
                                          gpio_irq_callback_t callback
                                        )
```

Convenience function which performs multiple GPIO IRQ related initializations.

This method is a slightly eclectic mix of initialization, that:

- Updates whether the specified events for the specified GPIO causes an interrupt on the calling core based on the enable flag.
- Sets the callback handler for the calling core to callback (or clears the handler if the callback is NULL).
- Enables GPIO IRQs on the current core if enabled is true.

This method is commonly used to perform a one time setup, and following that any additional IRQs/events are enabled via gpio_set_irq_enabled. All GPIOs/events added in this way on the same core share the same callback; for multiple independent handlers for different GPIOs you should use gpio_add_raw_irq_handler and related functions.

This method is equivalent to:

```
gpio_set_irq_enabled(gpio, event_mask, enabled);
gpio_set_irq_callback(callback);
if (enabled) irq_set_enabled(IO_IRQ_BANK0, true);
```

NOTE

The IO IRQs are independent per-processor. This method affects only the processor that calls the function.

Parameters

gpio	GPIO number
event_mask	Which events will cause an interrupt. See gpio_irq_level for details.
enabled	Enable or disable flag
callback	user function to call on GPIO irq. if NULL, the callback is removed

◆ gpio_set_irqover()

```
void gpio_set_irqover ( uint gpio,
                        uint value
                      )
```

Set GPIO IRQ override.

Optionally invert a GPIO IRQ signal, or drive it high or low

Parameters

gpio	GPIO number
value	See gpio_override

◆ gpio_set_mask()

```
static void gpio_set_mask ( uint32_t mask )                    inline static
```

Drive high every GPIO appearing in mask.

Parameters

mask	Bitmask of GPIO values to set, as bits 0-29

◆ gpio_set_oeover()

```
void gpio_set_oeover ( uint gpio,
                       uint value
                     )
```

Select GPIO output enable override.

Parameters

gpio	GPIO number
value	See gpio_override

◆ gpio_set_outover()

```
void gpio_set_outover ( uint gpio,
                        uint value
                      )
```

Set GPIO output override.

Parameters

gpio	GPIO number
value	See gpio_override

◆ gpio_set_pulls()

```
void gpio_set_pulls ( uint gpio,
                      bool up,
                      bool down
                    )
```

Select up and down pulls on specific GPIO.

Parameters

gpio	GPIO number
up	If true set a pull up on the GPIO
down	If true set a pull down on the GPIO

NOTE

On the RP2040, setting both pulls enables a "bus keep" function, i.e. a weak pull to whatever is current high/low state of GPIO.

◆ gpio_set_slew_rate()

```
void gpio_set_slew_rate ( uint                  gpio,
                          enum gpio_slew_rate   slew
                        )
```

Set slew rate for a specified GPIO.

See also gpio_get_slew_rate

Parameters

gpio	GPIO number
slew	GPIO output slew rate

◆ gpio_xor_mask()

```
static void gpio_xor_mask ( uint32_t mask )                    inline  static
```

Toggle every GPIO appearing in mask.

Parameters

mask Bitmask of GPIO values to toggle, as bits 0-29

hardware_i2c

Part of: Hardware APIs

Functions

- **uint i2c_init (i2c_inst_t *i2c, uint baudrate)**

  Initialise the I2C HW block.

- **void i2c_deinit (i2c_inst_t *i2c)**

  Disable the I2C HW block.

- **uint i2c_set_baudrate (i2c_inst_t *i2c, uint baudrate)**

  Set I2C baudrate.

- **void i2c_set_slave_mode (i2c_inst_t *i2c, bool slave, uint8_t addr)**

  Set I2C port to slave mode.

- **static uint i2c_hw_index (i2c_inst_t *i2c)**

  Convert I2C instance to hardware instance number.

- **int i2c_write_blocking_until (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop, absolute_time_t until)**

  Attempt to write specified number of bytes to address, blocking until the specified absolute time is reached.

- **int i2c_read_blocking_until (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len, bool nostop, absolute_time_t until)**

  Attempt to read specified number of bytes from address, blocking until the specified absolute time is reached.

- **static int i2c_write_timeout_us (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop, uint timeout_us)**

  Attempt to write specified number of bytes to address, with timeout.

- **static int i2c_read_timeout_us (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len, bool nostop, uint timeout_us)**

  Attempt to read specified number of bytes from address, with timeout.

- **int i2c_write_blocking (i2c_inst_t *i2c, uint8_t addr, const uint8_t *src, size_t len, bool nostop)**

  Attempt to write specified number of bytes to address, blocking.

- **int i2c_read_blocking (i2c_inst_t *i2c, uint8_t addr, uint8_t *dst, size_t len, bool nostop)**

  Attempt to read specified number of bytes from address, blocking.

- **static size_t i2c_get_write_available (i2c_inst_t *i2c)**

  Determine non-blocking write space available.

- **static size_t i2c_get_read_available (i2c_inst_t *i2c)**

  Determine number of bytes received.

- **static void i2c_write_raw_blocking (i2c_inst_t *i2c, const uint8_t *src, size_t len)**

  Write direct to TX FIFO.

- **static void i2c_read_raw_blocking (i2c_inst_t *i2c, uint8_t *dst, size_t len)**

  Read direct from RX FIFO.

- **static uint8_t i2c_read_byte_raw (i2c_inst_t *i2c)**

  Pop a byte from I2C Rx FIFO.

- **static void i2c_write_byte_raw (i2c_inst_t *i2c, uint8_t value)**

  Push a byte into I2C Tx FIFO.

- **static uint i2c_get_dreq (i2c_inst_t *i2c, bool is_tx)**

  Return the DREQ to use for pacing transfers to/from a particular I2C instance.

- **i2c_inst_t i2c0_inst**

Detailed Description

I2C Controller API

The I2C bus is a two-wire serial interface, consisting of a serial data line SDA and a serial clock SCL. These wires carry information between the devices connected to the bus. Each device is recognized by a unique 7-bit address and can operate as either a "transmitter" or "receiver", depending on the function of the device. Devices can also be considered as masters or slaves when performing data transfers. A master is a device that initiates a data transfer on the bus and generates the clock signals to permit that transfer. The first byte in the data transfer always contains the 7-bit address and a read/write bit in the LSB position. This API takes care of toggling the read/write bit. After this, any device addressed is considered a slave.

This API allows the controller to be set up as a master or a slave using the i2c_set_slave_mode function.

The external pins of each controller are connected to GPIO pins as defined in the GPIO muxing table in the datasheet. The muxing options give some IO flexibility, but each controller external pin should be connected to only one GPIO.

Note that the controller does NOT support High speed mode or Ultra-fast speed mode, the fastest operation being fast mode plus at up to 1000Kb/s.

See the datasheet for more information on the I2C controller and its usage.

Example

```
// Sweep through all 7-bit I2C addresses, to see if any slaves are present on
// the I2C bus. Print out a table that looks like this:
//
// I2C Bus Scan
//   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
// 0
// 1       @
// 2
// 3             @
// 4
// 5
// 6
// 7
//
// E.g. if slave addresses 0x12 and 0x34 were acknowledged.

#include <stdio.h>
#include "pico/stdlib.h"
#include "pico/binary_info.h"
#include "hardware/i2c.h"

// I2C reserves some addresses for special purposes. We exclude these from the scan.
// These are any addresses of the form 000 0xxx or 111 1xxx

bool reserved_addr(uint8_t addr) {
    return (addr & 0x78) == 0 || (addr & 0x78) == 0x78;
}
```

```
int main() {
    // Enable UART so we can print status output
    stdio_init_all();
#if !defined(i2c_default) || !defined(PICO_DEFAULT_I2C_SDA_PIN) || !defined(PICO_DEFAULT_I2C_SC
L_PIN)
#warning i2c/bus_scan example requires a board with I2C pins
    puts("Default I2C pins were not defined");
#else
    // This example will use I2C0 on the default SDA and SCL pins (GP4, GP5 on a Pico)
    i2c_init(i2c_default, 100 * 1000);
    gpio_set_function(PICO_DEFAULT_I2C_SDA_PIN, GPIO_FUNC_I2C);
    gpio_set_function(PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2C);
    gpio_pull_up(PICO_DEFAULT_I2C_SDA_PIN);
    gpio_pull_up(PICO_DEFAULT_I2C_SCL_PIN);
    // Make the I2C pins available to picotool
    bi_decl(bi_2pins_with_func(PICO_DEFAULT_I2C_SDA_PIN, PICO_DEFAULT_I2C_SCL_PIN, GPIO_FUNC_I2
C));

    printf("\nI2C Bus Scan\n");
    printf("   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F\n");

    for (int addr = 0; addr < (1 << 7); ++addr) {
        if (addr % 16 == 0) {
            printf("%02x ", addr);
        }

        // Perform a 1-byte dummy read from the probe address. If a slave
        // acknowledges this address, the function returns the number of bytes
        // transferred. If the address byte is ignored, the function returns
        // -1.

        // Skip over any reserved addresses.
        int ret;
        uint8_t rxdata;
        if (reserved_addr(addr))
            ret = PICO_ERROR_GENERIC;
        else
            ret = i2c_read_blocking(i2c_default, addr, &rxdata, 1, false);

        printf(ret < 0 ? "." : "@");
        printf(addr % 16 == 15 ? "\n" : "  ");
    }
    printf("Done.\n");
    return 0;
#endif
}
```

Function Documentation

◆ i2c_deinit()

void i2c_deinit (i2c_inst_t * ***i2c***)

Disable the I2C HW block.

Parameters

i2c Either i2c0 or i2c1

Disable the I2C again if it is no longer used. Must be reinitialised before being used again.

◆ i2c_get_dreq()

static uint i2c_get_dreq (i2c_inst_t * ***i2c***,
bool ***is_tx***
) inline static

Return the DREQ to use for pacing transfers to/from a particular I2C instance.

Parameters

i2c Either i2c0 or i2c1
is_tx true for sending data to the I2C instance, false for receiving data from the I2C instance

◆ i2c_get_read_available()

static size_t i2c_get_read_available (i2c_inst_t * ***i2c***) inline static

Determine number of bytes received.

Parameters

i2c Either i2c0 or i2c1

Returns

0 if no data available, if return is nonzero at least that many bytes can be read without blocking.

◆ i2c_get_write_available()

static size_t i2c_get_write_available (i2c_inst_t * ***i2c***) inline static

Determine non-blocking write space available.

Parameters

i2c Either i2c0 or i2c1

Returns

0 if no space is available in the I2C to write more data. If return is nonzero, at least that many bytes can be written without blocking.

◆ i2c_hw_index()

static uint i2c_hw_index (i2c_inst_t * ***i2c***) inline static

Convert I2C instance to hardware instance number.

Parameters

i2c I2C instance

Returns

Number of I2C, 0 or 1.

◆ i2c_init()

uint i2c_init (i2c_inst_t * ***i2c***,
uint ***baudrate***
)

Initialise the I2C HW block.

Put the I2C hardware into a known state, and enable it. Must be called before other functions. By default, the I2C is configured to operate as a master.

The I2C bus frequency is set as close as possible to requested, and the actual rate set is returned

Parameters

i2c	Either i2c0 or i2c1
baudrate	Baudrate in Hz (e.g. 100kHz is 100000)

Returns

Actual set baudrate

◆ i2c_read_blocking()

```
int i2c_read_blocking ( i2c_inst_t *  i2c,
                        uint8_t       addr,
                        uint8_t *     dst,
                        size_t        len,
                        bool          nostop
                      )
```

Attempt to read specified number of bytes from address, blocking.

Parameters

i2c	Either i2c0 or i2c1
addr	7-bit address of device to read from
dst	Pointer to buffer to receive data
len	Length of data in bytes to receive
nostop	If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.

Returns

Number of bytes read, or PICO_ERROR_GENERIC if address not acknowledged or no device present.

◆ i2c_read_blocking_until()

```
int i2c_read_blocking_until ( i2c_inst_t *       i2c,
                              uint8_t            addr,
                              uint8_t *          dst,
                              size_t             len,
                              bool               nostop,
                              absolute_time_t    until
                            )
```

Attempt to read specified number of bytes from address, blocking until the specified absolute time is reached.

Parameters

i2c	Either i2c0 or i2c1
addr	7-bit address of device to read from
dst	Pointer to buffer to receive data
len	Length of data in bytes to receive
nostop	If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.
until	The absolute time that the block will wait until the entire transaction is complete.

Returns

Number of bytes read, or PICO_ERROR_GENERIC if address not acknowledged, no device present, or PICO_ERROR_TIMEOUT if a timeout occurred.

◆ i2c_read_byte_raw()

```
static uint8_t i2c_read_byte_raw ( i2c_inst_t *  i2c )
```
inline static

Pop a byte from I2C Rx FIFO.

This function is non-blocking and assumes the Rx FIFO isn't empty.

Parameters

i2c I2C instance.

Returns

uint8_t Byte value.

◆ i2c_read_raw_blocking()

```
static void i2c_read_raw_blocking ( i2c_inst_t *  i2c,
                                    uint8_t *     dst,
                                    size_t        len
                                  )
```
inline static

Read direct from RX FIFO.

Parameters

i2c Either i2c0 or i2c1
dst Buffer to accept data
len Number of bytes to read

Reads directly from the I2C RX FIFO which is mainly useful for slave-mode operation.

◆ i2c_read_timeout_us()

```
static int i2c_read_timeout_us ( i2c_inst_t *  i2c,
                                 uint8_t       addr,
                                 uint8_t *     dst,
                                 size_t        len,
                                 bool          nostop,
                                 uint          timeout_us
                               )
```
inline static

Attempt to read specified number of bytes from address, with timeout.

Parameters

i2c	Either i2c0 or i2c1
addr	7-bit address of device to read from
dst	Pointer to buffer to receive data
len	Length of data in bytes to receive
nostop	If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.
timeout_us	The time that the function will wait for the entire transaction to complete

Returns

Number of bytes read, or PICO_ERROR_GENERIC if address not acknowledged, no device present, or PICO_ERROR_TIMEOUT if a timeout occurred.

◆ i2c_set_baudrate()

```
uint i2c_set_baudrate ( i2c_inst_t *  i2c,
                        uint          baudrate
                      )
```

Set I2C baudrate.

Set I2C bus frequency as close as possible to requested, and return actual rate set. Baudrate may not be as exactly requested due to clocking limitations.

Parameters

i2c	Either i2c0 or i2c1
baudrate	Baudrate in Hz (e.g. 100kHz is 100000)

Returns

Actual set baudrate

◆ i2c_set_slave_mode()

```
void i2c_set_slave_mode ( i2c_inst_t *  i2c,
                          bool          slave,
                          uint8_t       addr
                        )
```

Set I2C port to slave mode.

Parameters

i2c	Either i2c0 or i2c1
slave	true to use slave mode, false to use master mode
addr	If `slave` is true, set the slave address to this value

◆ i2c_write_blocking()

```
int i2c_write_blocking ( i2c_inst_t *       i2c,
                         uint8_t            addr,
                         const uint8_t *    src,
                         size_t             len,
                         bool               nostop
                       )
```

Attempt to write specified number of bytes to address, blocking.

Parameters

i2c	Either i2c0 or i2c1
addr	7-bit address of device to write to
src	Pointer to data to send
len	Length of data in bytes to send
nostop	If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.

Returns

Number of bytes written, or PICO_ERROR_GENERIC if address not acknowledged, no device present.

◆ i2c_write_blocking_until()

```
int i2c_write_blocking_until ( i2c_inst_t *      i2c,
                               uint8_t           addr,
                               const uint8_t *   src,
                               size_t            len,
                               bool              nostop,
                               absolute_time_t   until
                             )
```

Attempt to write specified number of bytes to address, blocking until the specified absolute time is reached.

Parameters

i2c	Either i2c0 or i2c1
addr	7-bit address of device to write to
src	Pointer to data to send
len	Length of data in bytes to send
nostop	If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.
until	The absolute time that the block will wait until the entire transaction is complete. Note, an individual timeout of this value divided by the length of data is applied for each byte transfer, so if the first or subsequent bytes fails to transfer within that sub timeout, the function will return with an error.

Returns

Number of bytes written, or PICO_ERROR_GENERIC if address not acknowledged, no device present, or PICO_ERROR_TIMEOUT if a timeout occurred.

◆ i2c_write_byte_raw()

```
static void i2c_write_byte_raw ( i2c_inst_t *  i2c,
                                 uint8_t       value
                               )                                    inline  static
```

Push a byte into I2C Tx FIFO.

This function is non-blocking and assumes the Tx FIFO isn't full.

Parameters

i2c	I2C instance.
value	Byte value.

◆ i2c_write_raw_blocking()

```
static void i2c_write_raw_blocking ( i2c_inst_t *      i2c,
                                     const uint8_t *   src,
                                     size_t            len
                                   )                                inline  static
```

Write direct to TX FIFO.

Parameters

i2c Either i2c0 or i2c1
src Data to send
len Number of bytes to send

Writes directly to the I2C TX FIFO which is mainly useful for slave-mode operation.

◆ i2c_write_timeout_us()

```
static int i2c_write_timeout_us ( i2c_inst_t *      i2c,
                                  uint8_t           addr,
                                  const uint8_t *   src,
                                  size_t            len,
                                  bool              nostop,
                                  uint              timeout_us
                                )                                  inline  static
```

Attempt to write specified number of bytes to address, with timeout.

Parameters

i2c	Either i2c0 or i2c1
addr	7-bit address of device to write to
src	Pointer to data to send
len	Length of data in bytes to send
nostop	If true, master retains control of the bus at the end of the transfer (no Stop is issued), and the next transfer will begin with a Restart rather than a Start.
timeout_us	The time that the function will wait for the entire transaction to complete. Note, an individual timeout of this value divided by the length of data is applied for each byte transfer, so if the first or subsequent bytes fails to transfer within that sub timeout, the function will return with an error.

Returns

Number of bytes written, or PICO_ERROR_GENERIC if address not acknowledged, no device present, or PICO_ERROR_TIMEOUT if a timeout occurred.

Variable Documentation

◆ i2c0_inst

```
i2c_inst_t i2c0_inst                                               extern
```

The I2C identifiers for use in I2C functions.

e.g. i2c_init(i2c0, 48000)

hardware_interp

Part of: Hardware APIs

Modules

- **interp_config**

  Interpolator configuration.

Functions

- **void interp_claim_lane (interp_hw_t *interp, uint lane)**

  Claim the interpolator lane specified.

- **void interp_claim_lane_mask (interp_hw_t *interp, uint lane_mask)**

  Claim the interpolator lanes specified in the mask.

- **void interp_unclaim_lane (interp_hw_t *interp, uint lane)**

  Release a previously claimed interpolator lane.

- **bool interp_lane_is_claimed (interp_hw_t *interp, uint lane)**

  Determine if an interpolator lane is claimed.

- **void interp_unclaim_lane_mask (interp_hw_t *interp, uint lane_mask)**

  Release previously claimed interpolator lanes.

- **static void interp_set_force_bits (interp_hw_t *interp, uint lane, uint bits)**

  Directly set the force bits on a specified lane.

- **void interp_save (interp_hw_t *interp, interp_hw_save_t *saver)**

  Save the specified interpolator state.

- **void interp_restore (interp_hw_t *interp, interp_hw_save_t *saver)**

  Restore an interpolator state.

- **static void interp_set_base (interp_hw_t *interp, uint lane, uint32_t val)**

  Sets the interpolator base register by lane.

- **static uint32_t interp_get_base (interp_hw_t *interp, uint lane)**

  Gets the content of interpolator base register by lane.

- **static void interp_set_base_both (interp_hw_t *interp, uint32_t val)**

  Sets the interpolator base registers simultaneously.

- **static void interp_set_accumulator (interp_hw_t *interp, uint lane, uint32_t val)**

  Sets the interpolator accumulator register by lane.

- **static uint32_t interp_get_accumulator (interp_hw_t *interp, uint lane)**

  Gets the content of the interpolator accumulator register by lane.

- **static uint32_t interp_pop_lane_result (interp_hw_t *interp, uint lane)**

  Read lane result, and write lane results to both accumulators to update the interpolator.

- **static uint32_t interp_peek_lane_result (interp_hw_t *interp, uint lane)**

  Read lane result.

- **static uint32_t interp_pop_full_result (interp_hw_t *interp)**

  Read lane result, and write lane results to both accumulators to update the interpolator.

- **static uint32_t interp_peek_full_result (interp_hw_t *interp)**

  Read lane result.

- **static void interp_add_accumulater (interp_hw_t *interp, uint lane, uint32_t val)**

Add to accumulator.

- **static uint32_t interp_get_raw (interp_hw_t *interp, uint lane)**

  Get raw lane value.

Detailed Description

Hardware Interpolator API

Each core is equipped with two interpolators (INTERP0 and INTERP1) which can be used to accelerate tasks by combining certain pre-configured simple operations into a single processor cycle. Intended for cases where the pre-configured operation is repeated a large number of times, this results in code which uses both fewer CPU cycles and fewer CPU registers in the time critical sections of the code.

The interpolators are used heavily to accelerate audio operations within the SDK, but their flexible configuration make it possible to optimise many other tasks such as quantization and dithering, table lookup address generation, affine texture mapping, decompression and linear feedback.

Please refer to the RP2040 datasheet for more information on the HW interpolators and how they work.

Function Documentation

◆ interp_add_accumulater()

```
static void interp_add_accumulater ( interp_hw_t *  interp,
                                     uint           lane,
                                     uint32_t       val
                                   )                                    inline  static
```

Add to accumulator.

Atomically add the specified value to the accumulator on the specified lane

Parameters

interp	Interpolator instance, interp0 or interp1.
lane	The lane number, 0 or 1
val	Value to add

◆ interp_claim_lane()

```
void interp_claim_lane ( interp_hw_t *  interp,
                         uint           lane
                       )
```

Claim the interpolator lane specified.

Use this function to claim exclusive access to the specified interpolator lane.

This function will panic if the lane is already claimed.

Parameters

interp	Interpolator on which to claim a lane. interp0 or interp1
lane	The lane number, 0 or 1.

◆ interp_claim_lane_mask()

```
void interp_claim_lane_mask ( interp_hw_t * interp,
                              uint          lane_mask
                            )
```

Claim the interpolator lanes specified in the mask.

Parameters

interp	Interpolator on which to claim lanes. interp0 or interp1
lane_mask	Bit pattern of lanes to claim (only bits 0 and 1 are valid)

◆ interp_get_accumulator()

```
static uint32_t interp_get_accumulator ( interp_hw_t * interp,
                                         uint          lane
                                       )                          inline  static
```

Gets the content of the interpolator accumulator register by lane.

Parameters

interp	Interpolator instance, interp0 or interp1.
lane	The lane number, 0 or 1

Returns

The current content of the register

◆ interp_get_base()

```
static uint32_t interp_get_base ( interp_hw_t * interp,
                                  uint          lane
                                )                                 inline  static
```

Gets the content of interpolator base register by lane.

Parameters

interp	Interpolator instance, interp0 or interp1.
lane	The lane number, 0 or 1 or 2

Returns

The current content of the lane base register

◆ interp_get_raw()

