ZS110A-API-Reference

发布 ***0.1.0***

2018 年 **11** 月 **02** 日

目录

CHAPTER 1

文档介绍

1.1 文档目的

介绍 ZS110A SDK 提供的 API，供用户开发时查阅。

1.2 术语说明

表 1: 术语说明

术语	说明
ZEPHYR	为所有资源受限设备，构建了针对低功耗、小型内存微处理器设备而进行优化的物联网嵌入式小型、可扩展的实时操作系统（RTOS）
API	Application Programming Interface, 应用程序编程接口

1.3 参考文档

- http://docs.zephyrproject.org/

1.4 版本历史

表 2: 版本历史

日期	版本	注释	作者
2018-08-22	1.0	初始版本	ZS110A 项目组

CHAPTER 2

API 概述

ZS110A 是基于 Zephyr OS 为 ATB110X Soc 开发的 SDK。

ZS110A 可以分为以下几个层次：

- kernel service (from Zephyr OS)
- hardware device driver (developed by Actions for ATB110X Soc)
- library (developed by Actions for ZS110A sample)

API 文档对应的将分成三个部分：

- Kernel APIs
- Driver APIs
- Library APIs

CHAPTER 3

Kernel APIs

ZS110A 中使用到的很多的 zephyr kernel service。下面以链接的方式给出相应的官方 API 文档。

3.1 Thread API

请参考 https://docs.zephyrproject.org/1.9.0/api/kernel_api.html#threads

3.2 Workqueue API

请参考 https://docs.zephyrproject.org/1.9.0/api/kernel_api.html#workqueues

3.3 Clock API

请参考 https://docs.zephyrproject.org/1.9.0/api/kernel_api.html#clocks

3.4 Timer API

请参考 https://docs.zephyrproject.org/1.9.0/api/kernel_api.html#timers

3.5 Memory Slab API

请参考 https://docs.zephyrproject.org/1.9.0/api/kernel_api.html#memory-slabs

3.6 Heap Momory Pool API

请参考 https://docs.zephyrproject.org/1.9.0/api/kernel_api.html#heap-memory-pool

3.7 Semaphore API

请参考 https://docs.zephyrproject.org/1.9.0/api/kernel_api.html#semaphores

3.8 Mutex API

请参考 https://docs.zephyrproject.org/1.9.0/api/kernel_api.html#mutexes

3.9 Alert API

请参考 https://docs.zephyrproject.org/1.9.0/api/kernel_api.html#alerts

3.10 Message Queue API

请参考 https://docs.zephyrproject.org/1.9.0/api/kernel_api.html#message-queues

3.11 Mailbox API

请参考 https://docs.zephyrproject.org/1.9.0/api/kernel_api.html#mailboxes

3.12 Pipe API

请参考 https://docs.zephyrproject.org/1.9.0/api/kernel_api.html#pipes

3.13 Atomic API

请参考 https://docs.zephyrproject.org/1.9.0/api/kernel_api.html#atomic-services

3.14 Network buffer API

请参考 https://docs.zephyrproject.org/1.9.0/api/networking.html#network-buffers

CHAPTER 4

Driver APIs

4.1 Adc API

ADC public API header file.

Functions

static void **adc_enable**(**struct** device *_dev_)

Enable ADC hardware.

This routine enables the ADC hardware block for data sampling for the specified device.

Return N/A

Parameters

- `dev`: Pointer to the device structure for the driver instance.

static void **adc_disable**(**struct** device *_dev_)

Disable ADC hardware.

This routine disables the ADC hardware block for data sampling for the specified device.

Return N/A

Parameters

- `dev`: Pointer to the device structure for the driver instance.

static int **adc_read**(**struct** device *_dev_, **struct** *adc_seq_table* *_seq_table_)

Set a read request.

This routine sends a read or sampling request to the ADC hardware block. A sequence table describes the read request. The routine returns once the ADC completes the read sequence. The sample data can be retrieved from the memory buffers in the sequence table structure.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `seq_table`: Pointer to the structure representing the sequence table.

Return Value

- `0`: On success
- `else`: Otherwise.

struct adc_seq_entry

#include <adc.h> ADC driver Sequence entry.

This structure defines a sequence entry used to define a sample from a specific channel.

Public Members

s32_t **sampling_delay**

Clock ticks delay before sampling the ADC.

u8_t ***buffer**

Buffer pointer where the sample is written.

u32_t **buffer_length**

Length of the sampling buffer.

u8_t **channel_id**

Channel ID that should be sampled from the ADC

u8_t **stride**[3]

struct adc_seq_table

#include <adc.h> ADC driver Sequence table.

This structure defines a list of sequence entries used to execute a sequence of samplings.

Public Members

struct *adc_seq_entry* ***entries**

u8_t **num_entries**

u8_t **stride**[3]

struct adc_driver_api

#include <adc.h> ADC driver API.

This structure holds all API function pointers.

Public Members

void (***enable**)(**struct** device *dev)

Pointer to the enable routine.

void (***disable**)(**struct** device *dev)

Pointer to the disable routine.

int (***read**)(**struct** device *dev, **struct** *adc_seq_table* *seq_table)

Pointer to the read routine.

4.2 Audio API

Audio public API header file.

Typedefs

typedef void (***pcmbuf_irq_cbk**)(uint32_t pending)

Enums

enum ain_source_type_e

Values:

AIN_SOURCE_AUX = 0

AIN_SOURCE_FM = 1

AIN_SOURCE_MIC = 2

enum mic_op_gain_e

Values:

MIC_OP_0DB = 0

MIC_OP_3DB = 1

MIC_OP_6DB = 2

MIC_OP_9DB = 3

MIC_OP_12DB = 4

MIC_OP_15DB = 5

MIC_OP_18DB = 6

MIC_OP_21DB = 7

MIC_OP_15DB_1 = 8

MIC_OP_18DB_1 = 9

MIC_OP_21DB_1 = 10

MIC_OP_24DB = 11

MIC_OP_27DB = 12

MIC_OP_30DB = 13

MIC_OP_33DB = 14

MIC_OP_36DB = 15

enum ain_op_gain_e

Values:

AIN_OP_M12DB = 0

AIN_OP_M3DB = 1

AIN_OP_0DB = 2

AIN_OP_1DB5 = 3

AIN_OP_3DB = 4

AIN_OP_4DB5 = 5

AIN_OP_6DB = 6

AIN_OP_7DB5 = 7

enum adc_gain_e

Values:

ADC_GAIN_0DB = 0

ADC_GAIN_0DB0 = 1

ADC_GAIN_6DB = 2

ADC_GAIN_12DB = 3

Functions

void **audio_in_enable**(**struct** device **dev*, *ain_setting_t* **ain_setting*, *adc_setting_t* **setting*)

Enables adc channel and starts adc transfer, the pcmbuf must be configured beforehand.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `ain_setting`: config of adc analog
- *`adc_setting_t`*: config of adc digital

void **audio_in_disable(struct** device *_dev_**)**
Stops the adc transfer and disables adc channel.

