API

Aio

class `mraa.`**`Aio`**(*pin*) [source]

Bases: **`object`**

API to Analog IO.

This file defines the aio interface for libmraa

C++ includes: aio.hpp

`getBit`(*Aio self*) → int [source]

self: mraa::Aio *

Gets the bit value mraa is shifting the analog read to.

bit value mraa is set return from the read function

`read`(*Aio self*) → unsigned int [source]

self: mraa::Aio *

Read a value from the AIO pin. By default mraa will shift the raw value up or down to a 10 bit value.

std::invalid_argument: in case of error

The current input voltage. By default, a 10bit value

`readFloat`(*Aio self*) → float [source]

self: mraa::Aio *

Read a value from the AIO pin and return it as a normalized float.

std::invalid_argument: in case of error

The current input voltage as a normalized float (0.0f-1.0f)

`setBit`(*Aio self*, *int bits*) → mraa::Result [source]

bits: int

Set the bit value which mraa will shift the raw reading from the ADC to. I.e. 10bits

bits: the bits the return from read should be i.e 10 mraa::Result type

I2c

class mraa.**I2c**(*bus*, *raw=False*) [source]

Bases: **object**

API to Inter-Integrated Circuit.

An I2c object represents an i2c master and can talk multiple i2c slaves by selecting the correct addressIt is considered best practice to make sure the address is correct before doing any calls on i2c, in case another application or even thread changed the addres on that bus. Multiple instances of the same bus can exist.

C++ includes: i2c.hpp

address(*I2c self*, *uint8_t address*) → mraa::Result [source]

address: uint8_t

Set the slave to talk to, typically called before every read/write operation

address: Communicate to the i2c slave on this address

Result of operation

frequency(*I2c self*, *mraa::I2cMode mode*) → mraa::Result [source]

mode: enum mraa::I2cMode

Sets the i2c Frequency for communication. Your board may not support the set frequency. Anyone can change this at any time and this will affect every slave on the bus

mode: Frequency to set the bus to

Result of operation

read(*I2c self*, *uint8_t * data*) → int [source]

data: uint8_t *

Read length bytes from the bus into *data pointer

data: Data to read into

length: Size of read in bytes to make

length of read, should match length

readByte(*I2c self*) → uint8_t [source]

self: mraa::I2c *

Read exactly one byte from the bus

std::invalid_argument: in case of error

char read from the bus

readBytesReg(*I2c self*, *uint8_t reg*, *uint8_t * data*) → int [source]

reg: uint8_t data: uint8_t *

Read length bytes from the bus into *data pointer starting from an i2c register

reg: Register to read from

data: pointer to the byte array to read data in to

length: max number of bytes to read

length passed to the function or -1

readReg(*I2c self*, *uint8_t reg*) → uint8_t [source]

reg: uint8_t

Read byte from an i2c register

reg: Register to read from

std::invalid_argument: in case of error

char read from register

readWordReg(*I2c self*, *uint8_t reg*) → uint16_t [source]

reg: uint8_t

Read word from an i2c register

reg: Register to read from

std::invalid_argument: in case of error

char read from register

write(*I2c self*, *uint8_t const * data*) → mraa::Result [source]

data: uint8_t const *

Write length bytes to the bus, the first byte in the array is the command/register to write

data: Buffer to send on the bus, first byte is i2c command

length: Size of buffer to send

Result of operation

`writeByte`(*I2c self*, *uint8_t data*) → mraa::Result [source]

data: uint8_t

Write a byte on the bus

data: The byte to send on the bus

Result of operation

`writeReg`(*I2c self*, *uint8_t reg*, *uint8_t data*) → mraa::Result [source]

reg: uint8_t data: uint8_t

Write a byte to an i2c register

reg: Register to write to

data: Value to write to register

Result of operation

`writeWordReg`(*I2c self*, *uint8_t reg*, *uint16_t data*) → mraa::Result [source]

reg: uint8_t data: uint16_t

Write a word to an i2c register

reg: Register to write to

data: Value to write to register

Result of operation

Gpio

class `mraa.`**`Gpio`**(*pin*, *owner=True*, *raw=False*) [source]

Bases: **`object`**

API to General Purpose IO.