```
static uint32_t interp_get_raw ( interp_hw_t * interp,
                                 uint          lane
                               )                                  inline  static
```

Get raw lane value.

Returns the raw shift and mask value from the specified lane, BASE0 is NOT added

Parameters

interp Interpolator instance, interp0 or interp1.
lane The lane number, 0 or 1

Returns

The raw shift/mask value

◆ interp_lane_is_claimed()

```
bool interp_lane_is_claimed ( interp_hw_t * interp,
                              uint          lane
                            )
```

Determine if an interpolator lane is claimed.

Parameters

interp Interpolator whose lane to check
lane The lane number, 0 or 1

Returns

true if claimed, false otherwise

See also interp_claim_lane interp_claim_lane_mask

◆ interp_peek_full_result()

```
static uint32_t interp_peek_full_result ( interp_hw_t * interp )                inline static
```

Read lane result.

Parameters

interp Interpolator instance, interp0 or interp1.

Returns

The content of the FULL register

◆ interp_peek_lane_result()

```
static uint32_t interp_peek_lane_result ( interp_hw_t * interp,
                                          uint          lane
                                        )                                       inline static
```

Read lane result.

Parameters

interp Interpolator instance, interp0 or interp1.
lane The lane number, 0 or 1

Returns

The content of the lane result register

◆ interp_pop_full_result()

static uint32_t interp_pop_full_result (interp_hw_t * ***interp***) inline static

Read lane result, and write lane results to both accumulators to update the interpolator.

Parameters

interp Interpolator instance, interp0 or interp1.

Returns

The content of the FULL register

◆ interp_pop_lane_result()

static uint32_t interp_pop_lane_result (interp_hw_t * ***interp***,
uint ***lane***
) inline static

Read lane result, and write lane results to both accumulators to update the interpolator.

Parameters

interp Interpolator instance, interp0 or interp1.
lane The lane number, 0 or 1

Returns

The content of the lane result register

◆ interp_restore()

void interp_restore (interp_hw_t * ***interp***,
interp_hw_save_t * ***saver***
)

Restore an interpolator state.

Parameters

interp Interpolator instance, interp0 or interp1.
saver Pointer to save structure to reapply to the specified interpolator

◆ interp_save()

void interp_save (interp_hw_t * ***interp***,
interp_hw_save_t * ***saver***
)

Save the specified interpolator state.

Can be used to save state if you need an interpolator for another purpose, state can then be recovered afterwards and continue from that point

Parameters

interp	Interpolator instance, interp0 or interp1.
saver	Pointer to the save structure to fill in

◆ interp_set_accumulator()

```
static void interp_set_accumulator ( interp_hw_t *  interp,
                                     uint           lane,
                                     uint32_t       val
                                   )                                  inline  static
```

Sets the interpolator accumulator register by lane.

Parameters

interp	Interpolator instance, interp0 or interp1.
lane	The lane number, 0 or 1
val	The value to apply to the register

◆ interp_set_base()

```
static void interp_set_base ( interp_hw_t *  interp,
                              uint           lane,
                              uint32_t       val
                            )                                         inline  static
```

Sets the interpolator base register by lane.

Parameters

interp	Interpolator instance, interp0 or interp1.
lane	The lane number, 0 or 1 or 2
val	The value to apply to the register

◆ interp_set_base_both()

```
static void interp_set_base_both ( interp_hw_t *  interp,
                                   uint32_t       val
                                 )                                    inline  static
```

Sets the interpolator base registers simultaneously.

The lower 16 bits go to BASE0, upper bits to BASE1 simultaneously. Each half is sign-extended to 32 bits if that lane's SIGNED flag is set.

Parameters

interp	Interpolator instance, interp0 or interp1.
val	The value to apply to the register

◆ interp_set_force_bits()

```
static void interp_set_force_bits ( interp_hw_t * interp,
                                    uint          lane,
                                    uint          bits
                                  )                                    inline  static
```

Directly set the force bits on a specified lane.

These bits are ORed into bits 29:28 of the lane result presented to the processor on the bus. There is no effect on the internal 32-bit datapath.

Useful for using a lane to generate sequence of pointers into flash or SRAM, saving a subsequent OR or add operation.

Parameters

interp	Interpolator instance, interp0 or interp1.
lane	The lane to set
bits	The bits to set (bits 0 and 1, value range 0-3)

◆ interp_unclaim_lane()

```
void interp_unclaim_lane ( interp_hw_t * interp,
                           uint          lane
                         )
```

Release a previously claimed interpolator lane.

Parameters

interp	Interpolator on which to release a lane. interp0 or interp1
lane	The lane number, 0 or 1

◆ interp_unclaim_lane_mask()

```
void interp_unclaim_lane_mask ( interp_hw_t * interp,
                                uint          lane_mask
                              )
```

Release previously claimed interpolator lanes.

See also interp_claim_lane_mask

Parameters

interp	Interpolator on which to release lanes. interp0 or interp1
lane_mask	Bit pattern of lanes to unclaim (only bits 0 and 1 are valid)

interp_config

Part of: Hardware APIs » hardware_interp
Interpolator configuration. More...

Functions

- **`static void interp_config_set_shift (interp_config *c, uint shift)`**

  Set the interpolator shift value.

- **static void interp_config_set_mask (interp_config *c, uint mask_lsb, uint mask_msb)**

  Set the interpolator mask range.

- **static void interp_config_set_cross_input (interp_config *c, bool cross_input)**

  Enable cross input.

- **static void interp_config_set_cross_result (interp_config *c, bool cross_result)**

  Enable cross results.

- **static void interp_config_set_signed (interp_config *c, bool _signed)**

  Set sign extension.

- **static void interp_config_set_add_raw (interp_config *c, bool add_raw)**

  Set raw add option.

- **static void interp_config_set_blend (interp_config *c, bool blend)**

  Set blend mode.

- **static void interp_config_set_clamp (interp_config *c, bool clamp)**

  Set interpolator clamp mode (Interpolator 1 only)

- **static void interp_config_set_force_bits (interp_config *c, uint bits)**

  Set interpolator Force bits.

- **static interp_config interp_default_config (void)**

  Get a default configuration.

- **static void interp_set_config (interp_hw_t *interp, uint lane, interp_config *config)**

  Send configuration to a lane.

Detailed Description

Interpolator configuration.

Each interpolator needs to be configured, these functions provide handy helpers to set up configuration structures.

Function Documentation

◆ interp_config_set_add_raw()

static void interp_config_set_add_raw (interp_config * ***c***,
bool ***add_raw***
) inline static

Set raw add option.

When enabled, mask + shift is bypassed for LANE0 result. This does not affect the FULL result.

Parameters

c	Pointer to interpolation config
add_raw	If true, enable raw add option.

◆ interp_config_set_blend()

```
static void interp_config_set_blend ( interp_config * c,
                                      bool            blend
                                    )                                    inline  static
```

Set blend mode.

If enabled, LANE1 result is a linear interpolation between BASE0 and BASE1, controlled by the 8 LSBs of lane 1 shift and mask value (a fractional number between 0 and 255/256ths)

LANE0 result does not have BASE0 added (yields only the 8 LSBs of lane 1 shift+mask value)

FULL result does not have lane 1 shift+mask value added (BASE2 + lane 0 shift+mask)

LANE1 SIGNED flag controls whether the interpolation is signed or unsig

Parameters

c	Pointer to interpolation config
blend	Set true to enable blend mode.

◆ interp_config_set_clamp()

```
static void interp_config_set_clamp ( interp_config * c,
                                      bool            clamp
                                    )                                    inline  static
```

Set interpolator clamp mode (Interpolator 1 only)

Only present on INTERP1 on each core. If CLAMP mode is enabled:

- LANE0 result is a shifted and masked ACCUM0, clamped by a lower bound of BASE0 and an upper bound of BASE1.
- Signedness of these comparisons is determined by LANE0_CTRL_SIGNED

Parameters

c	Pointer to interpolation config
clamp	Set true to enable clamp mode

◆ interp_config_set_cross_input()

```
static void interp_config_set_cross_input ( interp_config * c,
                                            bool            cross_input
                                          )                              inline  static
```

Enable cross input.

Allows feeding of the accumulator content from the other lane back in to this lanes shift+mask hardware. This will take effect even if the interp_config_set_add_raw option is set as the cross input mux is before the shift+mask bypass

Parameters

c	Pointer to interpolation config
cross_input	If true, enable the cross input.

◆ interp_config_set_cross_result()

```
static void interp_config_set_cross_result ( interp_config * c,
```

bool *cross_result*
)
inline static

Enable cross results.

Allows feeding of the other lane's result into this lane's accumulator on a POP operation.

Parameters

c	Pointer to interpolation config
cross_result	If true, enables the cross result

◆ **interp_config_set_force_bits()**

static void interp_config_set_force_bits (interp_config * ***c***,
uint ***bits***
)
inline static

Set interpolator Force bits.

ORed into bits 29:28 of the lane result presented to the processor on the bus.

No effect on the internal 32-bit datapath. Handy for using a lane to generate sequence of pointers into flash or SRAM

Parameters

c	Pointer to interpolation config
bits	Sets the force bits to that specified. Range 0-3 (two bits)

◆ **interp_config_set_mask()**

static void interp_config_set_mask (interp_config * ***c***,
uint ***mask_lsb***,
uint ***mask_msb***
)
inline static

Set the interpolator mask range.

Sets the range of bits (least to most) that are allowed to pass through the interpolator

Parameters

c	Pointer to interpolation config
mask_lsb	The least significant bit allowed to pass
mask_msb	The most significant bit allowed to pass

◆ **interp_config_set_shift()**

static void interp_config_set_shift (interp_config * ***c***,
uint ***shift***
)
inline static

Set the interpolator shift value.

Sets the number of bits the accumulator is shifted before masking, on each iteration.

Parameters

c	Pointer to an interpolator config

shift Number of bits

◆ **interp_config_set_signed()**

```
static void interp_config_set_signed ( interp_config * c,
                                       bool            _signed
                                     )                                  inline  static
```

Set sign extension.

Enables signed mode, where the shifted and masked accumulator value is sign-extended to 32 bits before adding to BASE1, and LANE1 PEEK/POP results appear extended to 32 bits when read by processor.

Parameters

c	Pointer to interpolation config
_signed	If true, enables sign extension

◆ **interp_default_config()**

```
static interp_config interp_default_config ( void )                     inline  static
```

Get a default configuration.

Returns

A default interpolation configuration

◆ **interp_set_config()**

```
static void interp_set_config ( interp_hw_t * interp,
                                uint          lane,
                                interp_config * config
                              )                                         inline  static
```

Send configuration to a lane.

If an invalid configuration is specified (ie a lane specific item is set on wrong lane), depending on setup this function can panic.

Parameters

interp	Interpolator instance, interp0 or interp1.
lane	The lane to set
config	Pointer to interpolation config

hardware_irq

Part of: Hardware APIs

Typedefs

- **`typedef void(* irq_handler_t) (void)`**

  Interrupt handler function type.

Functions

- **void irq_set_priority (uint num, uint8_t hardware_priority)**

  Set specified interrupt's priority.

- **uint irq_get_priority (uint num)**

  Get specified interrupt's priority.

- **void irq_set_enabled (uint num, bool enabled)**

  Enable or disable a specific interrupt on the executing core.

- **bool irq_is_enabled (uint num)**

  Determine if a specific interrupt is enabled on the executing core.

- **void irq_set_mask_enabled (uint32_t mask, bool enabled)**

  Enable/disable multiple interrupts on the executing core.

- **void irq_set_exclusive_handler (uint num, irq_handler_t handler)**

  Set an exclusive interrupt handler for an interrupt on the executing core.

- **irq_handler_t irq_get_exclusive_handler (uint num)**

  Get the exclusive interrupt handler for an interrupt on the executing core.

- **void irq_add_shared_handler (uint num, irq_handler_t handler, uint8_t order_priority)**

  Add a shared interrupt handler for an interrupt on the executing core.

- **void irq_remove_handler (uint num, irq_handler_t handler)**

  Remove a specific interrupt handler for the given irq number on the executing core.

- **bool irq_has_shared_handler (uint num)**

  Determine if the current handler for the given number is shared.

- **irq_handler_t irq_get_vtable_handler (uint num)**

  Get the current IRQ handler for the specified IRQ from the currently installed hardware vector table (VTOR) of the execution core.

- **static void irq_clear (uint int_num)**

  Clear a specific interrupt on the executing core.

- **void irq_set_pending (uint num)**

  Force an interrupt to be pending on the executing core.

- **void user_irq_claim (uint irq_num)**

  Claim ownership of a user IRQ on the calling core.

- **void user_irq_unclaim (uint irq_num)**

  Mark a user IRQ as no longer used on the calling core.

- **int user_irq_claim_unused (bool required)**

  Claim ownership of a free user IRQ on the calling core.

Detailed Description

Hardware interrupt handling

The RP2040 uses the standard ARM nested vectored interrupt controller (NVIC).

Interrupts are identified by a number from 0 to 31.

On the RP2040, only the lower 26 IRQ signals are connected on the NVIC; IRQs 26 to 31 are tied to zero (never firing).

There is one NVIC per core, and each core's NVIC has the same hardware interrupt lines routed to it, with the exception of the IO interrupts where there is one IO interrupt per bank, per core. These are completely independent, so, for example, processor 0 can be interrupted by GPIO 0 in bank 0, and processor 1 by GPIO 1 in the same bank.

NOTE

That all IRQ APIs affect the executing core only (i.e. the core calling the function).
You should not enable the same (shared) IRQ number on both cores, as this will lead to race conditions or starvation of one of the cores. Additionally, don't forget that disabling interrupts on one core does not disable interrupts on the other core.

There are three different ways to set handlers for an IRQ:

- Calling irq_add_shared_handler() at runtime to add a handler for a multiplexed interrupt (e.g. GPIO bank) on the current core. Each handler, should check and clear the relevant hardware interrupt source
- Calling irq_set_exclusive_handler() at runtime to install a single handler for the interrupt on the current core
- Defining the interrupt handler explicitly in your application (e.g. by defining void `isr_dma_0` will make that function the handler for the DMA_IRQ_0 on core 0, and you will not be able to change it using the above APIs at runtime). Using this method can cause link conflicts at runtime, and offers no runtime performance benefit (i.e, it should not generally be used).

NOTE

If an IRQ is enabled and fires with no handler installed, a breakpoint will be hit and the IRQ number will be in register r0.

Interrupt Numbers

Interrupts are numbered as follows, a set of defines is available (intctrl.h) with these names to avoid using the numbers directly.

IRQ	Interrupt Source
0	TIMER_IRQ_0
1	TIMER_IRQ_1
2	TIMER_IRQ_2
3	TIMER_IRQ_3
4	PWM_IRQ_WRAP
5	USBCTRL_IRQ
6	XIP_IRQ
7	PIO0_IRQ_0
8	PIO0_IRQ_1
9	PIO1_IRQ_0
10	PIO1_IRQ_1
11	DMA_IRQ_0
12	DMA_IRQ_1
13	IO_IRQ_BANK0
14	IO_IRQ_QSPI
15	SIO_IRQ_PROC0
16	SIO_IRQ_PROC1
17	CLOCKS_IRQ
18	SPI0_IRQ

19	SPI1_IRQ
20	UART0_IRQ
21	UART1_IRQ
22	ADC0_IRQ_FIFO
23	I2C0_IRQ
24	I2C1_IRQ
25	RTC_IRQ

Typedef Documentation

◆ irq_handler_t

```
typedef void(* irq_handler_t) (void)
```

Interrupt handler function type.

All interrupts handlers should be of this type, and follow normal ARM EABI register saving conventions

Function Documentation

◆ irq_add_shared_handler()

```
void irq_add_shared_handler ( uint          num,
                              irq_handler_t handler,
                              uint8_t       order_priority
                            )
```

Add a shared interrupt handler for an interrupt on the executing core.

Use this method to add a handler on an irq number shared between multiple distinct hardware sources (e.g. GPIO, DMA or PIO IRQs). Handlers added by this method will all be called in sequence from highest order_priority to lowest. The irq_set_exclusive_handler() method should be used instead if you know there will or should only ever be one handler for the interrupt.

This method will assert if there is an exclusive interrupt handler set for this irq number on this core, or if the (total across all IRQs on both cores) maximum (configurable via PICO_MAX_SHARED_IRQ_HANDLERS) number of shared handlers would be exceeded.

Parameters

num	Interrupt number Interrupt Numbers
handler	The handler to set. See irq_handler_t
order_priority	The order priority controls the order that handlers for the same IRQ number on the core are called. The shared irq handlers for an interrupt are all called when an IRQ fires, however the order of the calls is based on the order_priority (higher priorities are called first, identical priorities are called in undefined order). A good rule of thumb is to use PICO_SHARED_IRQ_HANDLER_DEFAULT_ORDER_PRIORITY if you don't much care, as it is in the middle of the priority range by default.

NOTE

The order_priority uses *higher* values for higher priorities which is the *opposite* of the CPU interrupt priorities passed to irq_set_priority() which use lower values for higher priorities.

See also irq_set_exclusive_handler()

◆ irq_clear()

static void irq_clear (uint ***int_num***) inline static

Clear a specific interrupt on the executing core.

This method is only useful for "software" IRQs that are not connected to hardware (i.e. IRQs 26-31) as the the NVIC always reflects the current state of the IRQ state of the hardware for hardware IRQs, and clearing of the IRQ state of the hardware is performed via the hardware's registers instead.

Parameters

int_num Interrupt number Interrupt Numbers

◆ irq_get_exclusive_handler()

irq_handler_t irq_get_exclusive_handler (uint ***num***)

Get the exclusive interrupt handler for an interrupt on the executing core.

This method will return an exclusive IRQ handler set on this core by irq_set_exclusive_handler if there is one.

Parameters

num Interrupt number Interrupt Numbers

See also irq_set_exclusive_handler()

Returns

handler The handler if an exclusive handler is set for the IRQ, NULL if no handler is set or shared/shareable handlers are installed

◆ irq_get_priority()

uint irq_get_priority (uint ***num***)

Get specified interrupt's priority.

Numerically-lower values indicate a higher priority. Hardware priorities range from 0 (highest priority) to 255 (lowest priority) though only the top 2 bits are significant on ARM Cortex-M0+. To make it easier to specify higher or lower priorities than the default, all IRQ priorities are initialized to PICO_DEFAULT_IRQ_PRIORITY by the SDK runtime at startup. PICO_DEFAULT_IRQ_PRIORITY defaults to 0x80

Parameters

num Interrupt number Interrupt Numbers

Returns

the IRQ priority

◆ irq_get_vtable_handler()

irq_handler_t irq_get_vtable_handler (uint ***num***)

Get the current IRQ handler for the specified IRQ from the currently installed hardware vector table (VTOR) of the execution core.

Parameters

num Interrupt number Interrupt Numbers

Returns

the address stored in the VTABLE for the given irq number

◆ irq_has_shared_handler()

bool irq_has_shared_handler (uint ***num***)

Determine if the current handler for the given number is shared.

Parameters

num Interrupt number Interrupt Numbers

Returns

true if the specified IRQ has a shared handler

◆ irq_is_enabled()

bool irq_is_enabled (uint ***num***)

Determine if a specific interrupt is enabled on the executing core.

Parameters

num Interrupt number Interrupt Numbers

Returns

true if the interrupt is enabled

◆ irq_remove_handler()

void irq_remove_handler (uint ***num***,
irq_handler_t ***handler***
)

Remove a specific interrupt handler for the given irq number on the executing core.

This method may be used to remove an irq set via either irq_set_exclusive_handler() or irq_add_shared_handler(), and will assert if the handler is not currently installed for the given IRQ number

NOTE

This method may *only* be called from user (non IRQ code) or from within the handler itself (i.e. an IRQ handler may remove itself as part of handling the IRQ). Attempts to call from another IRQ will cause an assertion.

Parameters

num Interrupt number Interrupt Numbers
handler The handler to removed.

See also irq_set_exclusive_handler() irq_add_shared_handler()

◆ irq_set_enabled()

```
void irq_set_enabled ( uint num,
                       bool enabled
                     )
```

Enable or disable a specific interrupt on the executing core.

Parameters

num	Interrupt number Interrupt Numbers
enabled	true to enable the interrupt, false to disable

◆ irq_set_exclusive_handler()

```
void irq_set_exclusive_handler ( uint num,
                                 irq_handler_t handler
                               )
```

Set an exclusive interrupt handler for an interrupt on the executing core.

Use this method to set a handler for single IRQ source interrupts, or when your code, use case or performance requirements dictate that there should no other handlers for the interrupt.

This method will assert if there is already any sort of interrupt handler installed for the specified irq number.

Parameters

num	Interrupt number Interrupt Numbers
handler	The handler to set. See irq_handler_t

See also irq_add_shared_handler()

◆ irq_set_mask_enabled()

```
void irq_set_mask_enabled ( uint32_t mask,
                            bool enabled
                          )
```

Enable/disable multiple interrupts on the executing core.

Parameters

mask	32-bit mask with one bits set for the interrupts to enable/disable Interrupt Numbers
enabled	true to enable the interrupts, false to disable them.

◆ irq_set_pending()

```
void irq_set_pending ( uint num )
```

Force an interrupt to be pending on the executing core.

This should generally not be used for IRQs connected to hardware.

Parameters

num Interrupt number Interrupt Numbers

◆ irq_set_priority()

```
void irq_set_priority ( uint     num,
                        uint8_t  hardware_priority
                      )
```

Set specified interrupt's priority.

Parameters

num	Interrupt number Interrupt Numbers
hardware_priority	Priority to set. Numerically-lower values indicate a higher priority. Hardware priorities range from 0 (highest priority) to 255 (lowest priority) though only the top 2 bits are significant on ARM Cortex-M0+. To make it easier to specify higher or lower priorities than the default, all IRQ priorities are initialized to PICO_DEFAULT_IRQ_PRIORITY by the SDK runtime at startup. PICO_DEFAULT_IRQ_PRIORITY defaults to 0x80

◆ user_irq_claim()

```
void user_irq_claim ( uint  irq_num )
```

Claim ownership of a user IRQ on the calling core.

User IRQs are numbered 26-31 and are not connected to any hardware, but can be triggered by irq_set_pending.

NOTE

> User IRQs are a core local feature; they cannot be used to communicate between cores. Therfore all functions dealing with Uer IRQs affect only the calling core

This method explicitly claims ownership of a user IRQ, so other code can know it is being used.

Parameters

irq_num the user IRQ to claim

◆ user_irq_claim_unused()