Parameters

- `dev`: Pointer to the device structure for the driver instance.

void **audio_in_pcmbuf_config(struct** device *_dev_, *pcmbuf_setting_t* *_setting_, uint16_t *_pcm_buf_, u32_t _data_cnt_**)**
Configure pcm_buf for ADC DMA transfer.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `setting`: config for adc dma
- `pcm_buf`: buffer for ac dma from adc to pcm_buf
- `data_cnt`: buffer size of pcm_buf

void **audio_in_pcmbuf_transfer_config(struct** device *_dev_, uint16_t *_pcm_buf_, u32_t _data_cnt_**)**
Configure ADC DMA transfer for a specific channel that has been configured.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `pcm_buf`: buffer for ac dma from adc to pcm_buf
- `data_cnt`: buffer size of pcm_buf

struct pcmbuf_setting_t

Public Members

pcmbuf_irq_cbk **irq_cbk**

uint16_t **half_empty_thres**

uint16_t **half_full_thres**

bool **he_ie**

bool **ep_ie**

bool **hf_ie**

bool **fu_ie**

union op_gain_u

Public Members

mic_op_gain_e **mic_op_gain**

ain_op_gain_e **ain_op_gain**

struct ain_setting_t

Public Members

ain_source_type_e **ain_src**

op_gain_u **op_gain**

u8_t **ain_input_mode**

struct adc_setting_t

Public Members

u8_t **sample_rate**

adc_gain_e **gain**

4.3 Dma API

DMA public API header file.

Enums

enum dma_transfer_width

Values:

TRANS_WIDTH_8 = 0x0

TRANS_WIDTH_16

TRANS_WIDTH_32

enum dma_channel_direction

Values:

MEMORY_TO_MEMORY = 0x0

MEMORY_TO_PERIPHERAL

PERIPHERAL_TO_MEMORY

Functions

static int **dma_config**(**struct** device *_dev_, u32_t _channel_, **struct** *dma_config* *_config_)

Configure individual channel for DMA transfer.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `channel`: Numeric identification of the channel to configure
- `config`: Data structure containing the intended configuration for the selected channel

Return Value

- `0`: if successful.
- `Negative`: errno code if failure.

static int **dma_start**(**struct** device *_dev_, u32_t _channel_)

Enables DMA channel and starts the transfer, the channel must be configured beforehand.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `channel`: Numeric identification of the channel where the transfer will be processed

Return Value

- `0`: if successful.
- `Negative`: errno code if failure.

static int **dma_stop**(**struct** device *_dev_, u32_t _channel_)

Stops the DMA transfer and disables the channel.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `channel`: Numeric identification of the channel where the transfer was being processed

Return Value

- `0`: if successful.

- `Negative`: errno code if failure.

static int **dma_transfer_config(struct** device *_dev_, u32_t _channel_, **struct** _dma_transfer_config_ *_config_**)**

Configure DMA transfer for a specific channel that has been configured.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `channel`: Numeric identification of the channel to configure
- `config`: Data structure containing transfer configuration for the selected channel

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **dma_transfer_start(struct** device *_dev_, u32_t _channel_**)**

Enables DMA channel and starts the transfer, the channel must be configured beforehand.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `channel`: Numeric identification of the channel where the transfer will be processed

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **dma_transfer_stop(struct** device *_dev_, u32_t _channel_**)**

Stops the DMA transfer and disables the channel.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `channel`: Numeric identification of the channel where the transfer was being processed

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

struct dma_transfer_config

#include <dma.h> DMA transfer Configuration.

This defines a single transfer on configured channel of the DMA controller.

Public Members

u32_t **block_size**
Total amount of data in bytes to transfer

u32_t ***source_address**
Source address for the transaction

u32_t ***destination_address**
Destination address

struct dma_config

#include <dma.h> DMA configuration structure.

Public Members

u32_t **dma_slot**
which peripheral and direction

u32_t **channel_direction**
000-memory to memory, 001-memory to peripheral, 010-peripheral to memory

u32_t **complete_callback_en**
0-no callback invoked at completion , 1-callback invoked at completion

u32_t **half_complete_callback_en**
0-no callback invoked at completion half transmission, 1-callback invoked at completion half transmission

u32_t **error_callback_en**
0-error callback enabled, 1-error callback disabled

u32_t **source_handshake**
0-HW, 1-SW

u32_t **dest_handshake**
0-HW, 1-SW

u32_t **channel_priority**
DMA channel priority

u32_t **source_chaining_en**
enable/disable source block chaining

u32_t **dest_chaining_en**
enable/disable destination block

u32_t **reserved**

u32_t **source_data_size**
: width of source data (in bytes)

u32_t **dest_data_size**
: width of dest data (in bytes)

u32_t **source_burst_length**
: number of source data units

u32_t **dest_burst_length**
: number of destination data units

u32_t **block_count**
: the number of blocks used for block chaining

struct dma_block_config ***head_block**
: first block config

void (***dma_callback**)(**struct** device *dev, u32_t channel, int reason)
: the callback function pointer

void ***callback_data**
: Client callback private data

4.4 Pinmux API

PINMUX public API header file.

Functions

int **acts_pinmux_set**(unsigned int *pin*, unsigned int *mode*)
: Set a pin function mode.

Parameters

- `pin`: Pin number.
- `mode`: Pin function.

Return Value

- `0`: On success
- `else`: Otherwise.

int **acts_pinmux_get**(unsigned int *pin*, unsigned int **mode*)
: Get a pin function mode.

Parameters

- `pin`: Pin number.

- `mode`: Pointer to save pin function.

Return Value

- `0`: On success
- `else`: Otherwise.

void **acts_pinmux_setup_pins**(**const struct** *acts_pin_config* *pinconf, int *pins*)

Set function mode for multiple pins.

Parameters

- `pinconf`: Fuction definations of multiple pins.
- `pins`: Pin count.

struct acts_pin_config

Public Members

unsigned int **pin_num**

Pin number.

unsigned int **mode**

Pin function mode.

4.5 Flash API

FLASH public API header file.

Functions

static int **flash_read**(**struct** device *dev, off_t *offset*, void *data, size_t *len*)

Read data from flash.

Return 0 on success, negative errno code on fail.

Parameters

- `dev`: : flash dev
- `offset`: : Offset (byte aligned) to read
- `data`: : Buffer to store read data
- `len`: : Number of bytes to read.

static int **flash_write(struct** device *_dev_, off_t _offset_, **const** void *_data_, size_t _len_**)**

Write buffer into flash memory.

