PIGPIO library Python Interface

pigpio is a Python module for the Raspberry which talks to the pigpio daemon to allow control of the general purpose input outputs (GPIO).

Features

o the pigpio Python module can run on Windows, Macs, or Linux
o controls one or more Pi's
o hardware timed PWM on any of GPIO 0-31
o hardware timed servo pulses on any of GPIO 0-31
o callbacks when any of GPIO 0-31 change state
o creating and transmitting precisely timed waveforms
o reading/writing GPIO and setting their modes
o wrappers for I2C, SPI, and serial links
o creating and running scripts on the pigpio daemon

GPIO

ALL GPIO are identified by their Broadcom number.

Notes

Transmitted waveforms are accurate to a microsecond.
Callback level changes are time-stamped and will be accurate to within a few microseconds.

Settings

A number of settings are determined when the pigpio daemon is started.
o the sample rate (1, 2, 4, 5, 8, or 10 us, default 5 us).
o the set of GPIO which may be updated (generally written to). The default set is those available on the Pi board revision.
o the available PWM frequencies (see set_PWM_frequency).

Exceptions

By default a fatal exception is raised if you pass an invalid argument to a pigpio function.

If you wish to handle the returned status yourself you should set pigpio.exceptions to False.

You may prefer to check the returned status in only a few parts of your code. In that case do the following:

Example

```
pigpio.exceptions = False

# Code where you want to test the error status.

pigpio.exceptions = True
```

Usage

This module uses the services of the C pigpio library. pigpio must be running on the Pi(s) whose GPIO are to be manipulated.

The normal way to start pigpio is as a daemon (during system start).

```
sudo pigpiod
```

Your Python program must import pigpio and create one or more instances of the pigpio.pi class. This class gives access to a specified Pi's GPIO.

Example

```
pi1 = pigpio.pi()      # pi1 accesses the local Pi's GPIO
pi2 = pigpio.pi('tom')  # pi2 accesses tom's GPIO
pi3 = pigpio.pi('dick') # pi3 accesses dick's GPIO

pi1.write(4, 0) # set local Pi's GPIO 4 low
pi2.write(4, 1) # set tom's GPIO 4 to high
pi3.read(4)     # get level of dick's GPIO 4
```

The later example code snippets assume that pi is an instance of the pigpio.pi class.

OVERVIEW

ESSENTIAL	
pigpio.pi	Initialise Pi connection
stop	Stop a Pi connection
BASIC	
set_mode	Set a GPIO mode
get_mode	Get a GPIO mode
set_pull_up_down	Set/clear GPIO pull up/down resistor
read	Read a GPIO
write	Write a GPIO
PWM (overrides servo commands on same GPIO)	
set_PWM_dutycycle	Start/stop PWM pulses on a GPIO
set_PWM_frequency	Set PWM frequency of a GPIO
set_PWM_range	Configure PWM range of a GPIO
get_PWM_dutycycle	Get PWM dutycycle set on a GPIO
get_PWM_frequency	Get PWM frequency of a GPIO
get_PWM_range	Get configured PWM range of a GPIO
get_PWM_real_range	Get underlying PWM range for a GPIO

Servo (overrides PWM commands on same GPIO)	
set_servo_pulsewidth	Start/Stop servo pulses on a GPIO
get_servo_pulsewidth	Get servo pulsewidth set on a GPIO
INTERMEDIATE	
gpio_trigger	Send a trigger pulse to a GPIO
set_watchdog	Set a watchdog on a GPIO
read_bank_1	Read all bank 1 GPIO
read_bank_2	Read all bank 2 GPIO
clear_bank_1	Clear selected GPIO in bank 1
clear_bank_2	Clear selected GPIO in bank 2
set_bank_1	Set selected GPIO in bank 1
set_bank_2	Set selected GPIO in bank 2
callback	Create GPIO level change callback
wait_for_edge	Wait for GPIO level change
ADVANCED	
notify_open	Request a notification handle
notify_begin	Start notifications for selected GPIO
notify_pause	Pause notifications
notify_close	Close a notification
hardware_clock	Start hardware clock on supported GPIO
hardware_PWM	Start hardware PWM on supported GPIO
set_glitch_filter	Set a glitch filter on a GPIO

set_noise_filter	Set a noise filter on a GPIO
set_pad_strength	Sets a pads drive strength
get_pad_strength	Gets a pads drive strength
shell	Executes a shell command
Custom	
custom_1	User custom function 1
custom_2	User custom function 2
Events	
event_callback	Sets a callback for an event
event_trigger	Triggers an event
wait_for_event	Wait for an event
Scripts	
store_script	Store a script
run_script	Run a stored script
update_script	Set a scripts parameters
script_status	Get script status and parameters
stop_script	Stop a running script
delete_script	Delete a stored script
I2C	
i2c_open	Opens an I2C device
i2c_close	Closes an I2C device
i2c_write_quick	SMBus write quick
i2c_read_byte	SMBus read byte
i2c_write_byte	SMBus write byte