```
int user_irq_claim_unused ( bool  required )
```

Claim ownership of a free user IRQ on the calling core.

User IRQs are numbered 26-31 and are not connected to any hardware, but can be triggered by irq_set_pending.

NOTE

> User IRQs are a core local feature; they cannot be used to communicate between cores. Therfore all functions dealing with Uer IRQs affect only the calling core

This method explicitly claims ownership of an unused user IRQ if there is one, so other code can know it is being used.

Parameters

required if true the function will panic if none are available

Returns

the user IRQ number or -1 if required was false, and none were free

◆ **user_irq_unclaim()**

void user_irq_unclaim (uint ***irq_num***)

Mark a user IRQ as no longer used on the calling core.

User IRQs are numbered 26-31 and are not connected to any hardware, but can be triggered by irq_set_pending.

NOTE

User IRQs are a core local feature; they cannot be used to communicate between cores. Therfore all functions dealing with Uer IRQs affect only the calling core

This method explicitly releases ownership of a user IRQ, so other code can know it is free to use.

NOTE

it is customary to have disabled the irq and removed the handler prior to calling this method.

Parameters

irq_num the irq irq_num to unclaim

hardware_pio

Part of: Hardware APIs

Modules

- `sm_config`

  PIO state machine configuration.

- `pio_instructions`

  PIO instruction encoding.

Macros

- `#define pio0 pio0_hw`
- `#define pio1 pio1_hw`

Enumerations

- `enum pio_fifo_join { PIO_FIFO_JOIN_NONE = 0 , PIO_FIFO_JOIN_TX = 1 , PIO_FIFO_JOIN_RX = 2 }`

  FIFO join states.

- `enum pio_mov_status_type { STATUS_TX_LESSTHAN = 0 , STATUS_RX_LESSTHAN = 1 }`

  MOV status types.

- `enum pio_interrupt_source { pis_interrupt0 = PIO_INTR_SM0_LSB , pis_interrupt1 = PIO_INTR_SM1_LSB , pis_interrupt2`

```
= PIO_INTR_SM2_LSB , pis_interrupt3 = PIO_INTR_SM3_LSB ,
  pis_sm0_tx_fifo_not_full = PIO_INTR_SM0_TXNFULL_LSB , pis_sm1_tx_fifo_not_full =
PIO_INTR_SM1_TXNFULL_LSB , pis_sm2_tx_fifo_not_full = PIO_INTR_SM2_TXNFULL_LSB ,
pis_sm3_tx_fifo_not_full = PIO_INTR_SM3_TXNFULL_LSB ,
  pis_sm0_rx_fifo_not_empty = PIO_INTR_SM0_RXNEMPTY_LSB , pis_sm1_rx_fifo_not_empty =
PIO_INTR_SM1_RXNEMPTY_LSB , pis_sm2_rx_fifo_not_empty = PIO_INTR_SM2_RXNEMPTY_LSB ,
pis_sm3_rx_fifo_not_empty = PIO_INTR_SM3_RXNEMPTY_LSB
}
```

PIO interrupt source numbers for pio related IRQs.

Functions

- **static void pio_sm_set_config (PIO pio, uint sm, const pio_sm_config *config)**

  Apply a state machine configuration to a state machine.

- **static uint pio_get_index (PIO pio)**

  Return the instance number of a PIO instance.

- **static void pio_gpio_init (PIO pio, uint pin)**

  Setup the function select for a GPIO to use output from the given PIO instance.

- **static uint pio_get_dreq (PIO pio, uint sm, bool is_tx)**

  Return the DREQ to use for pacing transfers to/from a particular state machine FIFO.

- **bool pio_can_add_program (PIO pio, const pio_program_t *program)**

  Determine whether the given program can (at the time of the call) be loaded onto the PIO instance.

- **bool pio_can_add_program_at_offset (PIO pio, const pio_program_t *program, uint offset)**

  Determine whether the given program can (at the time of the call) be loaded onto the PIO instance starting at a particular location.

- **uint pio_add_program (PIO pio, const pio_program_t *program)**

  Attempt to load the program, panicking if not possible.

- **void pio_add_program_at_offset (PIO pio, const pio_program_t *program, uint offset)**

  Attempt to load the program at the specified instruction memory offset, panicking if not possible.

- **void pio_remove_program (PIO pio, const pio_program_t *program, uint loaded_offset)**

  Remove a program from a PIO instance's instruction memory.

- **void pio_clear_instruction_memory (PIO pio)**

  Clears all of a PIO instance's instruction memory.

- **void pio_sm_init (PIO pio, uint sm, uint initial_pc, const pio_sm_config *config)**

  Resets the state machine to a consistent state, and configures it.

- **static void pio_sm_set_enabled (PIO pio, uint sm, bool enabled)**

  Enable or disable a PIO state machine.

- **static void pio_set_sm_mask_enabled (PIO pio, uint32_t mask, bool enabled)**

  Enable or disable multiple PIO state machines.

- **static void pio_sm_restart (PIO pio, uint sm)**

  Restart a state machine with a known state.

- **static void pio_restart_sm_mask (PIO pio, uint32_t mask)**

  Restart multiple state machine with a known state.

- **static void pio_sm_clkdiv_restart (PIO pio, uint sm)**

  Restart a state machine's clock divider from a phase of 0.

- **static void pio_clkdiv_restart_sm_mask (PIO pio, uint32_t mask)**

  Restart multiple state machines' clock dividers from a phase of 0.

- **static void pio_enable_sm_mask_in_sync (PIO pio, uint32_t mask)**

  Enable multiple PIO state machines synchronizing their clock dividers.

- **static void pio_set_irq0_source_enabled (PIO pio, enum pio_interrupt_source source, bool enabled)**

  Enable/Disable a single source on a PIO's IRQ 0.

- **static void pio_set_irq1_source_enabled (PIO pio, enum pio_interrupt_source source, bool enabled)**

  Enable/Disable a single source on a PIO's IRQ 1.

- **static void pio_set_irq0_source_mask_enabled (PIO pio, uint32_t source_mask, bool enabled)**

  Enable/Disable multiple sources on a PIO's IRQ 0.

- **static void pio_set_irq1_source_mask_enabled (PIO pio, uint32_t source_mask, bool enabled)**

  Enable/Disable multiple sources on a PIO's IRQ 1.

- **static void pio_set_irqn_source_enabled (PIO pio, uint irq_index, enum pio_interrupt_source source, bool enabled)**

  Enable/Disable a single source on a PIO's specified (0/1) IRQ index.

- **static void pio_set_irqn_source_mask_enabled (PIO pio, uint irq_index, uint32_t source_mask, bool enabled)**

  Enable/Disable multiple sources on a PIO's specified (0/1) IRQ index.

- **static bool pio_interrupt_get (PIO pio, uint pio_interrupt_num)**

  Determine if a particular PIO interrupt is set.

- **static void pio_interrupt_clear (PIO pio, uint pio_interrupt_num)**

  Clear a particular PIO interrupt.

- **static uint8_t pio_sm_get_pc (PIO pio, uint sm)**

  Return the current program counter for a state machine.

- **static void pio_sm_exec (PIO pio, uint sm, uint instr)**

  Immediately execute an instruction on a state machine.

- **static bool pio_sm_is_exec_stalled (PIO pio, uint sm)**

  Determine if an instruction set by pio_sm_exec() is stalled executing.

- **static void pio_sm_exec_wait_blocking (PIO pio, uint sm, uint instr)**

  Immediately execute an instruction on a state machine and wait for it to complete.

- **static void pio_sm_set_wrap (PIO pio, uint sm, uint wrap_target, uint wrap)**

  Set the current wrap configuration for a state machine.

- **static void pio_sm_set_out_pins (PIO pio, uint sm, uint out_base, uint out_count)**

  Set the current 'out' pins for a state machine.

- **static void pio_sm_set_set_pins (PIO pio, uint sm, uint set_base, uint set_count)**

  Set the current 'set' pins for a state machine.

- **static void pio_sm_set_in_pins (PIO pio, uint sm, uint in_base)**

  Set the current 'in' pins for a state machine.

- **static void pio_sm_set_sideset_pins (PIO pio, uint sm, uint sideset_base)**

  Set the current 'sideset' pins for a state machine.

- **static void pio_sm_put (PIO pio, uint sm, uint32_t data)**

  Write a word of data to a state machine's TX FIFO.

- **static uint32_t pio_sm_get (PIO pio, uint sm)**

  Read a word of data from a state machine's RX FIFO.

- **static bool pio_sm_is_rx_fifo_full (PIO pio, uint sm)**

  Determine if a state machine's RX FIFO is full.

- **static bool pio_sm_is_rx_fifo_empty (PIO pio, uint sm)**

  Determine if a state machine's RX FIFO is empty.

- **static uint pio_sm_get_rx_fifo_level (PIO pio, uint sm)**

  Return the number of elements currently in a state machine's RX FIFO.

- **static bool pio_sm_is_tx_fifo_full (PIO pio, uint sm)**

  Determine if a state machine's TX FIFO is full.

- **static bool pio_sm_is_tx_fifo_empty (PIO pio, uint sm)**

  Determine if a state machine's TX FIFO is empty.

- **static uint pio_sm_get_tx_fifo_level (PIO pio, uint sm)**

  Return the number of elements currently in a state machine's TX FIFO.

- **static void pio_sm_put_blocking (PIO pio, uint sm, uint32_t data)**

  Write a word of data to a state machine's TX FIFO, blocking if the FIFO is full.

- **static uint32_t pio_sm_get_blocking (PIO pio, uint sm)**

  Read a word of data from a state machine's RX FIFO, blocking if the FIFO is empty.

- **void pio_sm_drain_tx_fifo (PIO pio, uint sm)**

  Empty out a state machine's TX FIFO.

- **static void pio_sm_set_clkdiv_int_frac (PIO pio, uint sm, uint16_t div_int, uint8_t div_frac)**

  set the current clock divider for a state machine using a 16:8 fraction

- **static void pio_sm_set_clkdiv (PIO pio, uint sm, float div)**

  set the current clock divider for a state machine

- **static void pio_sm_clear_fifos (PIO pio, uint sm)**

  Clear a state machine's TX and RX FIFOs.

- **void pio_sm_set_pins (PIO pio, uint sm, uint32_t pin_values)**

  Use a state machine to set a value on all pins for the PIO instance.

- **void pio_sm_set_pins_with_mask (PIO pio, uint sm, uint32_t pin_values, uint32_t pin_mask)**

  Use a state machine to set a value on multiple pins for the PIO instance.

- **void pio_sm_set_pindirs_with_mask (PIO pio, uint sm, uint32_t pin_dirs, uint32_t pin_mask)**

  Use a state machine to set the pin directions for multiple pins for the PIO instance.

- **void pio_sm_set_consecutive_pindirs (PIO pio, uint sm, uint pin_base, uint pin_count, bool is_out)**

  Use a state machine to set the same pin direction for multiple consecutive pins for the PIO instance.

- **void pio_sm_claim (PIO pio, uint sm)**

  Mark a state machine as used.

- **void pio_claim_sm_mask (PIO pio, uint sm_mask)**

  Mark multiple state machines as used.

- **void pio_sm_unclaim (PIO pio, uint sm)**

  Mark a state machine as no longer used.

- **int pio_claim_unused_sm (PIO pio, bool required)**

  Claim a free state machine on a PIO instance.

- **bool pio_sm_is_claimed (PIO pio, uint sm)**

  Determine if a PIO state machine is claimed.

Detailed Description

Programmable I/O (PIO) API

A programmable input/output block (PIO) is a versatile hardware interface which can support a number of different IO standards. There are two PIO blocks in the RP2040.

Each PIO is programmable in the same sense as a processor: the four state machines independently execute short, sequential programs, to manipulate GPIOs and transfer data. Unlike a general purpose processor, PIO state machines are highly specialised for IO, with a focus on determinism, precise timing, and close integration with fixed-function hardware. Each state machine is equipped with:

- Two 32-bit shift registers – either direction, any shift count
- Two 32-bit scratch registers
- 4×32 bit bus FIFO in each direction (TX/RX), reconfigurable as 8×32 in a single direction
- Fractional clock divider (16 integer, 8 fractional bits)
- Flexible GPIO mapping

- DMA interface, sustained throughput up to 1 word per clock from system DMA
- IRQ flag set/clear/status

Full details of the PIO can be found in the RP2040 datasheet.

Macro Definition Documentation

◆ pio0

```
#define pio0   pio0_hw
```

Identifier for the first (PIO 0) hardware PIO instance (for use in PIO functions).

e.g. pio_gpio_init(pio0, 5)

◆ pio1

```
#define pio1   pio1_hw
```

Identifier for the second (PIO 1) hardware PIO instance (for use in PIO functions).

e.g. pio_gpio_init(pio1, 5)

Function Documentation

◆ pio_add_program()

```
uint pio_add_program ( PIO                        pio,
                       const pio_program_t *      program
                     )
```

Attempt to load the program, panicking if not possible.

See also pio_can_add_program() if you need to check whether the program can be loaded

Parameters

pio	The PIO instance; either pio0 or pio1
program	the program definition

Returns

the instruction memory offset the program is loaded at

◆ pio_add_program_at_offset()

```
void pio_add_program_at_offset ( PIO                        pio,
                                 const pio_program_t *      program,
                                 uint                       offset
                               )
```

Attempt to load the program at the specified instruction memory offset, panicking if not possible.

See also pio_can_add_program_at_offset() if you need to check whether the program can be loaded

Parameters

pio	The PIO instance; either pio0 or pio1
program	the program definition
offset	the instruction memory offset wanted for the start of the program

◆ pio_can_add_program()

```
bool pio_can_add_program ( PIO                        pio,
                           const pio_program_t *      program
                         )
```

Determine whether the given program can (at the time of the call) be loaded onto the PIO instance.

Parameters

pio	The PIO instance; either pio0 or pio1
program	the program definition

Returns

true if the program can be loaded; false if there is not suitable space in the instruction memory

◆ pio_can_add_program_at_offset()

```
bool pio_can_add_program_at_offset ( PIO                        pio,
                                     const pio_program_t *      program,
                                     uint                       offset
                                   )
```

Determine whether the given program can (at the time of the call) be loaded onto the PIO instance starting at a particular location.

Parameters

pio	The PIO instance; either pio0 or pio1
program	the program definition
offset	the instruction memory offset wanted for the start of the program

Returns

true if the program can be loaded at that location; false if there is not space in the instruction memory

◆ pio_claim_sm_mask()

```
void pio_claim_sm_mask ( PIO  pio,
                         uint  sm_mask
                       )
```

Mark multiple state machines as used.

Method for cooperative claiming of hardware. Will cause a panic if any of the state machines are already claimed. Use of this method by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters

pio	The PIO instance; either pio0 or pio1

sm_mask Mask of state machine indexes

◆ pio_claim_unused_sm()

```
int pio_claim_unused_sm ( PIO  pio,
                          bool  required
                        )
```

Claim a free state machine on a PIO instance.

Parameters

pio The PIO instance; either pio0 or pio1
required if true the function will panic if none are available

Returns

the state machine index or -1 if required was false, and none were free

◆ pio_clear_instruction_memory()

```
void pio_clear_instruction_memory ( PIO  pio )
```

Clears all of a PIO instance's instruction memory.

Parameters

pio The PIO instance; either pio0 or pio1

◆ pio_clkdiv_restart_sm_mask()

```
static void pio_clkdiv_restart_sm_mask ( PIO      pio,
                                         uint32_t mask
                                       )                    inline  static
```

Restart multiple state machines' clock dividers from a phase of 0.

Each state machine's clock divider is a free-running piece of hardware, that generates a pattern of clock enable pulses for the state machine, based *only* on the configured integer/fractional divisor. The pattern of running/halted cycles slows the state machine's execution to some controlled rate.

This function simultaneously clears the integer and fractional phase accumulators of multiple state machines' clock dividers. If these state machines all have the same integer and fractional divisors configured, their clock dividers will run in precise deterministic lockstep from this point.

With their execution clocks synchronised in this way, it is then safe to e.g. have multiple state machines performing a 'wait irq' on the same flag, and all clear it on the same cycle.

Also note that this function can be called whilst state machines are running (e.g. if you have just changed the clock divisors of some state machines and wish to resynchronise them), and that disabling a state machine does not halt its clock divider: that is, if multiple state machines have their clocks synchronised, you can safely disable and reenable one of the state machines without losing synchronisation.

Parameters

pio The PIO instance; either pio0 or pio1
mask bit mask of state machine indexes to modify the enabled state of

◆ pio_enable_sm_mask_in_sync()

```
static void pio_enable_sm_mask_in_sync ( PIO      pio,
                                         uint32_t mask
                                       )
```

inline static

Enable multiple PIO state machines synchronizing their clock dividers.

This is equivalent to calling both pio_set_sm_mask_enabled() and pio_clkdiv_restart_sm_mask() on the *same* clock cycle. All state machines specified by 'mask' are started simultaneously and, assuming they have the same clock divisors, their divided clocks will stay precisely synchronised.

Parameters

pio	The PIO instance; either pio0 or pio1
mask	bit mask of state machine indexes to modify the enabled state of

◆ pio_get_dreq()

```
static uint pio_get_dreq ( PIO  pio,
                           uint sm,
                           bool is_tx
                         )
```

inline static

Return the DREQ to use for pacing transfers to/from a particular state machine FIFO.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)
is_tx	true for sending data to the state machine, false for receiving data from the state machine

◆ pio_get_index()

```
static uint pio_get_index ( PIO pio )
```

inline static

Return the instance number of a PIO instance.

Parameters

pio	The PIO instance; either pio0 or pio1

Returns

the PIO instance number (either 0 or 1)

◆ pio_gpio_init()

```
static void pio_gpio_init ( PIO  pio,
                            uint pin
                          )
```

inline static

Setup the function select for a GPIO to use output from the given PIO instance.

PIO appears as an alternate function in the GPIO muxing, just like an SPI or UART. This function configures that multiplexing to connect a given PIO instance to a GPIO. Note that this is not necessary for a state machine to be able to read the *input* value from a GPIO, but only for it to set the output value or output enable.

Parameters

pio	The PIO instance; either pio0 or pio1
pin	the GPIO pin whose function select to set

◆ pio_interrupt_clear()

```
static void pio_interrupt_clear ( PIO  pio,
                                  uint  pio_interrupt_num
                                )                                    inline  static
```

Clear a particular PIO interrupt.

Parameters

pio	The PIO instance; either pio0 or pio1
pio_interrupt_num	the PIO interrupt number 0-7

◆ pio_interrupt_get()

```
static bool pio_interrupt_get ( PIO  pio,
                                uint  pio_interrupt_num
                              )                                      inline  static
```

Determine if a particular PIO interrupt is set.

Parameters

pio	The PIO instance; either pio0 or pio1
pio_interrupt_num	the PIO interrupt number 0-7

Returns

true if corresponding PIO interrupt is currently set

◆ pio_remove_program()

```
void pio_remove_program ( PIO                     pio,
                          const pio_program_t *   program,
                          uint                    loaded_offset
                        )
```

Remove a program from a PIO instance's instruction memory.

Parameters

pio	The PIO instance; either pio0 or pio1
program	the program definition
loaded_offset	the loaded offset returned when the program was added

◆ pio_restart_sm_mask()

```
static void pio_restart_sm_mask ( PIO      pio,
                                  uint32_t mask
                                )                                    inline static
```

Restart multiple state machine with a known state.

This method clears the ISR, shift counters, clock divider counter pin write flags, delay counter, latched EXEC instruction, and IRQ wait condition.

Parameters

pio	The PIO instance; either pio0 or pio1
mask	bit mask of state machine indexes to modify the enabled state of

◆ pio_set_irq0_source_enabled()

```
static void pio_set_irq0_source_enabled ( PIO                      pio,
                                          enum pio_interrupt_source source,
                                          bool                     enabled
                                        )                            inline static
```

Enable/Disable a single source on a PIO's IRQ 0.

Parameters

pio	The PIO instance; either pio0 or pio1
source	the source number (see pio_interrupt_source)
enabled	true to enable IRQ 0 for the source, false to disable.

◆ pio_set_irq0_source_mask_enabled()

```
static void pio_set_irq0_source_mask_enabled ( PIO      pio,
                                               uint32_t source_mask,
                                               bool     enabled
                                             )                       inline static
```

Enable/Disable multiple sources on a PIO's IRQ 0.

Parameters

pio	The PIO instance; either pio0 or pio1
source_mask	Mask of bits, one for each source number (see pio_interrupt_source) to affect
enabled	true to enable all the sources specified in the mask on IRQ 0, false to disable all the sources specified in the mask on IRQ 0

◆ pio_set_irq1_source_enabled()

```
static void pio_set_irq1_source_enabled ( PIO                      pio,
                                          enum pio_interrupt_source source,
                                          bool                     enabled
                                        )                            inline static
```

Enable/Disable a single source on a PIO's IRQ 1.

Parameters

pio	The PIO instance; either pio0 or pio1
source	the source number (see pio_interrupt_source)
enabled	true to enable IRQ 0 for the source, false to disable.

◆ pio_set_irq1_source_mask_enabled()

```
static void pio_set_irq1_source_mask_enabled ( PIO      pio,
                                               uint32_t source_mask,
                                               bool     enabled
                                             )                        inline  static
```

Enable/Disable multiple sources on a PIO's IRQ 1.

Parameters

pio	The PIO instance; either pio0 or pio1
source_mask	Mask of bits, one for each source number (see pio_interrupt_source) to affect
enabled	true to enable all the sources specified in the mask on IRQ 1, false to disable all the source specified in the mask on IRQ 1

◆ pio_set_irqn_source_enabled()

```
static void pio_set_irqn_source_enabled ( PIO                      pio,
                                          uint                     irq_index,
                                          enum pio_interrupt_source source,
                                          bool                     enabled
                                        )                          inline  static
```

Enable/Disable a single source on a PIO's specified (0/1) IRQ index.

Parameters

pio	The PIO instance; either pio0 or pio1
irq_index	the IRQ index; either 0 or 1
source	the source number (see pio_interrupt_source)
enabled	true to enable the source on the specified IRQ, false to disable.

◆ pio_set_irqn_source_mask_enabled()

```
static void pio_set_irqn_source_mask_enabled ( PIO      pio,
                                               uint     irq_index,
                                               uint32_t source_mask,
                                               bool     enabled
                                             )                        inline  static
```

Enable/Disable multiple sources on a PIO's specified (0/1) IRQ index.

Parameters

pio	The PIO instance; either pio0 or pio1
irq_index	the IRQ index; either 0 or 1
source_mask	Mask of bits, one for each source number (see pio_interrupt_source) to affect
enabled	true to enable all the sources specified in the mask on the specified IRQ, false to disable all the sources specified in the mask on the specified IRQ

◆ pio_set_sm_mask_enabled()

```
static void pio_set_sm_mask_enabled ( PIO      pio,
                                      uint32_t mask,
                                      bool     enabled
                                    )
```
inline static

Enable or disable multiple PIO state machines.

Note that this method just sets the enabled state of the state machine; if now enabled they continue exactly from where they left off.

See also pio_enable_sm_mask_in_sync() if you wish to enable multiple state machines and ensure their clock dividers are in sync.

Parameters

pio	The PIO instance; either pio0 or pio1
mask	bit mask of state machine indexes to modify the enabled state of
enabled	true to enable the state machines; false to disable

◆ pio_sm_claim()

```
void pio_sm_claim ( PIO  pio,
                    uint sm
                  )
```

Mark a state machine as used.

Method for cooperative claiming of hardware. Will cause a panic if the state machine is already claimed. Use of this method by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)

◆ pio_sm_clear_fifos()