Prior to the invocation of this API, the flash_write_protection_set needs to be called first to disable the write protection.

Return 0 on success, negative errno code on fail.

Parameters

- `dev`: : flash device
- `offset`: : starting offset for the write
- `data`: : data to write
- `len`: : Number of bytes to write

static int **flash_erase(struct** device *_dev_, off_t _offset_, size_t _size_**)**

Erase part or all of a flash memory.

Acceptable values of erase size and offset are subject to hardware-specific multiples of sector size and offset. Please check the API implemented by the underlying sub driver.

Prior to the invocation of this API, the flash_write_protection_set needs to be called first to disable the write protection.

Return 0 on success, negative errno code on fail.

Parameters

- `dev`: : flash device
- `offset`: : erase area starting offset
- `size`: : size of area to be erased

static int **flash_write_protection_set(struct** device *_dev_, bool _enable_**)**

Enable or disable write protection for a flash memory.

This API is required to be called before the invocation of write or erase API. Please note that on some flash components, the write protection is automatically turned on again by the device after the completion of each write or erase calls. Therefore, on those flash parts, write protection needs to be disabled before each invocation of the write or erase API. Please refer to the sub-driver API or the data sheet of the flash component to get details on the write protection behavior.

Return 0 on success, negative errno code on fail.

Parameters

- `dev`: : flash device
- `enable`: : enable or disable flash write protection

4.6 Gpio API

GPIO public API header file.

Typedefs

typedef gpio_callback_handler_t

Define the application callback handler function signature.

Note: cb pointer can be used to retrieve private data through CONTAINER_OF() if original struct *gpio_callback* is stored in another private structure.

Parameters

- `struct device *port`: Device struct for the GPIO device.
- `struct gpio_callback *cb`: Original struct *gpio_callback* owning this handler
- `u32_t pins`: Mask of pins that triggers the callback handler

Functions

static int **gpio_pin_configure(struct** device *_port_, u32_t _pin_, int _flags_**)**

Configure a single pin.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to device structure for the driver instance.
- `pin`: Pin number to configure.
- `flags`: Flags for pin configuration. IN/OUT, interrupt ···

static int **gpio_pin_write(struct** device *_port_, u32_t _pin_, u32_t _value_**)**

Write the data value to a single pin.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `pin`: Pin number where the data is written.
- `value`: Value set on the pin.

static int **gpio_pin_read(struct** device *_port_, u32_t _pin_, u32_t *_value_**)**

Read the data value of a single pin.

Read the input state of a pin, returning the value 0 or 1.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `pin`: Pin number where data is read.
- `value`: Integer pointer to receive the data values from the pin.

static void **gpio_init_callback(struct** *gpio_callback* **callback*, *gpio_callback_handler_t* *handler*, u32_t *pin_mask***)**

Helper to initialize a struct *gpio_callback* properly.

Parameters

- `callback`: A valid Application's callback structure pointer.
- `handler`: A valid handler function pointer.
- `pin_mask`: A bit mask of relevant pins for the handler

static int **gpio_add_callback(struct** device **port*, **struct** *gpio_callback* **callback***)**

Add an application callback.

Note: enables to add as many callback as needed on the same port.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `callback`: A valid Application's callback structure pointer.

static int **gpio_remove_callback(struct** device **port*, **struct** *gpio_callback* **callback***)**

Remove an application callback.

Note: enables to remove as many callbacks as added through gpio_add_callback().

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `callback`: A valid application's callback structure pointer.

static int **gpio_pin_enable_callback(struct** device **port*, u32_t *pin***)**

Enable callback(s) for a single pin.

Note: Depending on the driver implementation, this function will enable the pin to trigger an interruption. So as a semantic detail, if no callback is registered, of course none will be called.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `pin`: Pin number where the callback function is enabled.

static int **gpio_pin_disable_callback(struct** device *_port_, u32_t _pin_**)**
Disable callback(s) for a single pin.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `pin`: Pin number where the callback function is disabled.

static int **gpio_port_configure(struct** device *_port_, int _flags_**)**
Configure all the pins the same way in the port. List out all flags on the detailed description.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `flags`: Flags for the port configuration. IN/OUT, interrupt ···

static int **gpio_port_write(struct** device *_port_, u32_t _value_**)**
Write a data value to the port.

Write the output state of a port. The state of each pin is represented by one bit in the value. Pin 0 corresponds to the least significant bit, pin 31 corresponds to the most significant bit. For ports with less that 32 physical pins the most significant bits which do not correspond to a physical pin are ignored.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `value`: Value to set on the port.

static int **gpio_port_read(struct** device *_port_, u32_t *_value_**)**
Read data value from the port.

Read the input state of a port. The state of each pin is represented by one bit in the returned value. Pin 0 corresponds to the least significant bit, pin 31 corresponds to the most significant bit. Unused bits for ports with less that 32 physical pins are returned as 0.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `value`: Integer pointer to receive the data value from the port.

static int **gpio_port_enable_callback(struct** device *_port_**)**

Enable callback(s) for the port.

Note: Depending on the driver implementation, this function will enable the port to trigger an interruption on all pins, as long as these are configured properly. So as a semantic detail, if no callback is registered, of course none will be called.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.

static int **gpio_port_disable_callback(struct** device *_port_**)**

Disable callback(s) for the port.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.

static int **gpio_get_pending_int(struct** device *_dev_**)**

Function to get pending interrupts.

The purpose of this function is to return the interrupt status register for the device. This is especially useful when waking up from low power states to check the wake up source.

Parameters

- `dev`: Pointer to the device structure for the driver instance.

Return Value

- `status`: != 0 if at least one gpio interrupt is pending.
- `0`: if no gpio interrupt is pending.

struct gpio_callback

#include <gpio.h> GPIO callback structure.

Used to register a callback in the driver instance callback list. As many callbacks as needed can be added as long as each of them are unique pointers of struct *gpio_callback*. Beware such structure should not be allocated on stack.

Note: To help setting it, see gpio_init_callback() below

Public Members

sys_snode_t **node**
: This is meant to be used in the driver and the user should not mess with it (see drivers/gpio/gpio_utils.h)

gpio_callback_handler_t **handler**
: Actual callback function being called when relevant.

u32_t **pin_mask**
: A mask of pins the callback is interested in, if 0 the callback will never be called. Such pin_mask can be modified whenever necessary by the owner, and thus will affect the handler being called or not. The selected pins must be configured to trigger an interrupt.

4.7 I2c API

I2C public API header file.

Functions

static int **i2c_configure(struct** device *_dev_, u32_t _dev_config_**)**
Configure operation of a host controller.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- *`dev_config`*: Bit-packed 32-bit value to the device runtime configuration for the I2C controller.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error, failed to configure device.

static int **i2c_write(struct** device *_dev_, u8_t *_buf_, u32_t _num_bytes_, u16_t _addr_**)**
Write a set amount of data to an I2C device.