i2c_read_byte_data	SMBus read byte data
i2c_write_byte_data	SMBus write byte data
i2c_read_word_data	SMBus read word data
i2c_write_word_data	SMBus write word data
i2c_read_block_data	SMBus read block data
i2c_write_block_data	SMBus write block data
i2c_read_i2c_block_data	SMBus read I2C block data
i2c_write_i2c_block_data	SMBus write I2C block data
i2c_read_device	Reads the raw I2C device
i2c_write_device	Writes the raw I2C device
i2c_process_call	SMBus process call
i2c_block_process_call	SMBus block process call
i2c_zip	Performs multiple I2C transactions
I2C BIT BANG	
bb_i2c_open	Opens GPIO for bit banging I2C
bb_i2c_close	Closes GPIO for bit banging I2C
bb_i2c_zip	Performs multiple bit banged I2C transactions
I2C/SPI SLAVE	
bsc_xfer	I2C/SPI as slave transfer
bsc_i2c	I2C as slave transfer
SERIAL	
serial_open	Opens a serial device
serial_close	Closes a serial device
serial_read_byte	Reads a byte from a serial device
serial_write_byte	Writes a byte to a serial device

serial_read	Reads bytes from a serial device
serial_write	Writes bytes to a serial device
serial_data_available	Returns number of bytes ready to be read
SERIAL BIT BANG (read only)	
bb_serial_read_open	Open a GPIO for bit bang serial reads
bb_serial_read_close	Close a GPIO for bit bang serial reads
bb_serial_invert	Invert serial logic (1 invert, 0 normal)
bb_serial_read	Read bit bang serial data from a GPIO
SPI	
spi_open	Opens a SPI device
spi_close	Closes a SPI device
spi_read	Reads bytes from a SPI device
spi_write	Writes bytes to a SPI device
spi_xfer	Transfers bytes with a SPI device
SPI BIT BANG	
bb_spi_open	Opens GPIO for bit banging SPI
bb_spi_close	Closes GPIO for bit banging SPI
bb_spi_xfer	Transfers bytes with bit banging SPI
FILES	
file_open	Opens a file
file_close	Closes a file
file_read	Reads bytes from a file
file_write	Writes bytes to a file
file_seek	Seeks to a position within a file

file_list	List files which match a pattern
WAVES	
wave_clear	Deletes all waveforms
wave_add_new	Starts a new waveform
wave_add_generic	Adds a series of pulses to the waveform
wave_add_serial	Adds serial data to the waveform
wave_create	Creates a waveform from added data
wave_create_and_pad	Creates a waveform of fixed size from added data
wave_delete	Deletes a waveform
wave_send_once	Transmits a waveform once
wave_send_repeat	Transmits a waveform repeatedly
wave_send_using_mode	Transmits a waveform in the chosen mode
wave_chain	Transmits a chain of waveforms
wave_tx_at	Returns the current transmitting waveform
wave_tx_busy	Checks to see if a waveform has ended
wave_tx_stop	Aborts the current waveform
wave_get_cbs	Length in cbs of the current waveform
wave_get_max_cbs	Absolute maximum allowed cbs
wave_get_micros	Length in microseconds of the current waveform
wave_get_max_micros	Absolute maximum allowed micros
wave_get_pulses	Length in pulses of the current waveform
wave_get_max_pulses	Absolute maximum allowed pulses
UTILITIES	

get_current_tick	Get current tick (microseconds)
get_hardware_revision	Get hardware revision
get_pigpio_version	Get the pigpio version
pigpio.error_text	Gets error text from error number
pigpio.tickDiff	Returns difference between two ticks

class pi(builtins.object)
pi(host, port, show_errors)
ods defined here:
pigpio.pi(builtins.object)(host, port, show_errors)
Grants access to a Pi's GPIO.

Parameters

host:= the host name of the Pi on which the pigpio daemon is
running. The default is localhost unless overridden by
the PIGPIO_ADDR environment variable.

Parameters

port:= the port number on which the pigpio daemon is listening.
The default is 8888 unless overridden by the PIGPIO_PORT
environment variable. The pigpio daemon must have been
started with the same port number.

This connects to the pigpio daemon and reserves resources to be used for sending commands and receiving notifications.

An instance attribute connected may be used to check the success of the connection. If the connection is established successfully connected will be True, otherwise False.

Example

```
pi = pigio.pi()            # use defaults
pi = pigpio.pi('mypi')     # specify host, default port
pi = pigpio.pi('mypi', 7777) # specify host and port

pi = pigpio.pi()           # exit script if no connection
if not pi.connected:
  exit()
```

__repr__()
Return repr(self).
bb_i2c_close(SDA)

This function stops bit banging I2C on a pair of GPIO previously opened with bb_i2c_open.