```
static void pio_sm_clear_fifos ( PIO  pio,
                                 uint sm
                               )
```
inline static

Clear a state machine's TX and RX FIFOs.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)

◆ pio_sm_clkdiv_restart()

```
static void pio_sm_clkdiv_restart ( PIO  pio,
                                    uint sm
                                  )
```
inline static

Restart a state machine's clock divider from a phase of 0.

Each state machine's clock divider is a free-running piece of hardware, that generates a pattern of clock enable pulses for the state machine, based *only* on the configured integer/fractional divisor. The pattern of running/halted cycles slows the state machine's execution to some controlled rate.

This function clears the divider's integer and fractional phase accumulators so that it restarts this pattern from the beginning. It is called automatically by pio_sm_init() but can also be called at a later time, when you enable the state machine, to ensure precisely consistent timing each time you load and run a given PIO program.

More commonly this hardware mechanism is used to synchronise the execution clocks of multiple state machines – see pio_clkdiv_restart_sm_mask().

Parameters

pio The PIO instance; either pio0 or pio1
sm State machine index (0..3)

◆ pio_sm_drain_tx_fifo()

```
void pio_sm_drain_tx_fifo ( PIO pio,
                            uint sm
                          )
```

Empty out a state machine's TX FIFO.

This method executes `pull` instructions on the state machine until the TX FIFO is empty. This disturbs the contents of the OSR, so see also pio_sm_clear_fifos() which clears both FIFOs but leaves the state machine's internal state undisturbed.

Parameters

pio The PIO instance; either pio0 or pio1
sm State machine index (0..3)

See also pio_sm_clear_fifos()

◆ pio_sm_exec()

```
static void pio_sm_exec ( PIO pio,
                          uint sm,
                          uint instr
                        )                                    inline static
```

Immediately execute an instruction on a state machine.

This instruction is executed instead of the next instruction in the normal control flow on the state machine. Subsequent calls to this method replace the previous executed instruction if it is still running.

See also pio_sm_is_exec_stalled() to see if an executed instruction is still running (i.e. it is stalled on some condition)

Parameters

pio The PIO instance; either pio0 or pio1
sm State machine index (0..3)
instr the encoded PIO instruction

◆ pio_sm_exec_wait_blocking()

```
static void pio_sm_exec_wait_blocking ( PIO pio,
                                        uint sm,
```

uint *instr*
)

inline static

Immediately execute an instruction on a state machine and wait for it to complete.

This instruction is executed instead of the next instruction in the normal control flow on the state machine. Subsequent calls to this method replace the previous executed instruction if it is still running.

See also pio_sm_is_exec_stalled() to see if an executed instruction is still running (i.e. it is stalled on some condition)

Parameters

pio The PIO instance; either pio0 or pio1
sm State machine index (0..3)
instr the encoded PIO instruction

◆ pio_sm_get()

static uint32_t pio_sm_get (PIO *pio*,
uint *sm*
)

inline static

Read a word of data from a state machine's RX FIFO.

This is a raw FIFO access that does not check for emptiness. If the FIFO is empty, the hardware ignores the attempt to read from the FIFO (the FIFO remains in an empty state following the read) and the sticky RXUNDER flag for this FIFO is set in FDEBUG to indicate that the system tried to read from this FIFO when empty. The data returned by this function is undefined when the FIFO is empty.

Parameters

pio The PIO instance; either pio0 or pio1
sm State machine index (0..3)

See also pio_sm_get_blocking()

◆ pio_sm_get_blocking()

static uint32_t pio_sm_get_blocking (PIO *pio*,
uint *sm*
)

inline static

Read a word of data from a state machine's RX FIFO, blocking if the FIFO is empty.

Parameters

pio The PIO instance; either pio0 or pio1
sm State machine index (0..3)

◆ pio_sm_get_pc()

static uint8_t pio_sm_get_pc (PIO *pio*,
uint *sm*
)

inline static

Return the current program counter for a state machine.

Parameters

pio The PIO instance; either pio0 or pio1
sm State machine index (0..3)

Returns

the program counter

◆ pio_sm_get_rx_fifo_level()

```
static uint pio_sm_get_rx_fifo_level ( PIO  pio,
                                       uint  sm
                                     )                                    inline  static
```

Return the number of elements currently in a state machine's RX FIFO.

Parameters

pio The PIO instance; either pio0 or pio1
sm State machine index (0..3)

Returns

the number of elements in the RX FIFO

◆ pio_sm_get_tx_fifo_level()

```
static uint pio_sm_get_tx_fifo_level ( PIO  pio,
                                       uint  sm
                                     )                                    inline  static
```

Return the number of elements currently in a state machine's TX FIFO.

Parameters

pio The PIO instance; either pio0 or pio1
sm State machine index (0..3)

Returns

the number of elements in the TX FIFO

◆ pio_sm_init()

```
void pio_sm_init ( PIO                    pio,
                   uint                   sm,
                   uint                   initial_pc,
                   const pio_sm_config *  config
                 )
```

Resets the state machine to a consistent state, and configures it.

This method:

- Disables the state machine (if running)
- Clears the FIFOs

- Applies the configuration specified by 'config'
- Resets any internal state e.g. shift counters
- Jumps to the initial program location given by 'initial_pc'

The state machine is left disabled on return from this call.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)
initial_pc	the initial program memory offset to run from
config	the configuration to apply (or NULL to apply defaults)

◆ pio_sm_is_claimed()

```
bool pio_sm_is_claimed ( PIO pio,
                         uint sm
                       )
```

Determine if a PIO state machine is claimed.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)

Returns

true if claimed, false otherwise

See also pio_sm_claim pio_claim_sm_mask

◆ pio_sm_is_exec_stalled()

```
static bool pio_sm_is_exec_stalled ( PIO pio,
                                     uint sm
                                   )                    inline  static
```

Determine if an instruction set by pio_sm_exec() is stalled executing.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)

Returns

true if the executed instruction is still running (stalled)

◆ pio_sm_is_rx_fifo_empty()

```
static bool pio_sm_is_rx_fifo_empty ( PIO pio,
                                      uint sm
                                    )                   inline  static
```

Determine if a state machine's RX FIFO is empty.

Parameters

pio The PIO instance; either pio0 or pio1
sm State machine index (0..3)

Returns

true if the RX FIFO is empty

◆ pio_sm_is_rx_fifo_full()

```
static bool pio_sm_is_rx_fifo_full ( PIO pio,
                                     uint sm
                                   )                                    inline static
```

Determine if a state machine's RX FIFO is full.

Parameters

pio The PIO instance; either pio0 or pio1
sm State machine index (0..3)

Returns

true if the RX FIFO is full

◆ pio_sm_is_tx_fifo_empty()

```
static bool pio_sm_is_tx_fifo_empty ( PIO pio,
                                      uint sm
                                    )                                   inline static
```

Determine if a state machine's TX FIFO is empty.

Parameters

pio The PIO instance; either pio0 or pio1
sm State machine index (0..3)

Returns

true if the TX FIFO is empty

◆ pio_sm_is_tx_fifo_full()

```
static bool pio_sm_is_tx_fifo_full ( PIO pio,
                                     uint sm
                                   )                                    inline static
```

Determine if a state machine's TX FIFO is full.

Parameters

pio The PIO instance; either pio0 or pio1
sm State machine index (0..3)

Returns

true if the TX FIFO is full

◆ pio_sm_put()

```
static void pio_sm_put ( PIO      pio,
                         uint     sm,
                         uint32_t data
                       )                                    inline  static
```

Write a word of data to a state machine's TX FIFO.

This is a raw FIFO access that does not check for fullness. If the FIFO is full, the FIFO contents and state are not affected by the write attempt. Hardware sets the TXOVER sticky flag for this FIFO in FDEBUG, to indicate that the system attempted to write to a full FIFO.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)
data	the 32 bit data value

See also pio_sm_put_blocking()

◆ pio_sm_put_blocking()

```
static void pio_sm_put_blocking ( PIO      pio,
                                  uint     sm,
                                  uint32_t data
                                )                           inline  static
```

Write a word of data to a state machine's TX FIFO, blocking if the FIFO is full.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)
data	the 32 bit data value

◆ pio_sm_restart()

```
static void pio_sm_restart ( PIO  pio,
                             uint sm
                           )                                inline  static
```

Restart a state machine with a known state.

This method clears the ISR, shift counters, clock divider counter pin write flags, delay counter, latched EXEC instruction, and IRQ wait condition.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)

◆ pio_sm_set_clkdiv()

```
static void pio_sm_set_clkdiv ( PIO   pio,
                                uint  sm,
                                float div
                              )                                    inline  static
```

set the current clock divider for a state machine

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)
div	the floating point clock divider

◆ pio_sm_set_clkdiv_int_frac()

```
static void pio_sm_set_clkdiv_int_frac ( PIO      pio,
                                         uint     sm,
                                         uint16_t div_int,
                                         uint8_t  div_frac
                                       )                           inline  static
```

set the current clock divider for a state machine using a 16:8 fraction

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)
div_int	the integer part of the clock divider
div_frac	the fractional part of the clock divider in 1/256s

◆ pio_sm_set_config()

```
static void pio_sm_set_config ( PIO                  pio,
                                uint                 sm,
                                const pio_sm_config * config
                              )                                    inline  static
```

Apply a state machine configuration to a state machine.

Parameters

pio	Handle to PIO instance; either pio0 or pio1
sm	State machine index (0..3)
config	the configuration to apply

◆ pio_sm_set_consecutive_pindirs()

```
void pio_sm_set_consecutive_pindirs ( PIO  pio,
                                      uint sm,
                                      uint pin_base,
                                      uint pin_count,
                                      bool is_out
                                    )
```

Use a state machine to set the same pin direction for multiple consecutive pins for the PIO instance.

This method repeatedly reconfigures the target state machine's pin configuration and executes 'set' instructions to set the pin direction on consecutive pins, before restoring the state machine's pin configuration to what it was.

This method is provided as a convenience to set initial pin directions, and should not be used against a state machine that is enabled.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3) to use
pin_base	the first pin to set a direction for
pin_count	the count of consecutive pins to set the direction for
is_out	the direction to set; true = out, false = in

◆ pio_sm_set_enabled()

```
static void pio_sm_set_enabled ( PIO  pio,
                                 uint  sm,
                                 bool  enabled
                               )                                    inline  static
```

Enable or disable a PIO state machine.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)
enabled	true to enable the state machine; false to disable

◆ pio_sm_set_in_pins()

```
static void pio_sm_set_in_pins ( PIO  pio,
                                 uint  sm,
                                 uint  in_base
                               )                                    inline  static
```

Set the current 'in' pins for a state machine.

Can overlap with the 'out', 'set' and 'sideset' pins

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)
in_base	0-31 First pin to use as input

◆ pio_sm_set_out_pins()

```
static void pio_sm_set_out_pins ( PIO  pio,
                                  uint  sm,
                                  uint  out_base,
                                  uint  out_count
                                )                                   inline  static
```

Set the current 'out' pins for a state machine.

Can overlap with the 'in', 'set' and 'sideset' pins

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)
out_base	0-31 First pin to set as output
out_count	0-32 Number of pins to set.

◆ pio_sm_set_pindirs_with_mask()

```
void pio_sm_set_pindirs_with_mask ( PIO      pio,
                                    uint     sm,
                                    uint32_t pin_dirs,
                                    uint32_t pin_mask
                                  )
```

Use a state machine to set the pin directions for multiple pins for the PIO instance.

This method repeatedly reconfigures the target state machine's pin configuration and executes 'set' instructions to set pin directions on up to 32 pins, before restoring the state machine's pin configuration to what it was.

This method is provided as a convenience to set initial pin directions, and should not be used against a state machine that is enabled.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3) to use
pin_dirs	the pin directions to set - 1 = out, 0 = in (if the corresponding bit in pin_mask is set)
pin_mask	a bit for each pin to indicate whether the corresponding pin_value for that pin should be applied.

◆ pio_sm_set_pins()

```
void pio_sm_set_pins ( PIO      pio,
                       uint     sm,
                       uint32_t pin_values
                     )
```

Use a state machine to set a value on all pins for the PIO instance.

This method repeatedly reconfigures the target state machine's pin configuration and executes 'set' instructions to set values on all 32 pins, before restoring the state machine's pin configuration to what it was.

This method is provided as a convenience to set initial pin states, and should not be used against a state machine that is enabled.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3) to use
pin_values	the pin values to set

◆ pio_sm_set_pins_with_mask()

```
void pio_sm_set_pins_with_mask ( PIO      pio,
                                 uint     sm,
                                 uint32_t pin_values,
                                 uint32_t pin_mask
                               )
```

Use a state machine to set a value on multiple pins for the PIO instance.

This method repeatedly reconfigures the target state machine's pin configuration and executes 'set' instructions to set values on up to 32 pins, before restoring the state machine's pin configuration to what it was.

This method is provided as a convenience to set initial pin states, and should not be used against a state machine that is enabled.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3) to use
pin_values	the pin values to set (if the corresponding bit in pin_mask is set)
pin_mask	a bit for each pin to indicate whether the corresponding pin_value for that pin should be applied.

◆ pio_sm_set_set_pins()

```
static void pio_sm_set_set_pins ( PIO  pio,
                                  uint  sm,
                                  uint  set_base,
                                  uint  set_count
                                )                                    inline  static
```

Set the current 'set' pins for a state machine.

Can overlap with the 'in', 'out' and 'sideset' pins

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)
set_base	0-31 First pin to set as
set_count	0-5 Number of pins to set.

◆ pio_sm_set_sideset_pins()

```
static void pio_sm_set_sideset_pins ( PIO  pio,
                                      uint  sm,
                                      uint  sideset_base
                                    )                                inline  static
```

Set the current 'sideset' pins for a state machine.

Can overlap with the 'in', 'out' and 'set' pins

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)
sideset_base	0-31 base pin for 'side set'

◆ pio_sm_set_wrap()

```
static void pio_sm_set_wrap ( PIO  pio,
                              uint  sm,
                              uint  wrap_target,
                              uint  wrap
                            )                                        inline  static
```

Set the current wrap configuration for a state machine.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)
wrap_target	the instruction memory address to wrap to
wrap	the instruction memory address after which to set the program counter to wrap_target if the instruction does not itself update the program_counter

◆ **pio_sm_unclaim()**

```
void pio_sm_unclaim ( PIO pio,
                      uint sm
                    )
```

Mark a state machine as no longer used.

Method for cooperative claiming of hardware.

Parameters

pio	The PIO instance; either pio0 or pio1
sm	State machine index (0..3)

sm_config

Part of: Hardware APIs » hardware_pio
PIO state machine configuration. More...

Data Structures

- **struct pio_sm_config**

  PIO Configuration structure. More...

Functions

- **static void sm_config_set_out_pins (pio_sm_config *c, uint out_base, uint out_count)**

  Set the 'out' pins in a state machine configuration.

- **static void sm_config_set_set_pins (pio_sm_config *c, uint set_base, uint set_count)**

  Set the 'set' pins in a state machine configuration.

- **static void sm_config_set_in_pins (pio_sm_config *c, uint in_base)**

  Set the 'in' pins in a state machine configuration.

- **static void sm_config_set_sideset_pins (pio_sm_config *c, uint sideset_base)**

  Set the 'sideset' pins in a state machine configuration.

- **static void sm_config_set_sideset (pio_sm_config *c, uint bit_count, bool optional, bool pindirs)**

  Set the 'sideset' options in a state machine configuration.

- **static void sm_config_set_clkdiv_int_frac (pio_sm_config *c, uint16_t div_int, uint8_t div_frac)**

  Set the state machine clock divider (from integer and fractional parts - 16:8) in a state machine configuration.

- **static void sm_config_set_clkdiv (pio_sm_config *c, float div)**

Set the state machine clock divider (from a floating point value) in a state machine configuration.

- **static void sm_config_set_wrap (pio_sm_config *c, uint wrap_target, uint wrap)**

  Set the wrap addresses in a state machine configuration.

- **static void sm_config_set_jmp_pin (pio_sm_config *c, uint pin)**

  Set the 'jmp' pin in a state machine configuration.

- **static void sm_config_set_in_shift (pio_sm_config *c, bool shift_right, bool autopush, uint push_threshold)**

  Setup 'in' shifting parameters in a state machine configuration.

- **static void sm_config_set_out_shift (pio_sm_config *c, bool shift_right, bool autopull, uint pull_threshold)**

  Setup 'out' shifting parameters in a state machine configuration.

- **static void sm_config_set_fifo_join (pio_sm_config *c, enum pio_fifo_join join)**

  Setup the FIFO joining in a state machine configuration.

- **static void sm_config_set_out_special (pio_sm_config *c, bool sticky, bool has_enable_pin, uint enable_pin_index)**

  Set special 'out' operations in a state machine configuration.

- **static void sm_config_set_mov_status (pio_sm_config *c, enum pio_mov_status_type status_sel, uint status_n)**

  Set source for 'mov status' in a state machine configuration.

- **static pio_sm_config pio_get_default_sm_config (void)**

  Get the default state machine configuration.

Detailed Description

PIO state machine configuration.

A PIO block needs to be configured, these functions provide helpers to set up configuration structures. See pio_sm_set_config

Function Documentation

◆ pio_get_default_sm_config()

static pio_sm_config pio_get_default_sm_config (void) inline static

Get the default state machine configuration.

Setting	Default
Out Pins	32 starting at 0
Set Pins	0 starting at 0
In Pins (base)	0
Side Set Pins (base)	0
Side Set	disabled
Wrap	wrap=31, wrap_to=0
In Shift	shift_direction=right, autopush=false, push_threshold=32
Out Shift	shift_direction=right, autopull=false, pull_threshold=32
Jmp Pin	0

Out Special	sticky=false, has_enable_pin=false, enable_pin_index=0
Mov Status	status_sel=STATUS_TX_LESSTHAN, n=0

Returns

the default state machine configuration which can then be modified.

◆ sm_config_set_clkdiv()

```
static void sm_config_set_clkdiv ( pio_sm_config *  c,
                                   float            div
                                 )                                    inline  static
```

Set the state machine clock divider (from a floating point value) in a state machine configuration.

The clock divider slows the state machine's execution by masking the system clock on some cycles, in a repeating pattern, so that the state machine does not advance. Effectively this produces a slower clock for the state machine to run from, which can be used to generate e.g. a particular UART baud rate. See the datasheet for further detail.

Parameters

c	Pointer to the configuration structure to modify
div	The fractional divisor to be set. 1 for full speed. An integer clock divisor of n will cause the state machine to run 1 cycle in every n. Note that for small n, the jitter introduced by a fractional divider (e.g. 2.5) may be unacceptable although it will depend on the use case.

◆ sm_config_set_clkdiv_int_frac()

```
static void sm_config_set_clkdiv_int_frac ( pio_sm_config *  c,
                                            uint16_t         div_int,
                                            uint8_t          div_frac
                                          )                           inline  static
```

Set the state machine clock divider (from integer and fractional parts - 16:8) in a state machine configuration.

The clock divider can slow the state machine's execution to some rate below the system clock frequency, by enabling the state machine on some cycles but not on others, in a regular pattern. This can be used to generate e.g. a given UART baud rate. See the datasheet for further detail.

Parameters

c	Pointer to the configuration structure to modify
div_int	Integer part of the divisor
div_frac	Fractional part in 1/256ths

See also sm_config_set_clkdiv()

◆ sm_config_set_fifo_join()

```
static void sm_config_set_fifo_join ( pio_sm_config *     c,
                                      enum pio_fifo_join  join
                                    )                                 inline  static
```

Setup the FIFO joining in a state machine configuration.

Parameters

c	Pointer to the configuration structure to modify
join	Specifies the join type.

See also enum pio_fifo_join

◆ **sm_config_set_in_pins()**

```
static void sm_config_set_in_pins ( pio_sm_config *  c,
                                    uint             in_base
                                  )                                  inline static
```

Set the 'in' pins in a state machine configuration.

Can overlap with the 'out', 'set' and 'sideset' pins

Parameters

c	Pointer to the configuration structure to modify
in_base	0-31 First pin to use as input

◆ **sm_config_set_in_shift()**

```
static void sm_config_set_in_shift ( pio_sm_config *  c,
                                     bool             shift_right,
                                     bool             autopush,
                                     uint             push_threshold
                                   )                                 inline static
```

Setup 'in' shifting parameters in a state machine configuration.

Parameters

c	Pointer to the configuration structure to modify
shift_right	true to shift ISR to right, false to shift ISR to left
autopush	whether autopush is enabled
push_threshold	threshold in bits to shift in before auto/conditional re-pushing of the ISR

◆ **sm_config_set_jmp_pin()**

```
static void sm_config_set_jmp_pin ( pio_sm_config *  c,
                                    uint             pin
                                  )                                  inline static
```

Set the 'jmp' pin in a state machine configuration.

Parameters

c	Pointer to the configuration structure to modify
pin	The raw GPIO pin number to use as the source for a `jmp pin` instruction

◆ **sm_config_set_mov_status()**

```
static void sm_config_set_mov_status ( pio_sm_config *               c,
                                       enum pio_mov_status_type      status_sel,
                                       uint                          status_n
                                     )                               inline static
```

Set source for 'mov status' in a state machine configuration.

Parameters

c	Pointer to the configuration structure to modify
status_sel	the status operation selector.

See also enum pio_mov_status_type

Parameters

status_n	parameter for the mov status operation (currently a bit count)

◆ sm_config_set_out_pins()

```
static void sm_config_set_out_pins ( pio_sm_config * c,
                                     uint            out_base,
                                     uint            out_count
                                   )                                  inline  static
```

Set the 'out' pins in a state machine configuration.

Can overlap with the 'in', 'set' and 'sideset' pins

Parameters

c	Pointer to the configuration structure to modify
out_base	0-31 First pin to set as output
out_count	0-32 Number of pins to set.

◆ sm_config_set_out_shift()

```
static void sm_config_set_out_shift ( pio_sm_config * c,
                                      bool            shift_right,
                                      bool            autopull,
                                      uint            pull_threshold
                                    )                                 inline  static
```

Setup 'out' shifting parameters in a state machine configuration.

Parameters

c	Pointer to the configuration structure to modify
shift_right	true to shift OSR to right, false to shift OSR to left
autopull	whether autopull is enabled
pull_threshold	threshold in bits to shift out before auto/conditional re-pulling of the OSR

◆ sm_config_set_out_special()

```
static void sm_config_set_out_special ( pio_sm_config * c,
                                        bool            sticky,
                                        bool            has_enable_pin,
                                        uint            enable_pin_index
                                      )                               inline  static
```

Set special 'out' operations in a state machine configuration.

Parameters

c	Pointer to the configuration structure to modify
sticky	to enable 'sticky' output (i.e. re-asserting most recent OUT/SET pin values on subsequent cycles)
has_enable_pin	true to enable auxiliary OUT enable pin

enable_pin_index pin index for auxiliary OUT enable

◆ **sm_config_set_set_pins()**