This routine writes a set amount of data synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `buf`: Memory pool from which the data is transferred.
- `num_bytes`: Number of bytes to write.

- `addr`: Address to the target I2C device for writing.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_read**(**struct** device *_dev_, u8_t *_buf_, u32_t _num_bytes_, u16_t _addr_)

Read a set amount of data from an I2C device.

This routine reads a set amount of data synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `buf`: Memory pool that stores the retrieved data.
- `num_bytes`: Number of bytes to read.
- `addr`: Address of the I2C device being read.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_transfer**(**struct** device *_dev_, **struct** *i2c_msg* *_msgs_, u8_t _num_msgs_, u16_t _addr_)

Perform data transfer to another I2C device.

This routine provides a generic interface to perform data transfer to another I2C device synchronously. Use i2c_read()/i2c_write() for simple read or write.

The array of message *msgs* must not be NULL. The number of message *num_msgs* may be zero,in which case no transfer occurs.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `msgs`: Array of messages to transfer.
- `num_msgs`: Number of messages to transfer.
- `addr`: Address of the I2C target device.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_burst_read**(**struct** device **dev*, u16_t *dev_addr*, u8_t *start_addr*, u8_t **buf*, u8_t *num_bytes*)

Read multiple bytes from an internal address of an I2C device.

This routine reads multiple bytes from an internal address of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for reading.
- `start_addr`: Internal address from which the data is being read.
- `buf`: Memory pool that stores the retrieved data.
- `num_bytes`: Number of bytes being read.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_burst_write**(**struct** device **dev*, u16_t *dev_addr*, u8_t *start_addr*, u8_t **buf*, u8_t *num_bytes*)

Write multiple bytes to an internal address of an I2C device.

This routine writes multiple bytes to an internal address of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for writing.
- `start_addr`: Internal address to which the data is being written.
- `buf`: Memory pool from which the data is transferred.
- `num_bytes`: Number of bytes being written.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_reg_read_byte**(**struct** device **dev*, u16_t *dev_addr*, u8_t *reg_addr*, u8_t **value*)

Read internal register of an I2C device.

This routine reads the value of an 8-bit internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for reading.
- `reg_addr`: Address of the internal register being read.
- `value`: Memory pool that stores the retrieved register value.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_reg_write_byte(struct** device *_dev_, u16_t _dev_addr_, u8_t _reg_addr_, u8_t _value_**)**

Write internal register of an I2C device.

This routine writes a value to an 8-bit internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for writing.
- `reg_addr`: Address of the internal register being written.
- `value`: Value to be written to internal register.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_reg_update_byte(struct** device *_dev_, u8_t _dev_addr_, u8_t _reg_addr_, u8_t _mask_, u8_t _value_**)**

Update internal register of an I2C device.

This routine updates the value of a set of bits from an 8-bit internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for updating.
- `reg_addr`: Address of the internal register being updated.
- `mask`: Bitmask for updating internal register.
- `value`: Value for updating internal register.

Return Value

- `0`: If successful.

- `-EIO`: General input / output error.

static int **i2c_burst_read16(struct** device **dev*, u16_t *dev_addr*, u16_t *start_addr*, u8_t **buf*, u8_t *num_bytes***)**

Read multiple bytes from an internal 16 bit address of an I2C device.

This routine reads multiple bytes from a 16 bit internal address of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for reading.
- `start_addr`: Internal 16 bit address from which the data is being read.
- `buf`: Memory pool that stores the retrieved data.
- `num_bytes`: Number of bytes being read.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_burst_write16(struct** device **dev*, u16_t *dev_addr*, u16_t *start_addr*, u8_t **buf*, u8_t *num_bytes***)**

Write multiple bytes to a 16 bit internal address of an I2C device.

This routine writes multiple bytes to a 16 bit internal address of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for writing.
- `start_addr`: Internal 16 bit address to which the data is being written.
- `buf`: Memory pool from which the data is transferred.
- `num_bytes`: Number of bytes being written.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_reg_read16(struct** device **dev*, u16_t *dev_addr*, u16_t *reg_addr*, u8_t **value***)**

Read internal 16 bit address register of an I2C device.

This routine reads the value of an 16-bit internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for reading.
- `reg_addr`: 16 bit address of the internal register being read.
- `value`: Memory pool that stores the retrieved register value.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_reg_write16(struct** device **dev*, u16_t *dev_addr*, u16_t *reg_addr*, u8_t *value***)**

Write internal 16 bit address register of an I2C device.

This routine writes a value to an 16-bit internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for writing.
- `reg_addr`: 16 bit address of the internal register being written.
- `value`: Value to be written to internal register.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_reg_update16(struct** device **dev*, u16_t *dev_addr*, u16_t *reg_addr*, u8_t *mask*, u8_t *value***)**

Update internal 16 bit address register of an I2C device.

This routine updates the value of a set of bits from a 16-bit internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for updating.
- `reg_addr`: 16 bit address of the internal register being updated.
- `mask`: Bitmask for updating internal register.
- `value`: Value for updating internal register.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_burst_read_addr(struct** device *_dev_, u16_t _dev_addr_, u8_t *_start_addr_, **const** u8_t _addr_size_, u8_t *_buf_, u8_t _num_bytes_**)**

Read multiple bytes from an internal variable byte size address of an I2C device.

This routine reads multiple bytes from an addr_size byte internal address of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for reading.
- `start_addr`: Array to an internal register address from which the data is being read.
- `addr_size`: Size in bytes of the register address.
- `buf`: Memory pool that stores the retrieved data.
- `num_bytes`: Number of bytes being read.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_burst_write_addr(struct** device *_dev_, u16_t _dev_addr_, u8_t *_start_addr_, **const** u8_t _addr_size_, u8_t *_buf_, u8_t _num_bytes_**)**

Write multiple bytes to an internal variable bytes size address of an I2C device. This routine writes multiple bytes to an addr_size byte internal address of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for writing.
- `start_addr`: Array to an internal register address from which the data is being read.
- `addr_size`: Size in bytes of the register address.
- `buf`: Memory pool from which the data is transferred.
- `num_bytes`: Number of bytes being written.

Return Value

- `0`: If successful.

- `Negative`: errno code if failure.

static int **i2c_reg_read_addr(struct** device *_dev_, u16_t _dev_addr_, u8_t *_reg_addr_, **const** u8_t _addr_size_, u8_t *_value_**)**

Read internal variable byte size address register of an I2C device.