This file defines the gpio interface for libmraa

C++ includes: gpio.hpp

`dir`(*Gpio self*, *mraa::Dir dir*) → mraa::Result [source]

dir: enum mraa::Dir

Change Gpio direction

dir: The direction to change the gpio into

Result of operation

`edge`(*Gpio self*, *mraa::Edge mode*) → mraa::Result [source]

mode: enum mraa::Edge

Set the edge mode for ISR

mode: The edge mode to set

Result of operation

`getPin`(*Gpio self*, *bool raw=False*) → int [source]

raw: bool

getPin(Gpio self) -> int

self: mraa::Gpio *

Get pin number of Gpio. If raw param is True will return the number as used within sysfs. Invalid will return -1.

raw: (optional) get the raw gpio number.

Pin number

`inputMode`(*Gpio self*, *mraa::InputMode mode*) → mraa::Result [source]

mode: enum mraa::InputMode

Change Gpio input mode

mode: The mode to change the gpio input

Result of operation

`isr`(*Gpio self*, *mraa::Edge mode*, *PyObject * pyfunc*, *PyObject * args*) → mraa::Result [source]

mode: enum mraa::Edge pyfunc: PyObject * args: PyObject *

Sets a callback to be called when pin value changes mode: The edge mode to set

fptr: Function pointer to function to be called when interrupt is triggered

args: Arguments passed to the interrupt handler (fptr)

Result of operation

isrExit(*Gpio self*) → mraa::Result [source]

self: mraa::Gpio *

Exits callback - this call will not kill the isr thread immediately but only when it is out of it’s critical section

Result of operation

mode(*Gpio self*, *mraa::Mode mode*) → mraa::Result [source]

mode: enum mraa::Mode

Change Gpio mode

mode: The mode to change the gpio into

Result of operation

outputMode(*Gpio self*, *mraa::OutputMode mode*) → mraa::Result[source]

mode: enum mraa::OutputMode

Change Gpio output driver mode

mode:

mode: Set output driver mode

Result of operation

read(*Gpio self*) → int [source]

self: mraa::Gpio *

Read value from Gpio

Gpio value

readDir(*Gpio self*) → mraa::Dir [source]

self: mraa::Gpio *

Read Gpio direction

std::runtime_error: in case of failure

Result of operation

useMmap(*Gpio self*, *bool enable*) → mraa::Result [source]

enable: bool

Enable use of mmap i/o if available.

enable: true to use mmap

Result of operation

write(*Gpio self*, *int value*) → mraa::Result [source]

value: int

Write value to Gpio

value: Value to write to Gpio

Result of operation

Pwm

class mraa.**Pwm**(*pin*, *owner=True*, *chipid=-1*) [source]

Bases: **object**

API to Pulse Width Modulation.

This file defines the PWM interface for libmraa

C++ includes: pwm.hpp

enable(*Pwm self*, *bool enable*) → mraa::Result [source]

enable: bool

Set the enable status of the PWM pin. None zero will assume on with output being driven and 0 will disable the output

enable: enable status of pin

Result of operation

max_period(*Pwm self*) → int [source]

self: mraa::Pwm *

Get the maximum PWM period in us

max PWM period in us

`min_period`(*Pwm self*) → int [source]

self: mraa::Pwm *

Get the minimum PWM period in us

min PWM period in us

`period`(*Pwm self*, *float period*) → mraa::Result [source]

period: float

Set the PWM period as seconds represented in a float

period: Period represented as a float in seconds

Result of operation

`period_ms`(*Pwm self*, *int ms*) → mraa::Result [source]

ms: int

Set period, milliseconds

ms: milliseconds for period

Result of operation

`period_us`(*Pwm self*, *int us*) → mraa::Result [source]

us: int

Set period, microseconds

us: microseconds as period

Result of operation

`pulsewidth`(*Pwm self*, *float seconds*) → mraa::Result [source]

seconds: float

Set pulsewidth, as represented by seconds in a float

seconds: The duration of a pulse

Result of operation

`pulsewidth_ms`(*Pwm self*, *int ms*) → mraa::Result [source]

ms: int

Set pulsewidth, milliseconds

ms: milliseconds for pulsewidth

Result of operation

pulsewidth_us(*Pwm self*, *int us*) → mraa::Result [source]

us: int

The pulsewidth, microseconds

us: microseconds for pulsewidth

Result of operation

read(*Pwm self*) → float [source]

self: mraa::Pwm *

Read the output duty-cycle percentage, as a float

A floating-point value representing percentage of output. The value should lie between 0.0f (representing 0%) and 1.0f Values above or below this range will be set at either 0.0f or 1.0f

write(*Pwm self*, *float percentage*) → mraa::Result [source]

percentage: float

Set the output duty-cycle percentage, as a float

percentage: A floating-point value representing percentage of output. The value should lie between 0.0f (representing 0%) and 1.0f Values above or below this range will be set at either 0.0f or 1.0f

Result of operation

Spi

class `mraa.`**Spi**(**args*) [source]

Bases: **`object`**

API to Serial Peripheral Interface.