```
static void sm_config_set_set_pins ( pio_sm_config *  c,
                                     uint             set_base,
                                     uint             set_count
                                   )                                  inline  static
```

Set the 'set' pins in a state machine configuration.

Can overlap with the 'in', 'out' and 'sideset' pins

Parameters

c	Pointer to the configuration structure to modify
set_base	0-31 First pin to set as
set_count	0-5 Number of pins to set.

◆ **sm_config_set_sideset()**

```
static void sm_config_set_sideset ( pio_sm_config *  c,
                                    uint             bit_count,
                                    bool             optional,
                                    bool             pindirs
                                  )                                   inline  static
```

Set the 'sideset' options in a state machine configuration.

Parameters

c	Pointer to the configuration structure to modify
bit_count	Number of bits to steal from delay field in the instruction for use of side set (max 5)
optional	True if the topmost side set bit is used as a flag for whether to apply side set on that instruction
pindirs	True if the side set affects pin directions rather than values

◆ **sm_config_set_sideset_pins()**

```
static void sm_config_set_sideset_pins ( pio_sm_config *  c,
                                         uint             sideset_base
                                       )                              inline  static
```

Set the 'sideset' pins in a state machine configuration.

Can overlap with the 'in', 'out' and 'set' pins

Parameters

c	Pointer to the configuration structure to modify
sideset_base	0-31 base pin for 'side set'

◆ **sm_config_set_wrap()**

```
static void sm_config_set_wrap ( pio_sm_config *  c,
                                 uint             wrap_target,
                                 uint             wrap
                               )                                      inline  static
```

Set the wrap addresses in a state machine configuration.

Parameters

c	Pointer to the configuration structure to modify
wrap_target	the instruction memory address to wrap to
wrap	the instruction memory address after which to set the program counter to wrap_target if the instruction does not itself update the program_counter

pio_instructions

Part of: Hardware APIs » hardware_pio
PIO instruction encoding. More...

Enumerations

- **enum pio_src_dest {**
  pio_pins = 0u , pio_x = 1u , pio_y = 2u , pio_null = 3u | 0x20u | 0x80u ,
  pio_pindirs = 4u | 0x08u | 0x40u | 0x80u , pio_exec_mov = 4u | 0x08u | 0x10u | 0x20u | 0x40u , pio_status = 5u | 0x08u | 0x10u | 0x20u | 0x80u , pio_pc = 5u | 0x08u | 0x20u | 0x40u ,
  pio_isr = 6u | 0x20u , pio_osr = 7u | 0x10u | 0x20u , pio_exec_out = 7u | 0x08u | 0x20u | 0x40u | 0x80u
  }

  Enumeration of values to pass for source/destination args for instruction encoding functions. More...

Functions

- **static uint pio_encode_delay (uint cycles)**

  Encode just the delay slot bits of an instruction.

- **static uint pio_encode_sideset (uint sideset_bit_count, uint value)**

  Encode just the side set bits of an instruction (in non optional side set mode)

- **static uint pio_encode_sideset_opt (uint sideset_bit_count, uint value)**

  Encode just the side set bits of an instruction (in optional `-opt` side set mode)

- **static uint pio_encode_jmp (uint addr)**

  Encode an unconditional JMP instruction.

- **static uint pio_encode_jmp_not_x (uint addr)**

  Encode a conditional JMP if scratch X zero instruction.

- **static uint pio_encode_jmp_x_dec (uint addr)**

  Encode a conditional JMP if scratch X non-zero (and post-decrement X) instruction.

- **static uint pio_encode_jmp_not_y (uint addr)**

  Encode a conditional JMP if scratch Y zero instruction.

- **static uint pio_encode_jmp_y_dec (uint addr)**

  Encode a conditional JMP if scratch Y non-zero (and post-decrement Y) instruction.

- **static uint pio_encode_jmp_x_ne_y (uint addr)**

  Encode a conditional JMP if scratch X not equal scratch Y instruction.

- **static uint pio_encode_jmp_pin (uint addr)**

  Encode a conditional JMP if input pin high instruction.

- **static uint pio_encode_jmp_not_osre (uint addr)**

  Encode a conditional JMP if output shift register not empty instruction.

- **static uint pio_encode_wait_gpio (bool polarity, uint gpio)**

  Encode a WAIT for GPIO pin instruction.

- **static uint pio_encode_wait_pin (bool polarity, uint pin)**

  Encode a WAIT for pin instruction.

- **static uint pio_encode_wait_irq (bool polarity, bool relative, uint irq)**

  Encode a WAIT for IRQ instruction.

- **static uint pio_encode_in (enum pio_src_dest src, uint count)**

  Encode an IN instruction.

- **static uint pio_encode_out (enum pio_src_dest dest, uint count)**

  Encode an OUT instruction.

- **static uint pio_encode_push (bool if_full, bool block)**

  Encode a PUSH instruction.

- **static uint pio_encode_pull (bool if_empty, bool block)**

  Encode a PULL instruction.

- **static uint pio_encode_mov (enum pio_src_dest dest, enum pio_src_dest src)**

  Encode a MOV instruction.

- **static uint pio_encode_mov_not (enum pio_src_dest dest, enum pio_src_dest src)**

  Encode a MOV instruction with bit invert.

- **static uint pio_encode_mov_reverse (enum pio_src_dest dest, enum pio_src_dest src)**

  Encode a MOV instruction with bit reverse.

- **static uint pio_encode_irq_set (bool relative, uint irq)**

  Encode a IRQ SET instruction.

- **static uint pio_encode_irq_wait (bool relative, uint irq)**

  Encode a IRQ WAIT instruction.

- **static uint pio_encode_irq_clear (bool relative, uint irq)**

  Encode a IRQ CLEAR instruction.

- **static uint pio_encode_set (enum pio_src_dest dest, uint value)**

  Encode a SET instruction.

- **static uint pio_encode_nop (void)**

  Encode a NOP instruction.

Detailed Description

PIO instruction encoding.

Functions for generating PIO instruction encodings programmatically. In debug builds `PARAM_ASSERTIONS_ENABLED_PIO_INSTRUCTIONS` can be set to 1 to enable validation of encoding function parameters.

For fuller descriptions of the instructions in question see the "RP2040 Datasheet"

Enumeration Type Documentation

◆ pio_src_dest

enum pio_src_dest

Enumeration of values to pass for source/destination args for instruction encoding functions.

NOTE

> Not all values are suitable for all functions. Validity is only checked in debug mode when `PARAM_ASSERTIONS_ENABLED_PIO_INSTRUCTIONS` is 1

Function Documentation

◆ pio_encode_delay()

static uint pio_encode_delay (uint ***cycles***) inline static

Encode just the delay slot bits of an instruction.

NOTE

> This function does not return a valid instruction encoding; instead it returns an encoding of the delay slot suitable for `OR`ing with the result of an encoding function for an actual instruction. Care should be taken when combining the results of this function with the results of pio_encode_sideset and pio_encode_sideset_opt as they share the same bits within the instruction encoding.

Parameters

cycles the number of cycles 0-31 (or less if side set is being used)

Returns

the delay slot bits to be ORed with an instruction encoding

◆ pio_encode_in()

static uint pio_encode_in (enum pio_src_dest ***src***,
uint ***count***
) inline static

Encode an IN instruction.

This is the equivalent of `IN <src>, <count>`

Parameters

src The source to take data from
count The number of bits 1-32

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_irq_clear()

```
static uint pio_encode_irq_clear ( bool relative,
                                   uint irq
                                 )
```
inline static

Encode a IRQ CLEAR instruction.

This is the equivalent of `IRQ CLEAR <irq> <relative>`

Parameters

relative	true for a `IRQ CLEAR <irq> REL`, false for regular `IRQ CLEAR <irq>`
irq	the irq number 0-7

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_irq_set()

```
static uint pio_encode_irq_set ( bool relative,
                                 uint irq
                               )
```
inline static

Encode a IRQ SET instruction.

This is the equivalent of `IRQ SET <irq> <relative>`

Parameters

relative	true for a `IRQ SET <irq> REL`, false for regular `IRQ SET <irq>`
irq	the irq number 0-7

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_irq_wait()

```
static uint pio_encode_irq_wait ( bool relative,
                                  uint irq
                                )
```
inline static

Encode a IRQ WAIT instruction.

This is the equivalent of `IRQ WAIT <irq> <relative>`

Parameters

relative	true for a `IRQ WAIT <irq> REL`, false for regular `IRQ WAIT <irq>`
irq	the irq number 0-7

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_jmp()

static uint pio_encode_jmp (uint ***addr***) inline static

Encode an unconditional JMP instruction.

This is the equivalent of `JMP <addr>`

Parameters

addr The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_jmp_not_osre()

static uint pio_encode_jmp_not_osre (uint ***addr***) inline static

Encode a conditional JMP if output shift register not empty instruction.

This is the equivalent of `JMP !OSRE <addr>`

Parameters

addr The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_jmp_not_x()

static uint pio_encode_jmp_not_x (uint ***addr***) inline static

Encode a conditional JMP if scratch X zero instruction.

This is the equivalent of `JMP !X <addr>`

Parameters

addr The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_jmp_not_y()

static uint pio_encode_jmp_not_y (uint ***addr***) inline static

Encode a conditional JMP if scratch Y zero instruction.

This is the equivalent of `JMP !Y <addr>`

Parameters

addr The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_jmp_pin()

static uint pio_encode_jmp_pin (uint ***addr***) inline static

Encode a conditional JMP if input pin high instruction.

This is the equivalent of `JMP PIN <addr>`

Parameters

addr The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_jmp_x_dec()

static uint pio_encode_jmp_x_dec (uint ***addr***) inline static

Encode a conditional JMP if scratch X non-zero (and post-decrement X) instruction.

This is the equivalent of `JMP X-- <addr>`

Parameters

addr The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_jmp_x_ne_y()

static uint pio_encode_jmp_x_ne_y (uint ***addr***) inline static

Encode a conditional JMP if scratch X not equal scratch Y instruction.

This is the equivalent of `JMP X!=Y <addr>`

Parameters

addr The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_jmp_y_dec()

static uint pio_encode_jmp_y_dec (uint ***addr***) inline static

Encode a conditional JMP if scratch Y non-zero (and post-decrement Y) instruction.

This is the equivalent of `JMP Y-- <addr>`

Parameters

addr The target address 0-31 (an absolute address within the PIO instruction memory)

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_mov()

static uint pio_encode_mov (enum pio_src_dest ***dest***,
enum pio_src_dest ***src***
) inline static

Encode a MOV instruction.

This is the equivalent of `MOV <dest>, <src>`

Parameters

dest The destination to write data to
src The source to take data from

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_mov_not()

static uint pio_encode_mov_not (enum pio_src_dest ***dest***,
enum pio_src_dest ***src***
) inline static

Encode a MOV instruction with bit invert.

This is the equivalent of `MOV <dest>, ~<src>`

Parameters

dest	The destination to write inverted data to
src	The source to take data from

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_mov_reverse()

```
static uint pio_encode_mov_reverse ( enum pio_src_dest dest,
                                     enum pio_src_dest src
                                   )                                  inline static
```

Encode a MOV instruction with bit reverse.

This is the equivalent of `MOV <dest>, ::<src>`

Parameters

dest	The destination to write bit reversed data to
src	The source to take data from

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_nop()

```
static uint pio_encode_nop ( void )                                   inline static
```

Encode a NOP instruction.

This is the equivalent of `NOP` which is itself encoded as `MOV y, y`

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_out()

```
static uint pio_encode_out ( enum pio_src_dest dest,
                             uint              count
                           )                                          inline static
```

Encode an OUT instruction.

This is the equivalent of `OUT <src>, <count>`

Parameters

dest	The destination to write data to
count	The number of bits 1-32

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_pull()

```
static uint pio_encode_pull ( bool if_empty,
                              bool block
                            )                                   inline static
```

Encode a PULL instruction.

This is the equivalent of `PULL <if_empty>, <block>`

Parameters

if_empty	true for `PULL IF_EMPTY ...`, false for `PULL ...`
block	true for `PULL ... BLOCK`, false for `PULL ...`

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_push()

```
static uint pio_encode_push ( bool if_full,
                              bool block
                            )                                   inline static
```

Encode a PUSH instruction.

This is the equivalent of `PUSH <if_full>, <block>`

Parameters

if_full	true for `PUSH IF_FULL ...`, false for `PUSH ...`
block	true for `PUSH ... BLOCK`, false for `PUSH ...`

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_set()

```
static uint pio_encode_set ( enum pio_src_dest dest,
                             uint              value
                           )                                    inline static
```

Encode a SET instruction.

This is the equivalent of `SET <dest>, <value>`

Parameters

dest	The destination to apply the value to

value The value 0-31

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_sideset()

static uint pio_encode_sideset (uint ***sideset_bit_count***,
uint ***value***
) inline static

Encode just the side set bits of an instruction (in non optional side set mode)

NOTE

This function does not return a valid instruction encoding; instead it returns an encoding of the side set bits suitable for `OR`ing with the result of an encoding function for an actual instruction. Care should be taken when combining the results of this function with the results of pio_encode_delay as they share the same bits within the instruction encoding.

Parameters

sideset_bit_count	number of side set bits as would be specified via `.sideset` in pioasm
value	the value to sideset on the pins

Returns

the side set bits to be ORed with an instruction encoding

◆ pio_encode_sideset_opt()

static uint pio_encode_sideset_opt (uint ***sideset_bit_count***,
uint ***value***
) inline static

Encode just the side set bits of an instruction (in optional `-opt` side set mode)

NOTE

This function does not return a valid instruction encoding; instead it returns an encoding of the side set bits suitable for `OR`ing with the result of an encoding function for an actual instruction. Care should be taken when combining the results of this function with the results of pio_encode_delay as they share the same bits within the instruction encoding.

Parameters

sideset_bit_count	number of side set bits as would be specified via `.sideset <n> opt` in pioasm
value	the value to sideset on the pins

Returns

the side set bits to be ORed with an instruction encoding

◆ pio_encode_wait_gpio()

static uint pio_encode_wait_gpio (bool ***polarity***,
uint ***gpio***
)

inline static

Encode a WAIT for GPIO pin instruction.

This is the equivalent of `WAIT <polarity> GPIO <gpio>`

Parameters

polarity	true for `WAIT 1`, false for `WAIT 0`
gpio	The real GPIO number 0-31

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_wait_irq()

static uint pio_encode_wait_irq (bool ***polarity***,
bool ***relative***,
uint ***irq***
)

inline static

Encode a WAIT for IRQ instruction.

This is the equivalent of `WAIT <polarity> IRQ <irq> <relative>`

Parameters

polarity	true for `WAIT 1`, false for `WAIT 0`
relative	true for a `WAIT IRQ <irq> REL`, false for regular `WAIT IRQ <irq>`
irq	the irq number 0-7

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

◆ pio_encode_wait_pin()

static uint pio_encode_wait_pin (bool ***polarity***,
uint ***pin***
)

inline static

Encode a WAIT for pin instruction.

This is the equivalent of `WAIT <polarity> PIN <pin>`

Parameters

polarity	true for `WAIT 1`, false for `WAIT 0`
pin	The pin number 0-31 relative to the executing SM's input pin mapping

Returns

The instruction encoding with 0 delay and no side set value

See also pio_encode_delay, pio_encode_sideset, pio_encode_sideset_opt

hardware_pll

Part of: Hardware APIs

Functions

- **`void pll_init (PLL pll, uint ref_div, uint vco_freq, uint post_div1, uint post_div2)`**

  Initialise specified PLL.

- **`void pll_deinit (PLL pll)`**

  Release/uninitialise specified PLL.

Detailed Description

Phase Locked Loop control APIs

There are two PLLs in RP2040. They are:

- pll_sys - Used to generate up to a 133MHz system clock
- pll_usb - Used to generate a 48MHz USB reference clock

For details on how the PLLs are calculated, please refer to the RP2040 datasheet.

Function Documentation

◆ pll_deinit()

void pll_deinit (PLL ***pll***)

Release/uninitialise specified PLL.

This will turn off the power to the specified PLL. Note this function does not currently check if the PLL is in use before powering it off so should be used with care.

Parameters

pll	pll_sys or pll_usb

◆ pll_init()

void pll_init (PLL ***pll***,
uint ***ref_div***,
uint ***vco_freq***,
uint ***post_div1***,
uint ***post_div2***
)

Initialise specified PLL.

Parameters

pll	pll_sys or pll_usb
ref_div	Input clock divider.
vco_freq	Requested output from the VCO (voltage controlled oscillator)

post_div1 Post Divider 1 - range 1-7. Must be >= post_div2
post_div2 Post Divider 2 - range 1-7

hardware_pwm

Part of: Hardware APIs

Enumerations

- **enum pwm_clkdiv_mode { PWM_DIV_FREE_RUNNING = 0 , PWM_DIV_B_HIGH = 1 , PWM_DIV_B_RISING = 2 , PWM_DIV_B_FALLING = 3 }**

  PWM Divider mode settings. More...

Functions

- **static uint pwm_gpio_to_slice_num (uint gpio)**

  Determine the PWM slice that is attached to the specified GPIO.

- **static uint pwm_gpio_to_channel (uint gpio)**

  Determine the PWM channel that is attached to the specified GPIO.

- **static void pwm_config_set_phase_correct (pwm_config *c, bool phase_correct)**

  Set phase correction in a PWM configuration.

- **static void pwm_config_set_clkdiv (pwm_config *c, float div)**

  Set PWM clock divider in a PWM configuration.

- **static void pwm_config_set_clkdiv_int_frac (pwm_config *c, uint8_t integer, uint8_t fract)**

  Set PWM clock divider in a PWM configuration using an 8:4 fractional value.

- **static void pwm_config_set_clkdiv_int (pwm_config *c, uint div)**

  Set PWM clock divider in a PWM configuration.

- **static void pwm_config_set_clkdiv_mode (pwm_config *c, enum pwm_clkdiv_mode mode)**

  Set PWM counting mode in a PWM configuration.

- **static void pwm_config_set_output_polarity (pwm_config *c, bool a, bool b)**

  Set output polarity in a PWM configuration.

- **static void pwm_config_set_wrap (pwm_config *c, uint16_t wrap)**

  Set PWM counter wrap value in a PWM configuration.

- **static void pwm_init (uint slice_num, pwm_config *c, bool start)**

  Initialise a PWM with settings from a configuration object.

- **static pwm_config pwm_get_default_config (void)**

  Get a set of default values for PWM configuration.

- **static void pwm_set_wrap (uint slice_num, uint16_t wrap)**

  Set the current PWM counter wrap value.

- **static void pwm_set_chan_level (uint slice_num, uint chan, uint16_t level)**

  Set the current PWM counter compare value for one channel.

- **static void pwm_set_both_levels (uint slice_num, uint16_t level_a, uint16_t level_b)**

  Set PWM counter compare values.

- **static void pwm_set_gpio_level (uint gpio, uint16_t level)**

  Helper function to set the PWM level for the slice and channel associated with a GPIO.

- **static uint16_t pwm_get_counter (uint slice_num)**

  Get PWM counter.

- **static void pwm_set_counter (uint slice_num, uint16_t c)**

  Set PWM counter.

- **static void pwm_advance_count (uint slice_num)**

  Advance PWM count.

- **static void pwm_retard_count (uint slice_num)**

  Retard PWM count.

- **static void pwm_set_clkdiv_int_frac (uint slice_num, uint8_t integer, uint8_t fract)**

  Set PWM clock divider using an 8:4 fractional value.

- **static void pwm_set_clkdiv (uint slice_num, float divider)**

  Set PWM clock divider.

- **static void pwm_set_output_polarity (uint slice_num, bool a, bool b)**

  Set PWM output polarity.

- **static void pwm_set_clkdiv_mode (uint slice_num, enum pwm_clkdiv_mode mode)**

  Set PWM divider mode.

- **static void pwm_set_phase_correct (uint slice_num, bool phase_correct)**

  Set PWM phase correct on/off.

- **static void pwm_set_enabled (uint slice_num, bool enabled)**

  Enable/Disable PWM.

- **static void pwm_set_mask_enabled (uint32_t mask)**

  Enable/Disable multiple PWM slices simultaneously.

- **static void pwm_set_irq_enabled (uint slice_num, bool enabled)**

  Enable PWM instance interrupt.

- **static void pwm_set_irq_mask_enabled (uint32_t slice_mask, bool enabled)**

  Enable multiple PWM instance interrupts.

- **static void pwm_clear_irq (uint slice_num)**

  Clear a single PWM channel interrupt.

- **static uint32_t pwm_get_irq_status_mask (void)**

Get PWM interrupt status, raw.

- **static void pwm_force_irq (uint slice_num)**

  Force PWM interrupt.

- **static uint pwm_get_dreq (uint slice_num)**

  Return the DREQ to use for pacing transfers to a particular PWM slice.

Detailed Description

Hardware Pulse Width Modulation (PWM) API

The RP2040 PWM block has 8 identical slices. Each slice can drive two PWM output signals, or measure the frequency or duty cycle of an input signal. This gives a total of up to 16 controllable PWM outputs. All 30 GPIOs can be driven by the PWM block.

The PWM hardware functions by continuously comparing the input value to a free-running counter. This produces a toggling output where the amount of time spent at the high output level is proportional to the input value. The fraction of time spent at the high signal level is known as the duty cycle of the signal.

The default behaviour of a PWM slice is to count upward until the wrap value (pwm_config_set_wrap) is reached, and then immediately wrap to 0. PWM slices also offer a phase-correct mode, where the counter starts to count downward after reaching TOP, until it reaches 0 again.

Example

```
// Output PWM signals on pins 0 and 1

#include "pico/stdlib.h"
#include "hardware/pwm.h"


int main() {

    // Tell GPIO 0 and 1 they are allocated to the PWM
    gpio_set_function(0, GPIO_FUNC_PWM);
    gpio_set_function(1, GPIO_FUNC_PWM);

    // Find out which PWM slice is connected to GPIO 0 (it's slice 0)
    uint slice_num = pwm_gpio_to_slice_num(0);

    // Set period of 4 cycles (0 to 3 inclusive)
    pwm_set_wrap(slice_num, 3);
    // Set channel A output high for one cycle before dropping
    pwm_set_chan_level(slice_num, PWM_CHAN_A, 1);
    // Set initial B output high for three cycles before dropping
    pwm_set_chan_level(slice_num, PWM_CHAN_B, 3);
    // Set the PWM running
    pwm_set_enabled(slice_num, true);

    // Note we could also use pwm_set_gpio_level(gpio, x) which looks up the
    // correct slice and channel for a given GPIO.
}
```

Enumeration Type Documentation

◆ pwm_clkdiv_mode

enum pwm_clkdiv_mode

PWM Divider mode settings.

Enumerator	
PWM_DIV_FREE_RUNNING	Free-running counting at rate dictated by fractional divider.
PWM_DIV_B_HIGH	Fractional divider is gated by the PWM B pin.
PWM_DIV_B_RISING	Fractional divider advances with each rising edge of the PWM B pin.

PWM_DIV_B_FALLING	Fractional divider advances with each falling edge of the PWM B pin.

Function Documentation

◆ pwm_advance_count()