This routine reads the value of an addr_size byte internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for reading.
- `reg_addr`: Array to an internal register address from which the data is being read.
- `addr_size`: Size in bytes of the register address.
- `value`: Memory pool that stores the retrieved register value.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_reg_write_addr(struct** device *_dev_, u16_t _dev_addr_, u8_t *_reg_addr_, **const** u8_t _addr_size_, u8_t _value_**)**

Write internal variable byte size address register of an I2C device.

This routine writes a value to an addr_size byte internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for writing.
- `reg_addr`: Array to an internal register address from which the data is being read.
- `addr_size`: Size in bytes of the register address.
- `value`: Value to be written to internal register.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_reg_update_addr**(**struct** device *_dev_, u16_t _dev_addr_, u8_t *_reg_addr_, u8_t _addr_size_, u8_t _mask_, u8_t _value_)

Update internal variable byte size address register of an I2C device.

This routine updates the value of a set of bits from a addr_size byte internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for updating.
- `reg_addr`: Array to an internal register address from which the data is being read.
- `addr_size`: Size in bytes of the register address.
- `mask`: Bitmask for updating internal register.
- `value`: Value for updating internal register.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

struct i2c_msg

#include <i2c.h> One I2C Message.

This defines one I2C message to transact on the I2C bus.

Public Members

u8_t ***buf**

Data buffer in bytes

u32_t **len**

Length of buffer in bytes

u8_t **flags**

Flags for this message

union dev_config

Public Members

u32_t **raw**

struct *dev_config*::*___bits* **bits**

struct __bits

Public Members

u32_t **use_10_bit_addr**

u32_t **speed**

u32_t **is_master_device**

u32_t **reserved**

4.8 Input API

INPUT public API header file.

Typedefs

typedef void (***input_notify_t**)(**struct** device *dev, **struct** *input_value* *val)

Functions

static void **input_dev_enable**(**struct** device **dev*)

Enable input device.

Enable input device. Must be the called before any calls that require communication with the input device.

Parameters

- `dev`: Pointer to the device structure for the driver instance.

static void **input_dev_disable**(**struct** device **dev*)

Disable input device.

Parameters

- `dev`: Pointer to the device structure for the driver instance.

static void **input_dev_register_notify**(**struct** device **dev*, *input_notify_t notify*)

register callback into input_dev driver

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `cb`: Callback to notify input information to caller

static void **input_dev_unregister_notify**(**struct** device *dev, input_notify_t notify)

unregister callback from input_dev driver

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `cb`: Callback to notify input information to caller

struct input_value

Public Members

uint16_t **type**
input type.

uint16_t **code**
input code.

uint32_t **value**
intput value.

4.9 Nvram API

NVRAM public API header file.

Functions

int **nvram_config_init**(**struct** device *dev)

Init nvram config region.

Parameters

- `dev`: Pointer to the device structure for the driver instance.

Return Value

- `0`: On success
- `-ENODEV`: no nvram storage.

int **nvram_config_get**(**const** char *name, void *data, int max_len)

Get nvram config.

Parameters

- `name`: name of nvram config item.
- `data`: buf to save nvram config val.

- `max_len`: the max len of nvram config val.

Return Value

- `0`: On success
- `else`: Otherwise.

int **nvram_config_set**(**const** char *_name_, **const** void *_data_, int _len_)
Set nvram config.

Parameters

- `name`: name of nvram config item.
- `data`: buf storing nvram config val.
- `len`: the len of nvram config val.

Return Value

- `0`: On success
- `else`: Otherwise.

int **nvram_config_clear_all**(void)
Clear all nvram config.

Return Value

- `0`: On success
- `else`: Otherwise.

void **nvram_config_dump**(void)
Print all nvram config item.

int **nvram_config_get_factory**(**const** char *_name_, void *_data_, int _max_len_)
Get nvram config from factory region.

Parameters

- `name`: name of nvram config item.
- `data`: buf to save nvram config val.
- `max_len`: the max len of nvram config val.

Return Value

- `0`: On success
- `else`: Otherwise.

int **nvram_config_set_factory**(**const** char *_name_, **const** void *_data_, int _len_)
Set nvram config to factory region.

Parameters

- `name`: name of nvram config item.
- `data`: buf storing nvram config val.
- `len`: the len of nvram config val.

Return Value

- `0`: On success
- `else`: Otherwise.

4.10 Pwm API

PWM public API header file.

Functions

static int **pwm_pin_set_cycles(struct** device *_dev_, u32_t _pwm_, u32_t _period_, u32_t _pulse_**)**

Set the period and pulse width for a single PWM output.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `pwm`: PWM pin.
- `period`: Period (in clock cycle) set to the PWM. HW specific.
- `pulse`: Pulse width (in clock cycle) set to the PWM. HW specific.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **pwm_pin_set_usec(struct** device *_dev_, u32_t _pwm_, u32_t _period_, u32_t _pulse_**)**

Set the period and pulse width for a single PWM output.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `pwm`: PWM pin.
- `period`: Period (in micro second) set to the PWM.
- `pulse`: Pulse width (in micro second) set to the PWM.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **pwm_get_cycles_per_sec(struct** device *_dev_, u32_t _pwm_, u64_t *_cycles_**)**

Get the clock rate (cycles per second) for a single PWM output.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `pwm`: PWM pin.
- `cycles`: Pointer to the memory to store clock rate (cycles per sec). HW specific.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **pwm_pol_set(struct** device *_dev_, u32_t _pwm_, u8_t _pol_**)**

Set the polarity for a single PWM output.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `pwm`: PWM pin.
- `pol`: Polarity of PWM.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

4.11 Rtc API

RTC public API header file.

Functions

static u32_t **rtc_read(struct** device *_dev_**)**

Read rtc time.

Parameters

- `dev`: Pointer to the device structure for the driver instance.

Return Value

- `0`: Fail
- `Positive`: The value of time

static void **rtc_enable**(**struct** device *_dev_)

Enable rtc.

Parameters

- `dev`: Pointer to the device structure for the driver instance.

static void **rtc_disable**(**struct** device *_dev_)

Disable rtc.

Parameters

- `dev`: Pointer to the device structure for the driver instance.

static int **rtc_set_config**(**struct** device *_dev_, **struct** *rtc_config* *_cfg_)

Init the rtc.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `cfg`: Rtc config arguments

Return Value

- `0`: if successful.
- `Negative`: errno code if failure.

static int **rtc_set_alarm**(**struct** device *_dev_, **const** u32_t _alarm_val_)

Set a rtc alarm .

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `alarm_val`: The time point of the alarm

Return Value

- `0`: if successful.
- `Negative`: errno code if failure.

static int **rtc_get_pending_int**(**struct** device *_dev_)

Function to get pending interrupts.

The purpose of this function is to return the interrupt status register for the device. This is especially useful when waking up from low power states to check the wake up source.

Parameters

- `dev`: Pointer to the device structure for the driver instance.