This file defines the SPI interface for libmraa

C++ includes: spi.hpp

bitPerWord(*Spi self*, *unsigned int bits*) → mraa::Result [source]

bits: unsigned int

Set bits per mode on transaction, default is 8

bits: bits per word

Result of operation

frequency(*Spi self*, *int hz*) → mraa::Result [source]

hz: int

Set the SPI device operating clock frequency

hz: the frequency to set in hz

Result of operation

lsbmode(*Spi self*, *bool lsb*) → mraa::Result [source]

lsb: bool

Change the SPI lsb mode

lsb: Use least significant bit transmission - 0 for msbi

Result of operation

mode(*Spi self*, *mraa::Spi_Mode mode*) → mraa::Result [source]

mode: enum mraa::Spi_Mode

Set the SPI device mode. see spidev0-3

mode: the mode. See Linux spidev doc

Result of operation

write(*Spi self*, *uint8_t * txBuf*) → uint8_t * [source]

txBuf: uint8_t *

Write buffer of bytes to SPI device The pointer return has to be free'd by the caller. It will return a NULL pointer in cases of error

txBuf: buffer to send

length: size of buffer to send

uint8_t* data received on the miso line. Same length as passed in

writeByte(*Spi self*, *uint8_t data*) → int [source]

data: uint8_t

Write single byte to the SPI device

data: the byte to send

data received on the miso line or -1 in case of error

writeWord(*Spi self*, *uint16_t data*) → int [source]

data: uint16_t

Write buffer of bytes to SPI device The pointer return has to be free’d by the caller. It will return a NULL pointer in cases of error

txBuf: buffer to send

length: size of buffer (in bytes) to send

uint8_t* data received on the miso line. Same length as passed in

Uart

class mraa.**Uart**(**args*) [source]

Bases: **object**

API to UART (enabling only)

This file defines the UART interface for libmraa

C++ includes: uart.hpp

dataAvailable(*Uart self*, *unsigned int millis=0*) → bool [source]

millis: unsigned int

dataAvailable(Uart self) -> bool

self: mraa::Uart *

Check to see if data is available on the device for reading

millis: number of milliseconds to wait, or 0 to return immediately

true if there is data available to read, false otherwise

flush(*Uart self*) → mraa::Result [source]

self: mraa::Uart *

Flush the outbound data. Blocks until complete.

Result of operation

getDevicePath(*Uart self*) → std::string [source] self: mraa::Uart *

Get string with tty device path within Linux For example. Could point to “/dev/ttyS0”

char pointer of device path

read(*Uart self*, *char * data*) → int [source]

data: char *

Read bytes from the device into char* buffer

data: buffer pointer

length: maximum size of buffer

numbers of bytes read

readStr(*Uart self*, *int length*) → std::string [source]

length: int

Read bytes from the device into a String object

length: to read

std::bad_alloc: If there is no space left for read.

string of data

sendBreak(*Uart self*, *int duration*) → mraa::Result [source]

duration: int

Send a break to the device. Blocks until complete.

duration: When 0, send a break lasting at least 250 milliseconds, and not more than 500 milliseconds. When non zero, the break duration is implementation specific.

Result of operation

setBaudRate(*Uart self*, *unsigned int baud*) → mraa::Result [source]

baud: unsigned int

Set the baudrate. Takes an int and will attempt to decide what baudrate is to be used on the UART hardware.

baud: unsigned int of baudrate i.e. 9600

Result of operation

setFlowcontrol(*Uart self*, *bool xonxoff*, *bool rtscts*) → mraa::Result [source]

xonxoff: bool rtscts: bool

Set the flowcontrol

xonxoff: XON/XOFF Software flow control.