```
static void pwm_advance_count ( uint slice_num )
```

inline static

Advance PWM count.

Advance the phase of a running the counter by 1 count.

This function will return once the increment is complete.

Parameters

slice_num PWM slice number

◆ pwm_clear_irq()

```
static void pwm_clear_irq ( uint slice_num )
```

inline static

Clear a single PWM channel interrupt.

Parameters

slice_num PWM slice number

◆ pwm_config_set_clkdiv()

```
static void pwm_config_set_clkdiv ( pwm_config * c,
                                    float        div
                                  )
```

inline static

Set PWM clock divider in a PWM configuration.

Parameters

c PWM configuration struct to modify
div Value to divide counting rate by. Must be greater than or equal to 1.

If the divide mode is free-running, the PWM counter runs at clk_sys / div. Otherwise, the divider reduces the rate of events seen on the B pin input (level or edge) before passing them on to the PWM counter.

◆ pwm_config_set_clkdiv_int()

```
static void pwm_config_set_clkdiv_int ( pwm_config * c,
                                        uint         div
                                      )
```

inline static

Set PWM clock divider in a PWM configuration.

Parameters

c PWM configuration struct to modify
div Integer value to reduce counting rate by. Must be greater than or equal to 1.

If the divide mode is free-running, the PWM counter runs at clk_sys / div. Otherwise, the divider reduces the rate of events seen on the B pin input (level or edge) before passing them on to the PWM counter.

◆ pwm_config_set_clkdiv_int_frac()

```
static void pwm_config_set_clkdiv_int_frac ( pwm_config *  c,
                                             uint8_t       integer,
                                             uint8_t       fract
                                           )                                    inline  static
```

Set PWM clock divider in a PWM configuration using an 8:4 fractional value.

Parameters

c PWM configuration struct to modify
integer 8 bit integer part of the clock divider. Must be greater than or equal to 1.
fract 4 bit fractional part of the clock divider

If the divide mode is free-running, the PWM counter runs at clk_sys / div. Otherwise, the divider reduces the rate of events seen on the B pin input (level or edge) before passing them on to the PWM counter.

◆ pwm_config_set_clkdiv_mode()

```
static void pwm_config_set_clkdiv_mode ( pwm_config *                c,
                                         enum pwm_clkdiv_mode        mode
                                       )                                    inline  static
```

Set PWM counting mode in a PWM configuration.

Parameters

c PWM configuration struct to modify
mode PWM divide/count mode

Configure which event gates the operation of the fractional divider. The default is always-on (free-running PWM). Can also be configured to count on high level, rising edge or falling edge of the B pin input.

◆ pwm_config_set_output_polarity()

```
static void pwm_config_set_output_polarity ( pwm_config *  c,
                                             bool          a,
                                             bool          b
                                           )                                    inline  static
```

Set output polarity in a PWM configuration.

Parameters

c PWM configuration struct to modify
a true to invert output A
b true to invert output B

◆ pwm_config_set_phase_correct()

static void pwm_config_set_phase_correct (pwm_config * ***c***,
bool ***phase_correct***
)

inline static

Set phase correction in a PWM configuration.

Parameters

c	PWM configuration struct to modify
phase_correct	true to set phase correct modulation, false to set trailing edge

Setting phase control to true means that instead of wrapping back to zero when the wrap point is reached, the PWM starts counting back down. The output frequency is halved when phase-correct mode is enabled.

◆ pwm_config_set_wrap()

static void pwm_config_set_wrap (pwm_config * ***c***,
uint16_t ***wrap***
)

inline static

Set PWM counter wrap value in a PWM configuration.

Set the highest value the counter will reach before returning to 0. Also known as TOP.

Parameters

c	PWM configuration struct to modify
wrap	Value to set wrap to

◆ pwm_force_irq()

static void pwm_force_irq (uint ***slice_num***)

inline static

Force PWM interrupt.

Parameters

slice_num	PWM slice number

◆ pwm_get_counter()

static uint16_t pwm_get_counter (uint ***slice_num***)

inline static

Get PWM counter.

Get current value of PWM counter

Parameters

slice_num	PWM slice number

Returns

Current value of the PWM counter

◆ pwm_get_default_config()

static pwm_config pwm_get_default_config (void) inline static

Get a set of default values for PWM configuration.

PWM config is free-running at system clock speed, no phase correction, wrapping at 0xffff, with standard polarities for channels A and B.

Returns

Set of default values.

◆ pwm_get_dreq()

static uint pwm_get_dreq (uint ***slice_num***) inline static

Return the DREQ to use for pacing transfers to a particular PWM slice.

Parameters

slice_num PWM slice number

◆ pwm_get_irq_status_mask()

static uint32_t pwm_get_irq_status_mask (void) inline static

Get PWM interrupt status, raw.

Returns

Bitmask of all PWM interrupts currently set

◆ pwm_gpio_to_channel()

static uint pwm_gpio_to_channel (uint ***gpio***) inline static

Determine the PWM channel that is attached to the specified GPIO.

Each slice 0 to 7 has two channels, A and B.

Returns

The PWM channel that controls the specified GPIO.

◆ pwm_gpio_to_slice_num()

static uint pwm_gpio_to_slice_num (uint ***gpio***) inline static

Determine the PWM slice that is attached to the specified GPIO.

Returns

The PWM slice number that controls the specified GPIO.

◆ pwm_init()

```
static void pwm_init ( uint              slice_num,
                       pwm_config *      c,
                       bool              start
                     )                                    inline  static
```

Initialise a PWM with settings from a configuration object.

Use the pwm_get_default_config() function to initialise a config structure, make changes as needed using the pwm_config_* functions, then call this function to set up the PWM.

Parameters

slice_num	PWM slice number
c	The configuration to use
start	If true the PWM will be started running once configured. If false you will need to start manually using pwm_set_enabled() or pwm_set_mask_enabled()

◆ pwm_retard_count()

```
static void pwm_retard_count ( uint slice_num )           inline  static
```

Retard PWM count.

Retard the phase of a running counter by 1 count

This function will return once the retardation is complete.

Parameters

slice_num	PWM slice number

◆ pwm_set_both_levels()

```
static void pwm_set_both_levels ( uint      slice_num,
                                  uint16_t  level_a,
                                  uint16_t  level_b
                                )                          inline  static
```

Set PWM counter compare values.

Set the value of the PWM counter compare values, A and B.

The counter compare register is double-buffered in hardware. This means that, when the PWM is running, a write to the counter compare values does not take effect until the next time the PWM slice wraps (or, in phase-correct mode, the next time the slice reaches 0). If the PWM is not running, the write is latched in immediately.

Parameters

slice_num	PWM slice number
level_a	Value to set compare A to. When the counter reaches this value the A output is deasserted
level_b	Value to set compare B to. When the counter reaches this value the B output is deasserted

◆ **pwm_set_chan_level()**

```
static void pwm_set_chan_level ( uint     slice_num,
                                 uint     chan,
                                 uint16_t level
                               )                                    inline  static
```

Set the current PWM counter compare value for one channel.

Set the value of the PWM counter compare value, for either channel A or channel B.

The counter compare register is double-buffered in hardware. This means that, when the PWM is running, a write to the counter compare values does not take effect until the next time the PWM slice wraps (or, in phase-correct mode, the next time the slice reaches 0). If the PWM is not running, the write is latched in immediately.

Parameters

slice_num	PWM slice number
chan	Which channel to update. 0 for A, 1 for B.
level	new level for the selected output

◆ **pwm_set_clkdiv()**

```
static void pwm_set_clkdiv ( uint  slice_num,
                             float divider
                           )                                        inline  static
```

Set PWM clock divider.

Set the clock divider. Counter increment will be on sysclock divided by this value, taking into account the gating.

Parameters

slice_num	PWM slice number
divider	Floating point clock divider, 1.f <= value < 256.f

◆ **pwm_set_clkdiv_int_frac()**

```
static void pwm_set_clkdiv_int_frac ( uint    slice_num,
                                      uint8_t integer,
                                      uint8_t fract
                                    )                               inline  static
```

Set PWM clock divider using an 8:4 fractional value.

Set the clock divider. Counter increment will be on sysclock divided by this value, taking into account the gating.

Parameters

slice_num	PWM slice number
integer	8 bit integer part of the clock divider
fract	4 bit fractional part of the clock divider

◆ **pwm_set_clkdiv_mode()**

```
static void pwm_set_clkdiv_mode ( uint slice_num,
```

```
                                    enum pwm_clkdiv_mode  mode
                                )                                                    inline  static
```

Set PWM divider mode.

Parameters

slice_num	PWM slice number
mode	Required divider mode

◆ pwm_set_counter()

```
static void pwm_set_counter ( uint      slice_num,
                              uint16_t  c
                            )                                                    inline  static
```

Set PWM counter.

Set the value of the PWM counter

Parameters

slice_num	PWM slice number
c	Value to set the PWM counter to

◆ pwm_set_enabled()

```
static void pwm_set_enabled ( uint  slice_num,
                              bool  enabled
                            )                                                    inline  static
```

Enable/Disable PWM.

When a PWM is disabled, it halts its counter, and the output pins are left high or low depending on exactly when the counter is halted. When re-enabled the PWM resumes immediately from where it left off.

If the PWM's output pins need to be low when halted:

- The counter compare can be set to zero whilst the PWM is enabled, and then the PWM disabled once both pins are seen to be low
- The GPIO output overrides can be used to force the actual pins low
- The PWM can be run for one cycle (i.e. enabled then immediately disabled) with a TOP of 0, count of 0 and counter compare of 0, to force the pins low when the PWM has already been halted. The same method can be used with a counter compare value of 1 to force a pin high.

Note that, when disabled, the PWM can still be advanced one count at a time by pulsing the PH_ADV bit in its CSR. The output pins transition as though the PWM were enabled.

Parameters

slice_num	PWM slice number
enabled	true to enable the specified PWM, false to disable.

◆ pwm_set_gpio_level()

```
static void pwm_set_gpio_level ( uint      gpio,
                                 uint16_t  level
```

)

inline static

Helper function to set the PWM level for the slice and channel associated with a GPIO.

Look up the correct slice (0 to 7) and channel (A or B) for a given GPIO, and update the corresponding counter compare field.

This PWM slice should already have been configured and set running. Also be careful of multiple GPIOs mapping to the same slice and channel (if GPIOs have a difference of 16).

The counter compare register is double-buffered in hardware. This means that, when the PWM is running, a write to the counter compare values does not take effect until the next time the PWM slice wraps (or, in phase-correct mode, the next time the slice reaches 0). If the PWM is not running, the write is latched in immediately.

Parameters

gpio	GPIO to set level of
level	PWM level for this GPIO

◆ pwm_set_irq_enabled()

static void pwm_set_irq_enabled (uint *slice_num*,
bool *enabled*
)

inline static

Enable PWM instance interrupt.

Used to enable a single PWM instance interrupt.

Parameters

slice_num	PWM block to enable/disable
enabled	true to enable, false to disable

◆ pwm_set_irq_mask_enabled()

static void pwm_set_irq_mask_enabled (uint32_t *slice_mask*,
bool *enabled*
)

inline static

Enable multiple PWM instance interrupts.

Use this to enable multiple PWM interrupts at once.

Parameters

slice_mask	Bitmask of all the blocks to enable/disable. Channel 0 = bit 0, channel 1 = bit 1 etc.
enabled	true to enable, false to disable

◆ pwm_set_mask_enabled()

static void pwm_set_mask_enabled (uint32_t *mask*)

inline static

Enable/Disable multiple PWM slices simultaneously.

Parameters

mask Bitmap of PWMs to enable/disable. Bits 0 to 7 enable slices 0-7 respectively

◆ pwm_set_output_polarity()

```
static void pwm_set_output_polarity ( uint slice_num,
                                      bool a,
                                      bool b
                                    )                                    inline static
```

Set PWM output polarity.

Parameters

slice_num	PWM slice number
a	true to invert output A
b	true to invert output B

◆ pwm_set_phase_correct()

```
static void pwm_set_phase_correct ( uint slice_num,
                                    bool phase_correct
                                  )                                      inline static
```

Set PWM phase correct on/off.

Parameters

slice_num	PWM slice number
phase_correct	true to set phase correct modulation, false to set trailing edge

Setting phase control to true means that instead of wrapping back to zero when the wrap point is reached, the PWM starts counting back down. The output frequency is halved when phase-correct mode is enabled.

◆ pwm_set_wrap()

```
static void pwm_set_wrap ( uint slice_num,
                           uint16_t wrap
                         )                                               inline static
```

Set the current PWM counter wrap value.

Set the highest value the counter will reach before returning to 0. Also known as TOP.

The counter wrap value is double-buffered in hardware. This means that, when the PWM is running, a write to the counter wrap value does not take effect until after the next time the PWM slice wraps (or, in phase-correct mode, the next time the slice reaches 0). If the PWM is not running, the write is latched in immediately.

Parameters

slice_num	PWM slice number
wrap	Value to set wrap to

hardware_resets

Part of: Hardware APIs

Functions

- **`static void reset_block (uint32_t bits)`**

  Reset the specified HW blocks.

- **`static void unreset_block (uint32_t bits)`**

  bring specified HW blocks out of reset

- **`static void unreset_block_wait (uint32_t bits)`**

  Bring specified HW blocks out of reset and wait for completion.

Detailed Description

Hardware Reset API

The reset controller allows software control of the resets to all of the peripherals that are not critical to boot the processor in the RP2040.

reset_bitmask

Multiple blocks are referred to using a bitmask as follows:

Block to reset	Bit
USB	24
UART 1	23
UART 0	22
Timer	21
TB Manager	20
SysInfo	19
System Config	18
SPI 1	17
SPI 0	16
RTC	15
PWM	14
PLL USB	13
PLL System	12
PIO 1	11
PIO 0	10
Pads - QSPI	9
Pads - bank 0	8
JTAG	7
IO Bank 1	6
IO Bank 0	5
I2C 1	4
I2C 0	3
DMA	2
Bus Control	1
ADC 0	0

Example

```c
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/resets.h"


int main() {
    stdio_init_all();

    printf("Hello, reset!\n");

    // Put the PWM block into reset
    reset_block(RESETS_RESET_PWM_BITS);

    // And bring it out
    unreset_block_wait(RESETS_RESET_PWM_BITS);

    // Put the PWM and RTC block into reset
    reset_block(RESETS_RESET_PWM_BITS | RESETS_RESET_RTC_BITS);

    // Wait for both to come out of reset
    unreset_block_wait(RESETS_RESET_PWM_BITS | RESETS_RESET_RTC_BITS);

    return 0;
}
```

Function Documentation

◆ reset_block()

static void reset_block (uint32_t ***bits***) inline static

Reset the specified HW blocks.

Parameters

bits Bit pattern indicating blocks to reset. See reset_bitmask

◆ unreset_block()

static void unreset_block (uint32_t ***bits***) inline static

bring specified HW blocks out of reset

Parameters

bits Bit pattern indicating blocks to unreset. See reset_bitmask

◆ unreset_block_wait()

static void unreset_block_wait (uint32_t ***bits***) inline static

Bring specified HW blocks out of reset and wait for completion.

Parameters

bits Bit pattern indicating blocks to unreset. See reset_bitmask

hardware_rtc

Part of: Hardware APIs

Typedefs

- **typedef void(* rtc_callback_t) (void)**

Functions

- **void rtc_init (void)**

  Initialise the RTC system.

- **bool rtc_set_datetime (datetime_t *t)**

  Set the RTC to the specified time.

- **bool rtc_get_datetime (datetime_t *t)**

  Get the current time from the RTC.

- **bool rtc_running (void)**

  Is the RTC running?

- **void rtc_set_alarm (datetime_t *t, rtc_callback_t user_callback)**

  Set a time in the future for the RTC to call a user provided callback.

- **void rtc_enable_alarm (void)**

  Enable the RTC alarm (if inactive)

- **void rtc_disable_alarm (void)**

  Disable the RTC alarm (if active)

Detailed Description

Hardware Real Time Clock API

The RTC keeps track of time in human readable format and generates events when the time is equal to a preset value. Think of a digital clock, not epoch time used by most computers. There are seven fields, one each for year (12 bit), month (4 bit), day (5 bit), day of the week (3 bit), hour (5 bit) minute (6 bit) and second (6 bit), storing the data in binary format.

See also datetime_t

Example

```
#include <stdio.h>
#include "hardware/rtc.h"
#include "pico/stdlib.h"
#include "pico/util/datetime.h"


int main() {
    stdio_init_all();
    printf("Hello RTC!\n");

    char datetime_buf[256];
    char *datetime_str = &datetime_buf[0];

    // Start on Friday 5th of June 2020 15:45:00
    datetime_t t = {
            .year  = 2020,
            .month = 06,
            .day   = 05,
            .dotw  = 5, // 0 is Sunday, so 5 is Friday
            .hour  = 15,
            .min   = 45,
            .sec   = 00
    };

    // Start the RTC
```

```
    rtc_init();
    rtc_set_datetime(&t);

    // clk_sys is >2000x faster than clk_rtc, so datetime is not updated immediately when rtc_g
et_datetime() is called.
    // tbe delay is up to 3 RTC clock cycles (which is 64us with the default clock settings)
    sleep_us(64);

    // Print the time
    while (true) {
        rtc_get_datetime(&t);
        datetime_to_str(datetime_str, sizeof(datetime_buf), &t);
        printf("\r%s      ", datetime_str);
        sleep_ms(100);
    }
}
```

Typedef Documentation

◆ rtc_callback_t

typedef void(* rtc_callback_t) (void)

Callback function type for RTC alarms

See also rtc_set_alarm()

Function Documentation

◆ rtc_get_datetime()

bool rtc_get_datetime (datetime_t * ***t***)

Get the current time from the RTC.

Parameters

t Pointer to a datetime_t structure to receive the current RTC time

Returns

true if datetime is valid, false if the RTC is not running.

◆ rtc_running()

bool rtc_running (void)

Is the RTC running?

◆ rtc_set_alarm()

void rtc_set_alarm (datetime_t * ***t,***
rtc_callback_t ***user_callback***
)

Set a time in the future for the RTC to call a user provided callback.

Parameters

t	Pointer to a datetime_t structure containing a time in the future to fire the alarm. Any values set to -1 will not be matched on.
user_callback	pointer to a rtc_callback_t to call when the alarm fires

◆ **rtc_set_datetime()**

bool rtc_set_datetime (datetime_t * ***t***)

Set the RTC to the specified time.

NOTE

Note that after setting the RTC date and time, a subsequent read of the values (e.g. via rtc_get_datetime()) may not reflect the new setting until up to three cycles of the potentially-much-slower RTC clock domain have passed. This represents a period of 64 microseconds with the default RTC clock configuration.

Parameters

t Pointer to a datetime_t structure contains time to set

Returns

true if set, false if the passed in datetime was invalid.

hardware_spi

Part of: Hardware APIs

Macros

- **#define spi0 ((spi_inst_t *)spi0_hw)**
- **#define spi1 ((spi_inst_t *)spi1_hw)**

Enumerations

- **enum spi_cpha_t { SPI_CPHA_0 = 0 , SPI_CPHA_1 = 1 }**

  Enumeration of SPI CPHA (clock phase) values.

- **enum spi_cpol_t { SPI_CPOL_0 = 0 , SPI_CPOL_1 = 1 }**

  Enumeration of SPI CPOL (clock polarity) values.

- **enum spi_order_t { SPI_LSB_FIRST = 0 , SPI_MSB_FIRST = 1 }**

  Enumeration of SPI bit-order values.

Functions

- **uint spi_init (spi_inst_t *spi, uint baudrate)**

  Initialise SPI instancesPuts the SPI into a known state, and enable it. Must be called before other functions.

- **void spi_deinit (spi_inst_t *spi)**

  Deinitialise SPI instancesPuts the SPI into a disabled state. Init will need to be called to reenable the device functions.

- **uint spi_set_baudrate (spi_inst_t *spi, uint baudrate)**

  Set SPI baudrate.

- **uint spi_get_baudrate (const spi_inst_t *spi)**

  Get SPI baudrate.

- **static uint spi_get_index (const spi_inst_t *spi)**

  Convert SPI instance to hardware instance number.

- **static void spi_set_format (spi_inst_t *spi, uint data_bits, spi_cpol_t cpol, spi_cpha_t cpha, __unused spi_order_t order)**

  Configure SPI.

- **static void spi_set_slave (spi_inst_t *spi, bool slave)**

  Set SPI master/slave.

- **static bool spi_is_writable (const spi_inst_t *spi)**

  Check whether a write can be done on SPI device.

- **static bool spi_is_readable (const spi_inst_t *spi)**

  Check whether a read can be done on SPI device.

- **static bool spi_is_busy (const spi_inst_t *spi)**

  Check whether SPI is busy.

- **int spi_write_read_blocking (spi_inst_t *spi, const uint8_t *src, uint8_t *dst, size_t len)**

  Write/Read to/from an SPI device.

- **int spi_write_blocking (spi_inst_t *spi, const uint8_t *src, size_t len)**

  Write to an SPI device, blocking.

- **int spi_read_blocking (spi_inst_t *spi, uint8_t repeated_tx_data, uint8_t *dst, size_t len)**

  Read from an SPI device.

- **int spi_write16_read16_blocking (spi_inst_t *spi, const uint16_t *src, uint16_t *dst, size_t len)**

  Write/Read half words to/from an SPI device.

- **int spi_write16_blocking (spi_inst_t *spi, const uint16_t *src, size_t len)**

  Write to an SPI device.

- **int spi_read16_blocking (spi_inst_t *spi, uint16_t repeated_tx_data, uint16_t *dst, size_t len)**

  Read from an SPI device.

- **static uint spi_get_dreq (spi_inst_t *spi, bool is_tx)**

  Return the DREQ to use for pacing transfers to/from a particular SPI instance.

Detailed Description

Hardware SPI API

RP2040 has 2 identical instances of the Serial Peripheral Interface (SPI) controller.

The PrimeCell SSP is a master or slave interface for synchronous serial communication with peripheral devices that have Motorola SPI, National Semiconductor Microwire, or Texas Instruments synchronous serial interfaces.

Controller can be defined as master or slave using the spi_set_slave function.

Each controller can be connected to a number of GPIO pins, see the datasheet GPIO function selection table for more information.

Macro Definition Documentation

◆ spi0

```
#define spi0   ((spi_inst_t *)spi0_hw)
```

Identifier for the first (SPI 0) hardware SPI instance (for use in SPI functions).

e.g. spi_init(spi0, 48000)

◆ spi1

```
#define spi1   ((spi_inst_t *)spi1_hw)
```

Identifier for the second (SPI 1) hardware SPI instance (for use in SPI functions).

e.g. spi_init(spi1, 48000)

Function Documentation

◆ spi_deinit()

```
void spi_deinit ( spi_inst_t *  spi )
```

Deinitialise SPI instancesPuts the SPI into a disabled state. Init will need to be called to reenable the device functions.

Parameters

spi SPI instance specifier, either spi0 or spi1

◆ spi_get_baudrate()