Return Value

- `1`: if the rtc interrupt is pending.
- `0`: if no rtc interrupt is pending.

struct rtc_config

Public Members

u32_t **init_val**
enable/disable alarm

u8_t **alarm_enable**
initial configuration value for the 32bit RTC alarm value

u32_t **alarm_val**
Pointer to function to call when alarm value matches current RTC value

void (*__cb_fn__)(__struct__ device *dev)

4.12 Uart API

UART public API header file.

Typedefs

typedef uart_irq_callback_t
Define the application callback function signature for UART.

Parameters

- `port`: Device struct for the UART device.

typedef uart_irq_config_func_t
For configuring IRQ on each individual UART device.

Functions

static int **uart_err_check(struct** device *_dev_**)**
Check whether an error was detected.

Parameters

- `dev`: UART device structure.

Return Value

- `UART_ERROR_OVERRUN`: if an overrun error was detected.
- `UART_ERROR_PARITY`: if a parity error was detected.
- `UART_ERROR_FRAMING`: if a framing error was detected.
- `UART_ERROR_BREAK`: if a break error was detected.
- `0`: Otherwise.

static int **uart_poll_in(struct** device *_dev_, unsigned char *_p_char_**)**
Poll the device for input.

Parameters

- `dev`: UART device structure.
- `p_char`: Pointer to character.

Return Value

- `0`: If a character arrived.
- `-1`: If no character was available to read (i.e., the UART input buffer was empty).
- `-ENOTSUP`: If the operation is not supported.

static unsigned char **uart_poll_out(struct** device *_dev_, unsigned char _out_char_**)**
Output a character in polled mode.

This routine checks if the transmitter is empty. When the transmitter is empty, it writes a character to the data register.

To send a character when hardware flow control is enabled, the handshake signal CTS must be asserted.

Parameters

- `dev`: UART device structure.
- `out_char`: Character to send.

Return Value

- `char`: Sent character.

static int **uart_fifo_fill**(**struct** device *dev, **const** u8_t *tx_data, int size)

Fill FIFO with data.

This function is expected to be called from UART interrupt handler (ISR), if uart_irq_tx_ready() returns true. Result of calling this function not from an ISR is undefined (hardware-dependent). Likewise, *not* calling this function from an ISR if uart_irq_tx_ready() returns true may lead to undefined behavior, e.g. infinite interrupt loops. It' s mandatory to test return value of this function, as different hardware has different FIFO depth (oftentimes just 1).

Return Number of bytes sent.

Parameters

- `dev`: UART device structure.
- `tx_data`: Data to transmit.
- `size`: Number of bytes to send.

static int **uart_fifo_read**(**struct** device *dev, u8_t *rx_data, **const** int size)

Read data from FIFO.

This function is expected to be called from UART interrupt handler (ISR), if uart_irq_rx_ready() returns true. Result of calling this function not from an ISR is undefined (hardware-dependent). It' s unspecified whether "RX ready" condition as returned by uart_irq_rx_ready() is level- or edge- triggered. That means that once uart_irq_rx_ready() is detected, uart_fifo_read() must be called until it reads all available data in the FIFO (i.e. until it returns less data than was requested).

Return Number of bytes read.

Parameters

- `dev`: UART device structure.
- `rx_data`: Data container.
- `size`: Container size.

static void **uart_irq_tx_enable**(**struct** device *dev)

Enable TX interrupt in IER.

Return N/A

Parameters

- `dev`: UART device structure.

static void **uart_irq_tx_disable(struct** device *_dev_**)**
Disable TX interrupt in IER.

Return N/A

Parameters

- `dev`: UART device structure.

static int **uart_irq_tx_ready(struct** device *_dev_**)**
Check if UART TX buffer can accept a new char.

Check if UART TX buffer can accept at least one character for transmission (i.e. uart_fifo_fill() will succeed and return non-zero). This function must be called in a UART interrupt handler, or its result is undefined. Before calling this function in the interrupt handler, uart_irq_update() must be called once per the handler invocation.

Parameters

- `dev`: UART device structure.

Return Value

- `1`: If at least one char can be written to UART.
- `0`: Otherwise.

static void **uart_irq_rx_enable(struct** device *_dev_**)**
Enable RX interrupt in IER.

Return N/A

Parameters

- `dev`: UART device structure.

static void **uart_irq_rx_disable(struct** device *_dev_**)**
Disable RX interrupt in IER.

Return N/A

Parameters

- `dev`: UART device structure.

static int **uart_irq_tx_complete(struct** device *_dev_**)**
Check if UART TX block finished transmission.

Check if any outgoing data buffered in UART TX block was fully transmitted and TX block is idle. When this condition is true, UART device (or whole system) can be power off. Note that this function is *not* useful to check if UART TX can accept more data, use uart_irq_tx_ready() for that. This function must be called in a UART interrupt handler, or its result is undefined. Before calling this function

in the interrupt handler, uart_irq_update() must be called once per the handler invocation.

Parameters

- `dev`: UART device structure.

Return Value

- `1`: If nothing remains to be transmitted.
- `0`: Otherwise.

static int __deprecated uart_irq_tx_empty(struct device * dev)

static int **uart_irq_rx_ready(struct** device *_dev_**)**

Check if UART RX buffer has a received char.

Check if UART RX buffer has at least one pending character (i.e. uart_fifo_read() will succeed and return non-zero). This function must be called in a UART interrupt handler, or its result is undefined. Before calling this function in the interrupt handler, uart_irq_update() must be called once per the handler invocation. It' s unspecified whether condition as returned by this function is level- or edge- triggered (i.e. if this function returns true when RX FIFO is non-empty, or when a new char was received since last call to it). See description of uart_fifo_read() for implication of this.

Parameters

- `dev`: UART device structure.

Return Value

- `1`: If a received char is ready.
- `0`: Otherwise.

static void **uart_irq_err_enable(struct** device *_dev_**)**

Enable error interrupt in IER.

Return N/A

Parameters

- `dev`: UART device structure.

static void **uart_irq_err_disable(struct** device *_dev_**)**

Disable error interrupt in IER.

Parameters

- `dev`: UART device structure.

Return Value

- `1`: If an IRQ is ready.

- `0`: Otherwise.

static int **uart_irq_is_pending**(**struct** device *dev)
Check if any IRQs is pending.

Parameters

- `dev`: UART device structure.

Return Value

- `1`: If an IRQ is pending.
- `0`: Otherwise.

static int **uart_irq_update**(**struct** device *dev)
Update cached contents of IIR.

Parameters

- `dev`: UART device structure.

Return Value

- `1`: Always.

static void **uart_irq_callback_set**(**struct** device *dev, *uart_irq_callback_t* cb)
Set the IRQ callback function pointer.

This sets up the callback for IRQ. When an IRQ is triggered, the specified function will be called.