Parameters

SDA:= 0-31, the SDA GPIO used in a prior call to bb_i2c_open

Returns 0 if OK, otherwise PI_BAD_USER_GPIO, or PI_NOT_I2C_GPIO.

Example

pi.bb_i2c_close(SDA)

bb_i2c_open(SDA, SCL, baud)
This function selects a pair of GPIO for bit banging I2C at a specified baud rate.

Bit banging I2C allows for certain operations which are not possible with the standard I2C driver.

o baud rates as low as 50 o repeated starts o clock stretching o I2C on any pair of spare GPIO

Parameters

SDA:= 0-31
SCL:= 0-31
baud:= 50-500000

Returns 0 if OK, otherwise PI_BAD_USER_GPIO, PI_BAD_I2C_BAUD, or PI_GPIO_IN_USE.

NOTE:

The GPIO used for SDA and SCL must have pull-ups to 3V3 connected. As a guide the hardware pull-ups on pins 3 and 5 are 1k8 in value.

Example

h = pi.bb_i2c_open(4, 5, 50000) # bit bang on GPIO 4/5 at 50kbps

bb_i2c_zip(SDA, data)
This function executes a sequence of bit banged I2C operations. The operations to be performed are specified by the contents of data which contains the concatenated command codes and associated data.

Parameters

SDA:= 0-31 (as used in a prior call to bb_i2c_open)
data:= the concatenated I2C commands, see below

The returned value is a tuple of the number of bytes read and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

Example

```
(count, data) = pi.bb_i2c_zip(
        SDA, [4, 0x53, 2, 7, 1, 0x32, 2, 6, 6, 3, 0])
```

The following command codes are supported:

Name	Cmd & Data	Meaning
End	0	No more commands
Escape	1	Next P is two bytes
Start	2	Start condition
Stop	3	Stop condition
Address	4 P	Set I2C address to P
Flags	5 lsb msb	Set I2C flags to lsb + (msb << 8)
Read	6 P	Read P bytes of data
Write	7 P ...	Write P bytes of data

The address, read, and write commands take a parameter P. Normally P is one byte (0-255). If the command is preceded by the Escape command then P is two bytes (0-65535, least significant byte first).

The address and flags default to 0. The address and flags maintain their previous value until updated.

No flags are currently defined.

Any read I2C data is concatenated in the returned bytearray.

Example

```
Set address 0x53
start, write 0x32, (re)start, read 6 bytes, stop
Set address 0x1E
start, write 0x03, (re)start, read 6 bytes, stop
Set address 0x68
start, write 0x1B, (re)start, read 8 bytes, stop
End

0x04 0x53
0x02 0x07 0x01 0x32   0x02 0x06 0x06 0x03

0x04 0x1E
0x02 0x07 0x01 0x03   0x02 0x06 0x06 0x03

0x04 0x68
0x02 0x07 0x01 0x1B   0x02 0x06 0x08 0x03
```

0x00

bb_serial_invert(user_gpio, invert)
Invert serial logic.

Parameters

user_gpio:= 0-31 (opened in a prior call to bb_serial_read_open)
invert:= 0-1 (1 invert, 0 normal)

Example

status = pi.bb_serial_invert(17, 1)

bb_serial_read(user_gpio)
Returns data from the bit bang serial cyclic buffer.

Parameters

user_gpio:= 0-31 (opened in a prior call to bb_serial_read_open)

The returned value is a tuple of the number of bytes read and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

The bytes returned for each character depend upon the number of data bits bb_bits specified in the bb_serial_read_open command.

For bb_bits 1-8 there will be one byte per character. For bb_bits 9-16 there will be two bytes per character. For bb_bits 17-32 there will be four bytes per character.

Example

(count, data) = pi.bb_serial_read(4)

bb_serial_read_close(user_gpio)
Closes a GPIO for bit bang reading of serial data.

Parameters

user_gpio:= 0-31 (opened in a prior call to bb_serial_read_open)

Example

status = pi.bb_serial_read_close(17)

bb_serial_read_open(user_gpio, baud, bb_bits)
Opens a GPIO for bit bang reading of serial data.

Parameters

user_gpio:= 0-31, the GPIO to use.
baud:= 50-250000, the baud rate.
bb_bits:= 1-32, the number of bits per word, default 8.

The serial data is held in a cyclic buffer and is read using bb_serial_read.

It is the caller's responsibility to read data from the cyclic buffer in a timely fashion.

Example

```
status = pi.bb_serial_read_open(4, 19200)
status = pi.bb_serial_read_open(17, 9600)
```

bb_spi_close(CS)
This function stops bit banging SPI on a set of GPIO opened with bb_spi_open.