```
uint spi_get_baudrate ( const spi_inst_t *  spi )
```

Get SPI baudrate.

Get SPI baudrate which was set by

See also spi_set_baudrate

Parameters

spi SPI instance specifier, either spi0 or spi1

Returns

The actual baudrate set

◆ spi_get_dreq()

```
static uint spi_get_dreq ( spi_inst_t *  spi,
                           bool          is_tx
                         )                                    inline  static
```

Return the DREQ to use for pacing transfers to/from a particular SPI instance.

Parameters

spi	SPI instance specifier, either spi0 or spi1
is_tx	true for sending data to the SPI instance, false for receiving data from the SPI instance

◆ spi_get_index()

```
static uint spi_get_index ( const spi_inst_t *  spi )        inline  static
```

Convert SPI instance to hardware instance number.

Parameters

spi	SPI instance

Returns

Number of SPI, 0 or 1.

◆ spi_init()

```
uint spi_init ( spi_inst_t *  spi,
                uint          baudrate
              )
```

Initialise SPI instancesPuts the SPI into a known state, and enable it. Must be called before other functions.

NOTE

There is no guarantee that the baudrate requested can be achieved exactly; the nearest will be chosen and returned

Parameters

spi	SPI instance specifier, either spi0 or spi1
baudrate	Baudrate requested in Hz

Returns

the actual baud rate set

◆ spi_is_busy()

```
static bool spi_is_busy ( const spi_inst_t *  spi )          inline  static
```

Check whether SPI is busy.

Parameters

spi SPI instance specifier, either spi0 or spi1

Returns

true if SPI is busy

◆ spi_is_readable()

static bool spi_is_readable (const spi_inst_t * ***spi***) inline static

Check whether a read can be done on SPI device.

Parameters

spi SPI instance specifier, either spi0 or spi1

Returns

true if a read is possible i.e. data is present

◆ spi_is_writable()

static bool spi_is_writable (const spi_inst_t * ***spi***) inline static

Check whether a write can be done on SPI device.

Parameters

spi SPI instance specifier, either spi0 or spi1

Returns

false if no space is available to write. True if a write is possible

◆ spi_read16_blocking()

```
int spi_read16_blocking ( spi_inst_t *  spi,
                          uint16_t      repeated_tx_data,
                          uint16_t *    dst,
                          size_t        len
                        )
```

Read from an SPI device.

Read `len` halfwords from SPI to `dst`. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate. `repeated_tx_data` is output repeatedly on TX as data is read in from RX. Generally this can be 0, but some devices require a specific value here, e.g. SD cards expect 0xff

NOTE

SPI should be initialised with 16 data_bits using spi_set_format first, otherwise this function will only read 8 data_bits.

Parameters

spi	SPI instance specifier, either spi0 or spi1
repeated_tx_data	Buffer of data to write
dst	Buffer for read data
len	Length of buffer `dst` in halfwords

Returns

Number of halfwords written/read

◆ spi_read_blocking()

```
int spi_read_blocking ( spi_inst_t *  spi,
                        uint8_t       repeated_tx_data,
                        uint8_t *     dst,
                        size_t        len
                      )
```

Read from an SPI device.

Read `len` bytes from SPI to `dst`. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate. `repeated_tx_data` is output repeatedly on TX as data is read in from RX. Generally this can be 0, but some devices require a specific value here, e.g. SD cards expect 0xff

Parameters

spi	SPI instance specifier, either spi0 or spi1
repeated_tx_data	Buffer of data to write
dst	Buffer for read data
len	Length of buffer `dst`

Returns

Number of bytes written/read

◆ spi_set_baudrate()

```
uint spi_set_baudrate ( spi_inst_t *  spi,
                        uint          baudrate
                      )
```

Set SPI baudrate.

Set SPI frequency as close as possible to baudrate, and return the actual achieved rate.

Parameters

spi	SPI instance specifier, either spi0 or spi1
baudrate	Baudrate required in Hz, should be capable of a bitrate of at least 2Mbps, or higher, depending on system clock settings.

Returns

The actual baudrate set

◆ spi_set_format()

```
static void spi_set_format ( spi_inst_t *  spi,
```

```
                                uint                data_bits,
                                spi_cpol_t          cpol,
                                spi_cpha_t          cpha,
                                __unused spi_order_t order
                              )                                                  inline  static
```

Configure SPI.

Configure how the SPI serialises and deserialises data on the wire

Parameters

spi	SPI instance specifier, either spi0 or spi1
data_bits	Number of data bits per transfer. Valid values 4..16.
cpol	SSPCLKOUT polarity, applicable to Motorola SPI frame format only.
cpha	SSPCLKOUT phase, applicable to Motorola SPI frame format only
order	Must be SPI_MSB_FIRST, no other values supported on the PL022

◆ spi_set_slave()

```
static void spi_set_slave ( spi_inst_t * spi,
                            bool         slave
                          )                                                  inline  static
```

Set SPI master/slave.

Configure the SPI for master- or slave-mode operation. By default, spi_init() sets master-mode.

Parameters

spi	SPI instance specifier, either spi0 or spi1
slave	true to set SPI device as a slave device, false for master.

◆ spi_write16_blocking()

```
int spi_write16_blocking ( spi_inst_t *       spi,
                           const uint16_t * src,
                           size_t           len
                         )
```

Write to an SPI device.

Write `len` halfwords from `src` to SPI. Discard any data received back. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

NOTE

SPI should be initialised with 16 data_bits using spi_set_format first, otherwise this function will only write 8 data_bits.

Parameters

spi	SPI instance specifier, either spi0 or spi1
src	Buffer of data to write
len	Length of buffers

Returns

Number of halfwords written/read

◆ spi_write16_read16_blocking()

```
int spi_write16_read16_blocking ( spi_inst_t *       spi,
                                  const uint16_t *   src,
                                  uint16_t *         dst,
                                  size_t             len
                                )
```

Write/Read half words to/from an SPI device.

Write `len` halfwords from `src` to SPI. Simultaneously read `len` halfwords from SPI to `dst`. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

NOTE

SPI should be initialised with 16 data_bits using spi_set_format first, otherwise this function will only read/write 8 data_bits.

Parameters

spi SPI instance specifier, either spi0 or spi1
src Buffer of data to write
dst Buffer for read data
len Length of BOTH buffers in halfwords

Returns

Number of halfwords written/read

◆ spi_write_blocking()

```
int spi_write_blocking ( spi_inst_t *       spi,
                         const uint8_t *    src,
                         size_t             len
                       )
```

Write to an SPI device, blocking.

Write `len` bytes from `src` to SPI, and discard any data received back Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

Parameters

spi SPI instance specifier, either spi0 or spi1
src Buffer of data to write
len Length of `src`

Returns

Number of bytes written/read

◆ spi_write_read_blocking()

```
int spi_write_read_blocking ( spi_inst_t *       spi,
                              const uint8_t *    src,
                              uint8_t *          dst,
                              size_t             len
                            )
```

Write/Read to/from an SPI device.

Write `len` bytes from `src` to SPI. Simultaneously read `len` bytes from SPI to `dst`. Blocks until all data is transferred. No timeout, as SPI hardware always transfers at a known data rate.

Parameters

spi SPI instance specifier, either spi0 or spi1
src Buffer of data to write
dst Buffer for read data
len Length of BOTH buffers

Returns

Number of bytes written/read

hardware_sync

Part of: Hardware APIs

Typedefs

- **`typedef volatile uint32_t spin_lock_t`**

  A spin lock identifier.

Functions

- **`static __force_inline void __sev (void)`**

  Insert a SEV instruction in to the code path.

- **`static __force_inline void __wfe (void)`**

  Insert a WFE instruction in to the code path.

- **`static __force_inline void __wfi (void)`**

  Insert a WFI instruction in to the code path.

- **`static __force_inline void __dmb (void)`**

  Insert a DMB instruction in to the code path.

- **`static __force_inline void __dsb (void)`**

  Insert a DSB instruction in to the code path.

- **`static __force_inline void __isb (void)`**

  Insert a ISB instruction in to the code path.

- **`static __force_inline void __mem_fence_acquire (void)`**

  Acquire a memory fence.

- **`static __force_inline void __mem_fence_release (void)`**

  Release a memory fence.

- **`static __force_inline uint32_t save_and_disable_interrupts (void)`**

  Save and disable interrupts.

- **static __force_inline void restore_interrupts (uint32_t status)**

  Restore interrupts to a specified state.

- **static __force_inline spin_lock_t * spin_lock_instance (uint lock_num)**

  Get HW Spinlock instance from number.

- **static __force_inline uint spin_lock_get_num (spin_lock_t *lock)**

  Get HW Spinlock number from instance.

- **static __force_inline void spin_lock_unsafe_blocking (spin_lock_t *lock)**

  Acquire a spin lock without disabling interrupts (hence unsafe)

- **static __force_inline void spin_unlock_unsafe (spin_lock_t *lock)**

  Release a spin lock without re-enabling interrupts.

- **static __force_inline uint32_t spin_lock_blocking (spin_lock_t *lock)**

  Acquire a spin lock safely.

- **static bool is_spin_locked (spin_lock_t *lock)**

  Check to see if a spinlock is currently acquired elsewhere.

- **static __force_inline void spin_unlock (spin_lock_t *lock, uint32_t saved_irq)**

  Release a spin lock safely.

- **spin_lock_t * spin_lock_init (uint lock_num)**

  Initialise a spin lock.

- **void spin_locks_reset (void)**

  Release all spin locks.

- **uint next_striped_spin_lock_num (void)**

  Return a spin lock number from the *striped* range.

- **void spin_lock_claim (uint lock_num)**

  Mark a spin lock as used.

- **void spin_lock_claim_mask (uint32_t lock_num_mask)**

  Mark multiple spin locks as used.

- **void spin_lock_unclaim (uint lock_num)**

  Mark a spin lock as no longer used.

- **int spin_lock_claim_unused (bool required)**

  Claim a free spin lock.

- **bool spin_lock_is_claimed (uint lock_num)**

  Determine if a spin lock is claimed.

Detailed Description

Low level hardware spin locks, barrier and processor event APIs

Spin Locks

The RP2040 provides 32 hardware spin locks, which can be used to manage mutually-exclusive access to shared software and hardware resources.

Generally each spin lock itself is a shared resource, i.e. the same hardware spin lock can be used by multiple higher level primitives (as long as the spin locks are neither held for long periods, nor held concurrently with other spin locks by the same core - which could lead to deadlock). A hardware spin lock that is exclusively owned can be used individually without more flexibility and without regard to other software. Note that no hardware spin lock may be acquired re-entrantly (i.e. hardware spin locks are not on their own safe for use by both thread code and IRQs) however the default spinlock related methods here (e.g. spin_lock_blocking) always disable interrupts while the lock is held as use by IRQ handlers and user code is common/desirable, and spin locks are only expected to be held for brief periods.

The SDK uses the following default spin lock assignments, classifying which spin locks are reserved for exclusive/special purposes vs those suitable for more general shared use:

Number (ID)	Description
0-13	Currently reserved for exclusive use by the SDK and other libraries. If you use these spin locks, you risk breaking SDK or other library functionality. Each reserved spin lock used individually has its own PICO_SPINLOCK_ID so you can search for those.
14,15	(PICO_SPINLOCK_ID_OS1 and PICO_SPINLOCK_ID_OS2). Currently reserved for exclusive use by an operating system (or other system level software) co-existing with the SDK.
16-23	(PICO_SPINLOCK_ID_STRIPED_FIRST - PICO_SPINLOCK_ID_STRIPED_LAST). Spin locks from this range are assigned in a round-robin fashion via next_striped_spin_lock_num(). These spin locks are shared, but assigning numbers from a range reduces the probability that two higher level locking primitives using *striped* spin locks will actually be using the same spin lock.
24-31	(PICO_SPINLOCK_ID_CLAIM_FREE_FIRST - PICO_SPINLOCK_ID_CLAIM_FREE_LAST). These are reserved for exclusive use and are allocated on a first come first served basis at runtime via spin_lock_claim_unused()

Function Documentation

◆ __dmb()

static __force_inline void __dmb (void) static

Insert a DMB instruction in to the code path.

The DMB (data memory barrier) acts as a memory barrier, all memory accesses prior to this instruction will be observed before any explicit access after the instruction.

◆ __dsb()

static __force_inline void __dsb (void) static

Insert a DSB instruction in to the code path.

The DSB (data synchronization barrier) acts as a special kind of data memory barrier (DMB). The DSB operation completes when all explicit memory accesses before this instruction complete.

◆ __isb()

static __force_inline void __isb (void) static

Insert a ISB instruction in to the code path.

ISB acts as an instruction synchronization barrier. It flushes the pipeline of the processor, so that all instructions following the ISB are fetched from cache or memory again, after the ISB instruction has been completed.

◆ __mem_fence_release()

static __force_inline void __mem_fence_release (void) static

Release a memory fence.

◆ __sev()

static __force_inline void __sev (void) static

Insert a SEV instruction in to the code path.

The SEV (send event) instruction sends an event to both cores.

◆ __wfe()

static __force_inline void __wfe (void) static

Insert a WFE instruction in to the code path.

The WFE (wait for event) instruction waits until one of a number of events occurs, including events signalled by the SEV instruction on either core.

◆ __wfi()

static __force_inline void __wfi (void) static

Insert a WFI instruction in to the code path.

The WFI (wait for interrupt) instruction waits for a interrupt to wake up the core.

◆ is_spin_locked()

static bool is_spin_locked (spin_lock_t * ***lock***) inline static

Check to see if a spinlock is currently acquired elsewhere.

Parameters

lock Spinlock instance

◆ next_striped_spin_lock_num()

uint next_striped_spin_lock_num (void)

Return a spin lock number from the *striped* range.

Returns a spin lock number in the range PICO_SPINLOCK_ID_STRIPED_FIRST to PICO_SPINLOCK_ID_STRIPED_LAST in a round robin fashion. This does not grant the caller exclusive access to the spin lock, so the caller must:

1. Abide (with other callers) by the contract of only holding this spin lock briefly (and with IRQs disabled - the default via spin_lock_blocking()), and not whilst holding other spin locks.
2. Be OK with any contention caused by the - brief due to the above requirement - contention with other possible users of the spin lock.

Returns

lock_num a spin lock number the caller may use (non exclusively)

See also PICO_SPINLOCK_ID_STRIPED_FIRST PICO_SPINLOCK_ID_STRIPED_LAST

◆ restore_interrupts()

static __force_inline void restore_interrupts (uint32_t ***status***) static

Restore interrupts to a specified state.

Parameters

status Previous interrupt status from save_and_disable_interrupts()

◆ save_and_disable_interrupts()

static __force_inline uint32_t save_and_disable_interrupts (void) static

Save and disable interrupts.

Returns

The prior interrupt enable status for restoration later via restore_interrupts()

◆ spin_lock_blocking()

static __force_inline uint32_t spin_lock_blocking (spin_lock_t * ***lock***) static

Acquire a spin lock safely.

This function will disable interrupts prior to acquiring the spinlock

Parameters

lock Spinlock instance

Returns

interrupt status to be used when unlocking, to restore to original state

◆ spin_lock_claim()

void spin_lock_claim (uint ***lock_num***)

Mark a spin lock as used.

Method for cooperative claiming of hardware. Will cause a panic if the spin lock is already claimed. Use of this method by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters

lock_num the spin lock number

◆ spin_lock_claim_mask()

void spin_lock_claim_mask (uint32_t ***lock_num_mask***)

Mark multiple spin locks as used.

Method for cooperative claiming of hardware. Will cause a panic if any of the spin locks are already claimed. Use of this method by libraries detects accidental configurations that would fail in unpredictable ways.

Parameters

lock_num_mask Bitfield of all required spin locks to claim (bit 0 == spin lock 0, bit 1 == spin lock 1 etc)

◆ spin_lock_claim_unused()

int spin_lock_claim_unused (bool ***required***)

Claim a free spin lock.

Parameters

required if true the function will panic if none are available

Returns

the spin lock number or -1 if required was false, and none were free

◆ spin_lock_get_num()

static __force_inline uint spin_lock_get_num (spin_lock_t * ***lock***) static

Get HW Spinlock number from instance.

Parameters

lock The Spinlock instance

Returns

The Spinlock ID

◆ spin_lock_init()

spin_lock_t * spin_lock_init (uint ***lock_num***)

Initialise a spin lock.

The spin lock is initially unlocked

Parameters

lock_num The spin lock number

Returns

The spin lock instance

◆ spin_lock_instance()

static __force_inline spin_lock_t * spin_lock_instance (uint ***lock_num***) static

Get HW Spinlock instance from number.

Parameters

lock_num Spinlock ID

Returns

The spinlock instance

◆ spin_lock_is_claimed()

bool spin_lock_is_claimed (uint ***lock_num***)

Determine if a spin lock is claimed.

Parameters

lock_num the spin lock number

Returns

true if claimed, false otherwise

See also spin_lock_claim spin_lock_claim_mask

◆ spin_lock_unclaim()

void spin_lock_unclaim (uint ***lock_num***)

Mark a spin lock as no longer used.

Method for cooperative claiming of hardware.

Parameters

lock_num the spin lock number to release

◆ spin_lock_unsafe_blocking()

static __force_inline void spin_lock_unsafe_blocking (spin_lock_t * ***lock***) static

Acquire a spin lock without disabling interrupts (hence unsafe)

Parameters

lock Spinlock instance

◆ **spin_unlock()**

static __force_inline void spin_unlock (spin_lock_t * ***lock***,
uint32_t ***saved_irq***
) static

Release a spin lock safely.

This function will re-enable interrupts according to the parameters.

Parameters

lock Spinlock instance
saved_irq Return value from the spin_lock_blocking() function.

See also spin_lock_blocking()

◆ **spin_unlock_unsafe()**

static __force_inline void spin_unlock_unsafe (spin_lock_t * ***lock***) static

Release a spin lock without re-enabling interrupts.

Parameters

lock Spinlock instance

hardware_timer

Part of: Hardware APIs

Typedefs

- **`typedef void(* hardware_alarm_callback_t) (uint alarm_num)`**

Functions

- **`static uint32_t time_us_32 (void)`**

  Return a 32 bit timestamp value in microseconds.

- **`uint64_t time_us_64 (void)`**

  Return the current 64 bit timestamp value in microseconds.

- **`void busy_wait_us_32 (uint32_t delay_us)`**

Busy wait wasting cycles for the given (32 bit) number of microseconds.

- **void busy_wait_us (uint64_t delay_us)**

  Busy wait wasting cycles for the given (64 bit) number of microseconds.

- **void busy_wait_ms (uint32_t delay_ms)**

  Busy wait wasting cycles for the given number of milliseconds.

- **void busy_wait_until (absolute_time_t t)**

  Busy wait wasting cycles until after the specified timestamp.

- **static bool time_reached (absolute_time_t t)**

  Check if the specified timestamp has been reached.

- **void hardware_alarm_claim (uint alarm_num)**

  cooperatively claim the use of this hardware alarm_num

- **int hardware_alarm_claim_unused (bool required)**

  cooperatively claim the use of this hardware alarm_num

- **void hardware_alarm_unclaim (uint alarm_num)**

  cooperatively release the claim on use of this hardware alarm_num

- **bool hardware_alarm_is_claimed (uint alarm_num)**

  Determine if a hardware alarm has been claimed.

- **void hardware_alarm_set_callback (uint alarm_num, hardware_alarm_callback_t callback)**

  Enable/Disable a callback for a hardware timer on this core.

- **bool hardware_alarm_set_target (uint alarm_num, absolute_time_t t)**

  Set the current target for the specified hardware alarm.

- **void hardware_alarm_cancel (uint alarm_num)**

  Cancel an existing target (if any) for a given hardware_alarm.

- **void hardware_alarm_force_irq (uint alarm_num)**

  Force and IRQ for a specific hardware alarm.

Detailed Description

Low-level hardware timer API

This API provides medium level access to the timer HW. See also pico_time which provides higher levels functionality using the hardware timer.

The timer peripheral on RP2040 supports the following features:

- single 64-bit counter, incrementing once per microsecond
- Latching two-stage read of counter, for race-free read over 32 bit bus
- Four alarms: match on the lower 32 bits of counter, IRQ on match.

By default the timer uses a one microsecond reference that is generated in the Watchdog (see Section 4.8.2) which is derived from the clk_ref.

The timer has 4 alarms, and can output a separate interrupt for each alarm. The alarms match on the lower 32 bits of the 64 bit counter which means they can be fired a maximum of 2^32 microseconds into the future. This is equivalent to:

- 2^32 ÷ 10^6: ~4295 seconds
- 4295 ÷ 60: ~72 minutes

The timer is expected to be used for short sleeps, if you want a longer alarm see the hardware_rtc functions.

Example

```
#include <stdio.h>
#include "pico/stdlib.h"


volatile bool timer_fired = false;

int64_t alarm_callback(alarm_id_t id, void *user_data) {
    printf("Timer %d fired!\n", (int) id);
    timer_fired = true;
    // Can return a value here in us to fire in the future
    return 0;
}


bool repeating_timer_callback(struct repeating_timer *t) {
    printf("Repeat at %lld\n", time_us_64());
    return true;
}


int main() {
    stdio_init_all();
    printf("Hello Timer!\n");

    // Call alarm_callback in 2 seconds
    add_alarm_in_ms(2000, alarm_callback, NULL, false);

    // Wait for alarm callback to set timer_fired
    while (!timer_fired) {
        tight_loop_contents();
    }

    // Create a repeating timer that calls repeating_timer_callback.
    // If the delay is > 0 then this is the delay between the previous callback ending and the
next starting.
    // If the delay is negative (see below) then the next call to the callback will be exactly
500ms after the
    // start of the call to the last callback
    struct repeating_timer timer;
    add_repeating_timer_ms(500, repeating_timer_callback, NULL, &timer);
    sleep_ms(3000);
    bool cancelled = cancel_repeating_timer(&timer);
    printf("cancelled... %d\n", cancelled);
    sleep_ms(2000);