Return N/A

Parameters

- `dev`: UART device structure.
- `cb`: Pointer to the callback function.

4.13 Spi API

SPI public API header file.

Functions

static int **spi_transceive**(**struct** *spi_config* *config, **const** **struct** *spi_buf* *tx_bufs, size_t tx_count, **struct** *spi_buf* *rx_bufs, size_t rx_count)
Read/write the specified amount of data from the SPI driver.

Note: This function is synchronous.

Parameters

- `config`: Pointer to a valid *spi_config* structure instance.
- `tx_bufs`: Buffer array where data to be sent originates from, or NULL if none.
- `tx_count`: Number of element in the tx_bufs array.
- `rx_bufs`: Buffer array where data to be read will be written to, or NULL if none.
- `rx_count`: Number of element in the rx_bufs array.

Return Value

- `0`: If successful, negative errno code otherwise.

static int **spi_read(struct** *spi_config* *config, **struct** *spi_buf* *rx_bufs, size_t *rx_count*)

Read the specified amount of data from the SPI driver.

Note: This function is synchronous.

Parameters

- `config`: Pointer to a valid *spi_config* structure instance.
- `rx_bufs`: Buffer array where data to be read will be written to.
- `rx_count`: Number of element in the rx_bufs array.

Return Value

- `0`: If successful, negative errno code otherwise.

static int **spi_write(struct** *spi_config* *config, **const struct** *spi_buf* *tx_bufs, size_t *tx_count*)

Write the specified amount of data from the SPI driver.

Note: This function is synchronous.

Parameters

- `config`: Pointer to a valid *spi_config* structure instance.
- `tx_bufs`: Buffer array where data to be sent originates from.
- `tx_count`: Number of element in the tx_bufs array.

Return Value

- `0`: If successful, negative errno code otherwise.

static int **spi_release(struct** *spi_config* ***config)**

Release the SPI device locked on by the current config.

Note: This synchronous function is used to release the lock on the SPI device that was kept if, and if only, given config parameter was the last one to be used (in any of the above functions) and if it has the SPI_LOCK_ON bit set into its operation bits field. This can be used if the caller needs to keep its hand on the SPI device for consecutive transactions.

Parameters

- **config**: Pointer to a valid *spi_config* structure instance.

struct spi_cs_control

#include <spi.h> SPI Chip Select control structure.

This can be used to control a CS line via a GPIO line, instead of using the controller inner CS logic.

gpio_dev is a valid pointer to an actual GPIO device gpio_pin is a number representing the gpio PIN that will be used to act as a CS line delay is a delay in microseconds to wait before starting the transmission and before releasing the CS line

Public Members

struct device ***gpio_dev**

u32_t **gpio_pin**

u32_t **delay**

struct spi_config

#include <spi.h> SPI controller configuration structure.

Public Members

struct device ***dev**

A valid pointer to an actual SPI device

u32_t **frequency**

The bus frequency in Hertz

u16_t **operation**

u16_t **vendor**

A vendor specific bitfield

u16_t **slave**

The slave number from 0 to host controller slave limit.

struct *spi_cs_control* ***cs**
a valid pointer on a struct *spi_cs_control* is CS line is emulated through a gpio line, or NULL otherwise.

struct spi_buf
#include <spi.h> SPI buffer structure.

buf is a valid pointer on a data buffer, or NULL otherwise. len is the length of the buffer or, if buf is NULL, will be the length which as to be sent as dummy bytes (as TX buffer) or the length of bytes that should be skipped (as RX buffer).

Public Members

void ***buf**

size_t **len**

4.14 Watchdog API

WATCHDOG public API header file.

Enums

enum wdt_mode
Values:

WDT_MODE_RESET = 0
Send reset signal when Watchdog timeout.

WDT_MODE_INTERRUPT_RESET
Send interrupt signal when Watchdog timeout.

Functions

static void **wdt_enable(struct** device ***dev)**
Enable watchdog.

Parameters

- `dev`: Pointer to the device structure for the driver instance.

static void **wdt_disable(struct** device ***dev)**
Disable watchdog.

Parameters

- `dev`: Pointer to the device structure for the driver instance.

static void **wdt_get_config**(**struct** device *dev, **struct** *wdt_config* *config)
Get watchdog config.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `config`: Pointer to save watchdog config.

static int **wdt_set_config**(**struct** device *dev, **struct** *wdt_config* *config)
Set watchdog config.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `config`: Pointer to store watchdog config.

static void **wdt_reload**(**struct** device *dev)
Clear watchdog.

Parameters

- `dev`: Pointer to the device structure for the driver instance.

struct wdt_config
#include <watchdog.h> WDT configuration struct.

Public Members

u32_t **timeout**
Watchdog timeout value.

wdt_mode **mode**
Watchdog mode.

void (***interrupt_fn**)(**struct** device *dev)
Pointer to interrrupt callback function.

CHAPTER 5

Library APIs

5.1 Irc Hal API

IRC HAL public API header file.

Functions

void *__irc_device_open__(char *_dev_name_)
open irc device

Parameters

- `irc`: device name which will be open

void __irc_device_enable_rx__(void *_handle_, _input_notify_t_ _cb_)
enable irc rx channel

Parameters

- `irc`: device handle
- `input_notify_t`: be used for notify irc rx channel keycode

void __irc_device_send_key__(void *_handle_, u32_t _key_code_, u8_t _key_state_)
send irc keycode using tx channel

Parameters

- `irc`: device handle

- **irc**: key code
- **key_state**: which is key_down or key_up

void **irc_device_disable_rx**(void *_handle_)
disable irc rx channel

Parameters

- **irc**: device handle

u8_t **irc_device_state_get**(void *_handle_)
enable irc rx channel

Return state (key down or key up)

Parameters

- **irc**: device handle

void **irc_device_close**(void *_handle_)
close irc device

Parameters

- **irc**: device handle which will be close

struct irc_handle
#include <irc_hal.h> irc handle

This structure defines a irc handle.

Public Members

struct device ***irc_dev**
dev Pointer to the device structure for the driver instance.

u8_t **state**
state of irc device.

5.2 Key Hal API

KEY HAL public API header file.

Typedefs

typedef void (***key_notify_cb**)(**struct** device *dev, **struct** *input_value* *val)

Functions

void *__key_device_open__(*key_notify_cb cb*, char **dev_name*)
open key device

Parameters

- `callback`: will be set for notifing keycode from input driver.
- `key_device`: which will be open

void __key_device_close__(void **handle*)
close key device

Parameters

- `key_device`: handle which will be close

__struct key_handle__

Public Members

__struct__ device *__input_dev__
dev Pointer to the device structure for the driver instance.

key_notify_cb __key_notify__
Pointer to notify callback function.

5.3 Led Hal API

LED HAL public API header file.