rtscts: RTS/CTS out of band hardware flow control

Result of operation

setMode(*Uart self*, *int bytesize*, *mraa::UartParity parity*, *int stopbits*) → mraa::Result [source]

bytesize: int parity: enum mraa::UartParity stopbits: int

Set the transfer mode For example setting the mode to 8N1 would be “dev.setMode(8,UART_PARITY_NONE , 1)”

bytesize: data bits

parity: Parity bit setting

stopbits: stop bits

Result of operation

setNonBlocking(*Uart self*, *bool nonblock*) → mraa::Result [source]

nonblock: bool

Set the blocking state for write operations

nonblock: new nonblocking state

Result of operation

setTimeout(*Uart self*, *int read*, *int write*, *int interchar*) → mraa::Result [source]

read: int write: int interchar: int

Set the timeout for read and write operations <= 0 will disable that timeout

read: read timeout

write: write timeout

interchar: inbetween char timeout

Result of operation

write(*Uart self*, *char const * data*) → int [source]

data: char const *

Write bytes in char* buffer to a device

data: buffer pointer

length: maximum size of buffer

the number of bytes written, or -1 if an error occurred

writeStr(*Uart self*, *std::string data*) → int [source]

data: std::string

Write bytes in String object to a device

data: string to write

the number of bytes written, or -1 if an error occurred

Common

Python interface to libmraa

class mraa.**Led**(*led*) [source]

Proxy of C++ mraa::Led class.

clearTrigger(*Led self*) → mraa::Result [source]

self: mraa::Led *

readBrightness(*Led self*) → int [source]

self: mraa::Led *

readMaxBrightness(*Led self*) → int [source]

self: mraa::Led *

setBrightness(*Led self*, *int value*) → mraa::Result [source]

value: int

trigger(*Led self*, *char const * trigger*) → mraa::Result [source]

trigger: char const *

mraa.**adcRawBits**() → unsigned int [source]

mraa.**adcSupportedBits**() → unsigned int [source] mraa.**addSubplatform**(*mraa::Platform subplatformtype*, *std::string dev*) → mraa::Result [source]

subplatformtype: enum mraa::Platform dev: std::string

mraa.**aioFromDesc**(*std::string desc*) → Aio [source]

desc: std::string

mraa.**getDefaultI2cBus**(*int platform_offset*) → int [source]

platform_offset: int

getDefaultI2cBus() -> int

mraa.**getGpioLookup**(*std::string pin_name*) → int [source]

pin_name: std::string

mraa.**getI2cBusCount**() → int [source]

mraa.**getI2cBusId**(*int i2c_bus*) → int [source]

i2c_bus: int

mraa.**getI2cLookup**(*std::string i2c_name*) → int [source]

i2c_name: std::string

mraa.**getPinCount**() → unsigned int [source]

mraa.**getPinName**(*int pin*) → std::string [source]

pin: int

mraa.**getPlatformName**() → std::string [source]

mraa.**getPlatformType**() → mraa::Platform [source]

mraa.**getPlatformVersion**(*int platform_offset*) → std::string [source]

platform_offset: int

getPlatformVersion() -> std::string

mraa.**getPwmLookup**(*std::string pwm_name*) → int [source]

pwm_name: std::string

mraa.**getSpiLookup**(*std::string spi_name*) → int [source]

spi_name: std::string

mraa.**getSubPlatformId**(*int pin_or_bus_index*) → int [source]

pin_or_bus_index: int

mraa.**getSubPlatformIndex**(*int pin_or_bus_id*) → int [source]

pin_or_bus_id: int

mraa.**getUartCount**() → int [source]

mraa.**getUartLookup**(*std::string uart_name*) → int [source]

uart_name: std::string

mraa.**getVersion**() → std::string [source]

mraa.**gpioFromDesc**(*std::string desc*) → Gpio [source]

desc: std::string

mraa.**hasSubPlatform**() → bool [source]

mraa.**i2cFromDesc**(*std::string desc*) → I2c [source]

desc: std::string

mraa.**init**() → mraa::Result [source]

mraa.**initJsonPlatform**(*std::string path*) → mraa::Result [source]

path: std::string

mraa.**isSubPlatformId**(*int pin_or_bus_id*) → bool [source]

pin_or_bus_id: int

mraa.**ledFromDesc**(*std::string desc*) → Led [source]

desc: std::string

mraa.**pinModeTest**(*int pin*, *mraa::Pinmodes mode*) → bool [source]

pin: int mode: enum mraa::Pinmodes

mraa.**printError**(*mraa::Result result*) [source]

result: enum mraa::Result

mraa.**pwmFromDesc**(*std::string desc*) → Pwm [source]

desc: std::string

mraa.**removeSubplatform**(*mraa::Platform subplatformtype*) → mraa::Result [source]

subplatformtype: enum mraa::Platform

mraa.**setLogLevel**(*int level*) → mraa::Result [source]

level: int

mraa.**setPriority**(*int const priority*) → int [source]

priority: int const

`mraa.`**`spiFromDesc`**(*std::string desc*) → Spi [source]

desc: std::string

`mraa.`**`uartFromDesc`**(*std::string desc*) → Uart [source]

desc: std::string