    // Negative delay so means we will call repeating_timer_callback, and call it again
    // 500ms later regardless of how long the callback took to execute
    add_repeating_timer_ms(-500, repeating_timer_callback, NULL, &timer);
    sleep_ms(3000);
    cancelled = cancel_repeating_timer(&timer);
    printf("cancelled... %d\n", cancelled);
    sleep_ms(2000);
    printf("Done\n");
    return 0;
}
```

See also pico_time

Typedef Documentation

◆ hardware_alarm_callback_t

typedef void(* hardware_alarm_callback_t) (uint alarm_num)

Callback function type for hardware alarms

Parameters

alarm_num the hardware alarm number

See also hardware_alarm_set_callback()

Function Documentation

◆ busy_wait_ms()

void busy_wait_ms (uint32_t ***delay_ms***)

Busy wait wasting cycles for the given number of milliseconds.

Parameters

delay_ms delay amount in milliseconds

◆ busy_wait_until()

void busy_wait_until (absolute_time_t ***t***)

Busy wait wasting cycles until after the specified timestamp.

Parameters

t Absolute time to wait until

◆ busy_wait_us()

void busy_wait_us (uint64_t ***delay_us***)

Busy wait wasting cycles for the given (64 bit) number of microseconds.

Parameters

delay_us delay amount in microseconds

◆ busy_wait_us_32()

void busy_wait_us_32 (uint32_t ***delay_us***)

Busy wait wasting cycles for the given (32 bit) number of microseconds.

Parameters

delay_us delay amount in microseconds

Busy wait wasting cycles for the given (32 bit) number of microseconds.

◆ hardware_alarm_cancel()

void hardware_alarm_cancel (uint ***alarm_num***)

Cancel an existing target (if any) for a given hardware_alarm.

Parameters

alarm_num the hardware alarm number

◆ hardware_alarm_claim()

void hardware_alarm_claim (uint ***alarm_num***)

cooperatively claim the use of this hardware alarm_num

This method hard asserts if the hardware alarm is currently claimed.

Parameters

alarm_num the hardware alarm to claim

See also hardware_claiming

◆ hardware_alarm_claim_unused()

int hardware_alarm_claim_unused (bool ***required***)

cooperatively claim the use of this hardware alarm_num

This method attempts to claim an unused hardware alarm

Returns

alarm_num the hardware alarm claimed or -1 if requires was false, and none are available

See also hardware_claiming

◆ hardware_alarm_force_irq()

void hardware_alarm_force_irq (uint ***alarm_num***)

Force and IRQ for a specific hardware alarm.

This method will forcibly make sure the current alarm callback (if present) for the hardware alarm is called from an IRQ context after this call. If an actual callback is due at the same time then the callback may only be called once.

Calling this method does not otherwise interfere with regular callback operations.

Parameters

alarm_num the hardware alarm number

◆ hardware_alarm_is_claimed()

bool hardware_alarm_is_claimed (uint ***alarm_num***)

Determine if a hardware alarm has been claimed.

Parameters

alarm_num the hardware alarm number

Returns

true if claimed, false otherwise

See also hardware_alarm_claim

◆ hardware_alarm_set_callback()

```
void hardware_alarm_set_callback ( uint                        alarm_num,
                                   hardware_alarm_callback_t   callback
                                 )
```

Enable/Disable a callback for a hardware timer on this core.

This method enables/disables the alarm IRQ for the specified hardware alarm on the calling core, and set the specified callback to be associated with that alarm.

This callback will be used for the timeout set via hardware_alarm_set_target

NOTE

This will install the handler on the current core if the IRQ handler isn't already set. Therefore the user has the opportunity to call this up from the core of their choice

Parameters

alarm_num the hardware alarm number
callback the callback to install, or NULL to unset

See also hardware_alarm_set_target()

◆ hardware_alarm_set_target()

```
bool hardware_alarm_set_target ( uint             alarm_num,
                                 absolute_time_t  t
                               )
```

Set the current target for the specified hardware alarm.

This will replace any existing target

Parameters

alarm_num the hardware alarm number
t the target timestamp

Returns

true if the target was "missed"; i.e. it was in the past, or occurred before a future hardware timeout could be set

◆ hardware_alarm_unclaim()

```
void hardware_alarm_unclaim ( uint alarm_num )
```

cooperatively release the claim on use of this hardware alarm_num

Parameters

alarm_num the hardware alarm to unclaim

See also hardware_claiming

◆ time_reached()

static bool time_reached (absolute_time_t ***t***) inline static

Check if the specified timestamp has been reached.

Parameters

t Absolute time to compare against current time

Returns

true if it is now after the specified timestamp

◆ time_us_32()

static uint32_t time_us_32 (void) inline static

Return a 32 bit timestamp value in microseconds.

Returns the low 32 bits of the hardware timer.

NOTE

This value wraps roughly every 1 hour 11 minutes and 35 seconds.

Returns

the 32 bit timestamp

◆ time_us_64()

uint64_t time_us_64 (void)

Return the current 64 bit timestamp value in microseconds.

Returns the full 64 bits of the hardware timer. The pico_time and other functions rely on the fact that this value monotonically increases from power up. As such it is expected that this value counts upwards and never wraps (we apologize for introducing a potential year 5851444 bug).

Returns

the 64 bit timestamp

Return the current 64 bit timestamp value in microseconds.

hardware_uart

Part of: Hardware APIs

Enumerations

- **enum uart_parity_t { UART_PARITY_NONE , UART_PARITY_EVEN , UART_PARITY_ODD }**

  UART Parity enumeration.

Functions

- **static uint uart_get_index (uart_inst_t *uart)**

  Convert UART instance to hardware instance number.

- **uint uart_init (uart_inst_t *uart, uint baudrate)**

  Initialise a UART.

- **void uart_deinit (uart_inst_t *uart)**

  DeInitialise a UART.

- **uint uart_set_baudrate (uart_inst_t *uart, uint baudrate)**

  Set UART baud rate.

- **static void uart_set_hw_flow (uart_inst_t *uart, bool cts, bool rts)**

  Set UART flow control CTS/RTS.

- **void uart_set_format (uart_inst_t *uart, uint data_bits, uint stop_bits, uart_parity_t parity)**

  Set UART data format.

- **static void uart_set_irq_enables (uart_inst_t *uart, bool rx_has_data, bool tx_needs_data)**

  Setup UART interrupts.

- **static bool uart_is_enabled (uart_inst_t *uart)**

  Test if specific UART is enabled.

- **void uart_set_fifo_enabled (uart_inst_t *uart, bool enabled)**

  Enable/Disable the FIFOs on specified UART.

- **static bool uart_is_writable (uart_inst_t *uart)**

  Determine if space is available in the TX FIFO.

- **static void uart_tx_wait_blocking (uart_inst_t *uart)**

  Wait for the UART TX fifo to be drained.

- **static bool uart_is_readable (uart_inst_t *uart)**

  Determine whether data is waiting in the RX FIFO.

- **static void uart_write_blocking (uart_inst_t *uart, const uint8_t *src, size_t len)**

  Write to the UART for transmission.

- **static void uart_read_blocking (uart_inst_t *uart, uint8_t *dst, size_t len)**

  Read from the UART.

- **static void uart_putc_raw (uart_inst_t *uart, char c)**

Write single character to UART for transmission.

- **static void uart_putc (uart_inst_t *uart, char c)**

  Write single character to UART for transmission, with optional CR/LF conversions.

- **static void uart_puts (uart_inst_t *uart, const char *s)**

  Write string to UART for transmission, doing any CR/LF conversions.

- **static char uart_getc (uart_inst_t *uart)**

  Read a single character from the UART.

- **void uart_set_break (uart_inst_t *uart, bool en)**

  Assert a break condition on the UART transmission.

- **void uart_set_translate_crlf (uart_inst_t *uart, bool translate)**

  Set CR/LF conversion on UART.

- **static void uart_default_tx_wait_blocking (void)**

  Wait for the default UART's TX FIFO to be drained.

- **bool uart_is_readable_within_us (uart_inst_t *uart, uint32_t us)**

  Wait for up to a certain number of microseconds for the RX FIFO to be non empty.

- **static uint uart_get_dreq (uart_inst_t *uart, bool is_tx)**

  Return the DREQ to use for pacing transfers to/from a particular UART instance.

- **#define uart0 ((uart_inst_t *)uart0_hw)**

  Identifier for UART instance 0.

- **#define uart1 ((uart_inst_t *)uart1_hw)**

  Identifier for UART instance 1.

Detailed Description

Hardware UART API

RP2040 has 2 identical instances of a UART peripheral, based on the ARM PL011. Each UART can be connected to a number of GPIO pins as defined in the GPIO muxing.

Only the TX, RX, RTS, and CTS signals are connected, meaning that the modem mode and IrDA mode of the PL011 are not supported.

Example

```
int main() {

    // Initialise UART 0
    uart_init(uart0, 115200);

    // Set the GPIO pin mux to the UART - 0 is TX, 1 is RX
    gpio_set_function(0, GPIO_FUNC_UART);
    gpio_set_function(1, GPIO_FUNC_UART);

    uart_puts(uart0, "Hello world!");
}
```

Macro Definition Documentation

◆ **uart0**

```
#define uart0   ((uart_inst_t *)uart0_hw)
```

Identifier for UART instance 0.

The UART identifiers for use in UART functions.

e.g. uart_init(uart1, 48000)

Function Documentation

◆ **uart_deinit()**

```
void uart_deinit ( uart_inst_t *  uart )
```

DeInitialise a UART.

Disable the UART if it is no longer used. Must be reinitialised before being used again.

Parameters

uart UART instance. uart0 or uart1

◆ **uart_get_dreq()**

```
static uint uart_get_dreq ( uart_inst_t *  uart,
                            bool           is_tx
                          )                                    inline  static
```

Return the DREQ to use for pacing transfers to/from a particular UART instance.

Parameters

uart UART instance. uart0 or uart1
is_tx true for sending data to the UART instance, false for receiving data from the UART instance

◆ **uart_get_index()**

```
static uint uart_get_index ( uart_inst_t *  uart )             inline  static
```

Convert UART instance to hardware instance number.

Parameters

uart UART instance

Returns

Number of UART, 0 or 1.

◆ **uart_getc()**

static char uart_getc (uart_inst_t * ***uart***) inline static

Read a single character from the UART.

This function will block until a character has been read

Parameters

uart UART instance. uart0 or uart1

Returns

The character read.

◆ uart_init()

uint uart_init (uart_inst_t * ***uart***,
uint ***baudrate***
)

Initialise a UART.

Put the UART into a known state, and enable it. Must be called before other functions.

This function always enables the FIFOs, and configures the UART for the following default line format:

- 8 data bits
- No parity bit
- One stop bit

NOTE

There is no guarantee that the baudrate requested will be possible, the nearest will be chosen, and this function will return the configured baud rate.

Parameters

uart UART instance. uart0 or uart1
baudrate Baudrate of UART in Hz

Returns

Actual set baudrate

◆ uart_is_enabled()

static bool uart_is_enabled (uart_inst_t * ***uart***) inline static

Test if specific UART is enabled.

Parameters

uart UART instance. uart0 or uart1

Returns

true if the UART is enabled

◆ uart_is_readable()

```
static bool uart_is_readable ( uart_inst_t * uart )
```

inline static

Determine whether data is waiting in the RX FIFO.

Parameters

uart UART instance. uart0 or uart1

Returns

true if the RX FIFO is not empty, otherwise false.

◆ uart_is_readable_within_us()

```
bool uart_is_readable_within_us ( uart_inst_t * uart,
                                  uint32_t      us
                                )
```

Wait for up to a certain number of microseconds for the RX FIFO to be non empty.

Parameters

uart UART instance. uart0 or uart1
us the number of microseconds to wait at most (may be 0 for an instantaneous check)

Returns

true if the RX FIFO became non empty before the timeout, false otherwise

◆ uart_is_writable()

```
static bool uart_is_writable ( uart_inst_t * uart )
```

inline static

Determine if space is available in the TX FIFO.

Parameters

uart UART instance. uart0 or uart1

Returns

false if no space available, true otherwise

◆ uart_putc()

```
static void uart_putc ( uart_inst_t * uart,
                        char          c
                      )
```

inline static

Write single character to UART for transmission, with optional CR/LF conversions.

This function will block until the character has been sent

Parameters

uart	UART instance. uart0 or uart1
c	The character to send

◆ uart_putc_raw()

```
static void uart_putc_raw ( uart_inst_t * uart,
                            char          c
                          )                                    inline  static
```

Write single character to UART for transmission.

This function will block until the entire character has been sent

Parameters

uart	UART instance. uart0 or uart1
c	The character to send

◆ uart_puts()

```
static void uart_puts ( uart_inst_t * uart,
                        const char *  s
                      )                                        inline  static
```

Write string to UART for transmission, doing any CR/LF conversions.

This function will block until the entire string has been sent

Parameters

uart	UART instance. uart0 or uart1
s	The null terminated string to send

◆ uart_read_blocking()

```
static void uart_read_blocking ( uart_inst_t * uart,
                                 uint8_t *     dst,
                                 size_t        len
                               )                               inline  static
```

Read from the UART.

This function blocks until len characters have been read from the UART

Parameters

uart	UART instance. uart0 or uart1
dst	Buffer to accept received bytes
len	The number of bytes to receive.

◆ uart_set_baudrate()

```
uint uart_set_baudrate ( uart_inst_t *  uart,
                         uint           baudrate
                       )
```

Set UART baud rate.

Set baud rate as close as possible to requested, and return actual rate selected.

The UART is paused for around two character periods whilst the settings are changed. Data received during this time may be dropped by the UART.

Any characters still in the transmit buffer will be sent using the new updated baud rate. uart_tx_wait_blocking() can be called before this function to ensure all characters at the old baud rate have been sent before the rate is changed.

This function should not be called from an interrupt context, and the UART interrupt should be disabled before calling this function.

Parameters

uart	UART instance. uart0 or uart1
baudrate	Baudrate in Hz

Returns

Actual set baudrate

◆ uart_set_break()

```
void uart_set_break ( uart_inst_t *  uart,
                      bool           en
                    )
```

Assert a break condition on the UART transmission.

Parameters

uart	UART instance. uart0 or uart1
en	Assert break condition (TX held low) if true. Clear break condition if false.

◆ uart_set_fifo_enabled()

```
void uart_set_fifo_enabled ( uart_inst_t *  uart,
                             bool           enabled
                           )
```

Enable/Disable the FIFOs on specified UART.

The UART is paused for around two character periods whilst the settings are changed. Data received during this time may be dropped by the UART.

Any characters still in the transmit FIFO will be lost if the FIFO is disabled. uart_tx_wait_blocking() can be called before this function to avoid this.

This function should not be called from an interrupt context, and the UART interrupt should be disabled when calling this function.

Parameters

uart	UART instance. uart0 or uart1

enabled true to enable FIFO (default), false to disable

◆ uart_set_format()

```
void uart_set_format ( uart_inst_t *  uart,
                       uint           data_bits,
                       uint           stop_bits,
                       uart_parity_t  parity
                     )
```

Set UART data format.

Configure the data format (bits etc) for the UART.

The UART is paused for around two character periods whilst the settings are changed. Data received during this time may be dropped by the UART.

Any characters still in the transmit buffer will be sent using the new updated data format. uart_tx_wait_blocking() can be called before this function to ensure all characters needing the old format have been sent before the format is changed.

This function should not be called from an interrupt context, and the UART interrupt should be disabled before calling this function.

Parameters

uart	UART instance. uart0 or uart1
data_bits	Number of bits of data. 5..8
stop_bits	Number of stop bits 1..2
parity	Parity option.

◆ uart_set_hw_flow()

```
static void uart_set_hw_flow ( uart_inst_t *  uart,
                               bool           cts,
                               bool           rts
                             )                                    inline  static
```

Set UART flow control CTS/RTS.

Parameters

uart	UART instance. uart0 or uart1
cts	If true enable flow control of TX by clear-to-send input
rts	If true enable assertion of request-to-send output by RX flow control

◆ uart_set_irq_enables()

```
static void uart_set_irq_enables ( uart_inst_t *  uart,
                                   bool           rx_has_data,
                                   bool           tx_needs_data
                                 )                                inline  static
```

Setup UART interrupts.

Enable the UART's interrupt output. An interrupt handler will need to be installed prior to calling this function.

Parameters

uart	UART instance. uart0 or uart1
rx_has_data	If true an interrupt will be fired when the RX FIFO contains data.
tx_needs_data	If true an interrupt will be fired when the TX FIFO needs data.

◆ **uart_set_translate_crlf()**

```
void uart_set_translate_crlf ( uart_inst_t *  uart,
                               bool           translate
                             )
```

Set CR/LF conversion on UART.

Parameters

uart	UART instance. uart0 or uart1
translate	If true, convert line feeds to carriage return on transmissions

◆ **uart_tx_wait_blocking()**

```
static void uart_tx_wait_blocking ( uart_inst_t *  uart )                    inline static
```

Wait for the UART TX fifo to be drained.

Parameters

uart	UART instance. uart0 or uart1

◆ **uart_write_blocking()**

```
static void uart_write_blocking ( uart_inst_t *       uart,
                                  const uint8_t *     src,
                                  size_t              len
                                )                                            inline static
```

Write to the UART for transmission.

This function will block until all the data has been sent to the UART

Parameters

uart	UART instance. uart0 or uart1
src	The bytes to send
len	The number of bytes to send

hardware_vreg

Part of: Hardware APIs

Functions

- **`void vreg_set_voltage (enum vreg_voltage voltage)`**

  Set voltage.

Detailed Description

Voltage Regulation API

Function Documentation

◆ vreg_set_voltage()

void vreg_set_voltage (enum vreg_voltage ***voltage***)

Set voltage.

Parameters

voltage The voltage (from enumeration vreg_voltage) to apply to the voltage regulator

hardware_watchdog

Part of: Hardware APIs

Functions

- **void watchdog_reboot (uint32_t pc, uint32_t sp, uint32_t delay_ms)**

  Define actions to perform at watchdog timeout.

- **void watchdog_start_tick (uint cycles)**

  Start the watchdog tick.

- **void watchdog_update (void)**

  Reload the watchdog counter with the amount of time set in watchdog_enable.

- **void watchdog_enable (uint32_t delay_ms, bool pause_on_debug)**

  Enable the watchdog.

- **bool watchdog_caused_reboot (void)**

  Did the watchdog cause the last reboot?

- **bool watchdog_enable_caused_reboot (void)**

  Did watchdog_enable cause the last reboot?

- **uint32_t watchdog_get_count (void)**

  Returns the number of microseconds before the watchdog will reboot the chip.

Detailed Description

Hardware Watchdog Timer API

Supporting functions for the Pico hardware watchdog timer.

The RP2040 has a built in HW watchdog Timer. This is a countdown timer that can restart parts of the chip if it reaches zero. For example, this can be used to restart the processor if the software running on it gets stuck in an infinite loop or similar. The programmer has to periodically write a value to the watchdog to stop it reaching zero.

Example

```
#include <stdio.h>
#include "pico/stdlib.h"
#include "hardware/watchdog.h"


int main() {
    stdio_init_all();

    if (watchdog_caused_reboot()) {
        printf("Rebooted by Watchdog!\n");
        return 0;
    } else {
        printf("Clean boot\n");
    }

    // Enable the watchdog, requiring the watchdog to be updated every 100ms or the chip will r
eboot
    // second arg is pause on debug which means the watchdog will pause when stepping through c
ode
    watchdog_enable(100, 1);

    for (uint i = 0; i < 5; i++) {
        printf("Updating watchdog %d\n", i);
        watchdog_update();
    }

    // Wait in an infinite loop and don't update the watchdog so it reboots us
    printf("Waiting to be rebooted by watchdog\n");
    while(1);
}
```

Function Documentation

◆ watchdog_caused_reboot()

bool watchdog_caused_reboot (void)

Did the watchdog cause the last reboot?

Returns

true If the watchdog timer or a watchdog force caused the last reboot

false If there has been no watchdog reboot since the last power on reset. A power on reset is typically caused by a power cycle or the run pin (reset button) being toggled.

◆ watchdog_enable()

void watchdog_enable (uint32_t ***delay_ms***,
bool ***pause_on_debug***
)

Enable the watchdog.

NOTE

If watchdog_start_tick value does not give a 1MHz clock to the watchdog system, then the `delay_ms` parameter will not be in microseconds. See the datasheet for more details.

By default the SDK assumes a 12MHz XOSC and sets the watchdog_start_tick appropriately.

This method sets a marker in the watchdog scratch register 4 that is checked by watchdog_enable_caused_reboot. If the device is subsequently reset via a call to watchdog_reboot (including for example by dragging a UF2 onto the RPI-RP2), then this value will be cleared, and so watchdog_enable_caused_reboot will return false.

Parameters

delay_ms	Number of milliseconds before watchdog will reboot without watchdog_update being called. Maximum of 0x7fffff, which is approximately 8.3 seconds
pause_on_debug	If the watchdog should be paused when the debugger is stepping through code

◆ watchdog_enable_caused_reboot()

```
bool watchdog_enable_caused_reboot ( void )
```

Did watchdog_enable cause the last reboot?

Perform additional checking along with watchdog_caused_reboot to determine if a watchdog timeout initiated by watchdog_enable caused the last reboot.

This method checks for a special value in watchdog scratch register 4 placed there by watchdog_enable. This would not be present if a watchdog reset is initiated by watchdog_reboot or by the RP2040 bootrom (e.g. dragging a UF2 onto the RPI-RP2 drive).

Returns

true If the watchdog timer or a watchdog force caused (see watchdog_caused_reboot) the last reboot and the watchdog reboot happened after watchdog_enable was called

false If there has been no watchdog reboot since the last power on reset, or the watchdog reboot was not caused by a watchdog timeout after watchdog_enable was called. A power on reset is typically caused by a power cycle or the run pin (reset button) being toggled.

◆ watchdog_get_count()

```
uint32_t watchdog_get_count ( void )
```

Returns the number of microseconds before the watchdog will reboot the chip.

Returns

The number of microseconds before the watchdog will reboot the chip.

◆ watchdog_reboot()

```
void watchdog_reboot ( uint32_t pc,
                       uint32_t sp,
                       uint32_t delay_ms
                     )
```

Define actions to perform at watchdog timeout.

NOTE

If watchdog_start_tick value does not give a 1MHz clock to the watchdog system, then the `delay_ms` parameter will not be in microseconds. See the datasheet for more details.

By default the SDK assumes a 12MHz XOSC and sets the watchdog_start_tick appropriately.

Parameters

pc	If Zero, a standard boot will be performed, if non-zero this is the program counter to jump to on reset.
sp	If `pc` is non-zero, this will be the stack pointer used.
delay_ms	Initial load value. Maximum value 0x7fffff, approximately 8.3s.

◆ **watchdog_start_tick()**

void watchdog_start_tick (uint ***cycles***)

Start the watchdog tick.

Parameters

cycles This needs to be a divider that when applied to the XOSC input, produces a 1MHz clock. So if the XOSC is 12MHz, this will need to be 12.

◆ **watchdog_update()**

void watchdog_update (void)

Reload the watchdog counter with the amount of time set in watchdog_enable.

hardware_xosc

Part of: Hardware APIs

Functions

- **void xosc_init (void)**

  Initialise the crystal oscillator system.

- **void xosc_disable (void)**

  Disable the Crystal oscillator.

- **void xosc_dormant (void)**

  Set the crystal oscillator system to dormant.

Detailed Description

Crystal Oscillator (XOSC) API

Function Documentation

◆ **xosc_disable()**

void xosc_disable (void)

Disable the Crystal oscillator.

Turns off the crystal oscillator source, and waits for it to become unstable

◆ **xosc_dormant()**

void xosc_dormant (void)

Set the crystal oscillator system to dormant.

Turns off the crystal oscillator until it is woken by an interrupt. This will block and hence the entire system will stop, until an interrupt wakes it up. This function will continue to block until the oscillator becomes stable after its wakeup.

◆ xosc_init()

```
void xosc_init ( void )
```

Initialise the crystal oscillator system.

This function will block until the crystal oscillator has stabilised.