Defines

__MAX_BLINK_NUM__
max logical leds on one physical led.

__NONE_PWM_CHAN__
flag of led without pwm.

Functions

void __led_device_init__(__struct__ *led_device* **led*, __struct__ *led_config* **led_config*)
led device initialization

Parameters

- *led_device*: which will be init.
- *led_config*: for *led_device* initialization.

u8_t **get_led_status**(**struct** *led_device* *led)
get *led_device* state

Parameters

- *led_device*: which will be operated.

void **reset_led_state**(**struct** *led_device* *led)
reset *led_device* state

Parameters

- *led_device*: which will be operated.

void **set_led_status**(**struct** *led_device* *led, u8_t *led_state*, u32_t *blink_period*)
set *led_device* state

Parameters

- *led_device*: which will be operated.
- `led_state(LED_ON/LED_OFF)`: of *led_device* will be set .
- `the`: blink period of *led_device* will be set.

void **led_device_uninit**(**struct** *led_device* *led)
led device uninit

Parameters

- *led_device*: which will be uninit.

struct led_config

Public Members

u8_t **led_pin**
gpio number.

u8_t **led_pwm**
pwm number.

u8_t **led_pol**
polarity

struct led_device

Public Members

struct device ***led_dev**
dev Pointer to the device structure for the driver instance.

struct k_timer **blink_timer**
timer for set led blink period.

u8_t **blink_state**
led blink or not.

u8_t **led_pin**
gpio number of led.

u8_t **led_pwm**
pwm number of led.

u8_t **led_pol**
led polarity

u8_t **led_state**
led on or off.

u32_t **on_mask**
record led mask for multiple logical led on one physical led.

u32_t **blink_period**[**MAX_BLINK_NUM**]
record led blink period for multiple logical led on one physical led.

5.4 Audioin Hal API

AUDIOIN HAL public API header file.

Functions

int **audioin_start_record**(void **handle*)
start to recording

This routine provides start to recording , mic or linein is enable by this routine.

Return 0 start succes , others is error

Parameters

- `handle`: audio handle , return by audioin_device_open
- `parama`: create stream parama

int **audioin_stop_record**(void **handle*)
stop record

This routine provides stop recording , mic or linein is disbale by this routine.

Return 0 stop succes , others is error

Parameters

- `handle`: audio handle , return by audioin_device_open

void ***audioin_device_open**(u8_t *sample_rate*)
open audio in device

This routine provides open audio in device, and create a audio in handle .

Return NULL if open failed, otherwise return audio in hanlde

Parameters

- `sample_rate`: sample rate of audio in devices set

void **audioin_device_close**(void **handle*)
close audio in handle

This routine provides close audio in hanlde ,and release resouce for audio in.

Return N/A

Parameters

- `handle`: audio handle , return by audioin_device_open

struct audioin_handle
#include <audioin_hal.h> audio in handle

Public Members

struct device ***dev**
device of audio in driver

int **samples**
samples audio in get

int **sample_rate**
sample rate of audio in

int **rofs**
read offset for pcm buffer

int **pcm_off**
read offset for pcm buffer

bool **running**
flag is audio in running

char ***pcm_buf**
pointer of pcm buffer

int **pcm_buf_len**
pcm buffer len

struct net_buf_pool ***pcm_pool**

struct k_fifo ***pcm_queue**

5.5 Input Manager API

INPUT MANAGER public API header file.

Defines

KEY_TYPE_NULL
key press type

KEY_TYPE_LONG_DOWN

KEY_TYPE_SHORT_DOWN

KEY_TYPE_DOWN

KEY_TYPE_UP

KEY_TYPE_HOLD

KEY_TYPE_SHORT_UP

KEY_TYPE_LONG_UP

KEY_TYPE_HOLD_UP

KEY_TYPE_ALL

Typedefs

typedef void (***event_trigger**)(u32_t key_value)

typedef bool (***is_need_report_hold_key_t**)(u32_t key_value)

Enums

enum KEY_VALUE
key action type

Values:

KEY_VALUE_DOWN = 1
key press down

KEY_VALUE_UP = 0
key press release

Functions

bool **input_manager_init**(*event_trigger event_cb*, *is_need_report_hold_key_t is_need_report_hold_key*)
input manager init funcion

This routine calls init input manager ,called by main

Return true if invoked succsess.

false if invoked failed.

Parameters

- **event_cb**: when keyevent report ,this callback called before key event dispatch.

bool **input_event_lock**(void)
Lock the input event.

This routine calls lock input event, input event may not send to system if this function called.

Return true if invoked succsess.

false if invoked failed.

bool **input_event_unlock**(void)
unLock the input event

This routine calls unlock input event, input event may send to system if this function called.

Note input_event_lock and input_event_unlock Must match

Return true if invoked succsess.

false if invoked failed.

bool **input_event_islock**(void)
get the status of input event lock

Return true if input event is locked

false if input event is not locked.

5.6 Led Manager API

LED MANAGER public API header file.

Enums

enum [anonymous]
led status

Values:

LED_OFF
led status off

LED_ON
led status on

Functions

void **led_set_state**(int *led_id*, int *state*, int *blink_period*)
set target led states

Return N/A

Parameters

- `led_id`: target led id
- `state`: led status LED_ON , LED_OFF ,LED_FAST_BLINK , LED_SLOW_BLINK

bool **led_manager_init**(void)
led manager init funcion

This routine calls init app manager ,called by main

Return true if invoked succsess.

false if invoked failed.

5.7 Msg Manager API

MSG MANAGER public API header file.

Defines

MSG_CONTENT_SIZE

Typedefs

typedef void (*__MSG_CALLBAK__)(**struct** *app_msg* *, int, void *)

Functions

bool **msg_manager_init**(void)
msg_manager_init

This routine msg_manager_init

Return true init success

false init failed

bool **send_msg**(**struct** *app_msg* **msg*, int *timeout*)
Send a Asynchronous message.

This routine Send a Asynchronous message

Return true send success

false send failed

Parameters

- `receiver`: name of message receiver
- `msg`: store the received message

bool **receive_msg**(**struct** *app_msg* **msg*, int *timeout*)
receive message

This routine receive msg from others

Return NULL The listener is not in the message linsener list

tid target thread id of the message linsener

Parameters

- `msg`: store the received message
- `timeout`: time out of receive message

struct app_msg
#include <msg_manager.h> message structure

Public Members

u8_t **sender**
who send this message

u8_t **type**
: message type ,keyword of message

u8_t **cmd**
: message cmd , used to send opration type

u8_t **reserve**
: resrved byte

char **content**[**MSG_CONTENT_SIZE**]

int **value**

void ***ptr**

union app_msg::@1 app_msg::@2
: user data , to transmission some user info

MSG_CALLBAK **callback**
: message callback

List of Figures

List of Tables