Parameters

CS:= 0-31, the CS GPIO used in a prior call to bb_spi_open

Returns 0 if OK, otherwise PI_BAD_USER_GPIO, or PI_NOT_SPI_GPIO.

Example

```
pi.bb_spi_close(CS)
```

bb_spi_open(CS, MISO, MOSI, SCLK, baud, spi_flags)
This function selects a set of GPIO for bit banging SPI at a specified baud rate.

Parameters

CS := 0-31
MISO := 0-31
MOSI := 0-31
SCLK := 0-31
baud := 50-250000
spiFlags := see below

spiFlags consists of the least significant 22 bits.

```
21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 0  0  0  0  0  0  R  T  0  0  0  0  0  0  0  0  0  0  p  m  m
```

mm defines the SPI mode, defaults to 0

Mode CPOL CPHA

0	0	0
1	0	1
2	1	0
3	1	1

The following constants may be used to set the mode:

pigpio.SPI_MODE_0
pigpio.SPI_MODE_1
pigpio.SPI_MODE_2
pigpio.SPI_MODE_3

Alternatively pigpio.SPI_CPOL and/or pigpio.SPI_CPHA may be used.

p is 0 if CS is active low (default) and 1 for active high. pigpio.SPI_CS_HIGH_ACTIVE may be used to set this flag.

T is 1 if the least significant bit is transmitted on MOSI first, the default (0) shifts the most significant bit out first. pigpio.SPI_TX_LSBFIRST may be used to set this flag.

R is 1 if the least significant bit is received on MISO first, the default (0) receives the most significant bit first. pigpio.SPI_RX_LSBFIRST may be used to set this flag.

The other bits in spiFlags should be set to zero.

Returns 0 if OK, otherwise PI_BAD_USER_GPIO, PI_BAD_SPI_BAUD, or PI_GPIO_IN_USE.

If more than one device is connected to the SPI bus (defined by SCLK, MOSI, and MISO) each must have its own CS.

Example

bb_spi_open(10, MISO, MOSI, SCLK, 10000, 0); // device 1
bb_spi_open(11, MISO, MOSI, SCLK, 20000, 3); // device 2

bb_spi_xfer(CS, data)
This function executes a bit banged SPI transfer.

Parameters

CS:= 0-31 (as used in a prior call to bb_spi_open)
data:= data to be sent

The returned value is a tuple of the number of bytes read and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

Example

```
#!/usr/bin/env python
```

```
import pigpio

CE0=5
CE1=6
MISO=13
MOSI=19
SCLK=12

pi = pigpio.pi()
if not pi.connected:
  exit()

pi.bb_spi_open(CE0, MISO, MOSI, SCLK, 10000, 0) # MCP4251 DAC
pi.bb_spi_open(CE1, MISO, MOSI, SCLK, 20000, 3) # MCP3008 ADC

for i in range(256):

  count, data = pi.bb_spi_xfer(CE0, [0, i]) # Set DAC value

  if count == 2:

    count, data = pi.bb_spi_xfer(CE0, [12, 0]) # Read back DAC

    if count == 2:

      set_val = data[1]

      count, data = pi.bb_spi_xfer(CE1, [1, 128, 0]) # Read ADC

      if count == 3:

        read_val = ((data[1]&3)<<8) | data[2]

        print("{} {}".format(set_val, read_val))

pi.bb_spi_close(CE0)
pi.bb_spi_close(CE1)

pi.stop()
```

bsc_i2c(i2c_address, data)
This function allows the Pi to act as a slave I2C device.

The data bytes (if any) are written to the BSC transmit FIFO and the bytes in the BSC receive FIFO are returned.

Parameters

i2c_address:= the I2C slave address.
data:= the data bytes to transmit.

The returned value is a tuple of the status, the number of bytes read, and a bytearray containing the read bytes.

See bsc_xfer for details of the status value.

If there was an error the status will be less than zero (and will contain the error code).

Note that an i2c_address of 0 may be used to close the BSC device and reassign the used GPIO as inputs.

This example assumes GPIO 2/3 are connected to GPIO 18/19 (GPIO 10/11 on the BCM2711).

Example

```
#!/usr/bin/env python
import time
import pigpio

I2C_ADDR=0x13

def i2c(id, tick):
   global pi

   s, b, d = pi.bsc_i2c(I2C_ADDR)
   if b:
      if d[0] == ord('t'): # 116 send 'HH:MM:SS*'

         print("sent={} FR={} received={} [{}]".
           format(s>>16, s&0xfff,b,d))

         s, b, d = pi.bsc_i2c(I2C_ADDR,
           "{}*".format(time.asctime()[11:19]))

      elif d[0] == ord('d'): # 100 send 'Sun Oct 30*'

         print("sent={} FR={} received={} [{}]".
           format(s>>16, s&0xfff,b,d))

         s, b, d = pi.bsc_i2c(I2C_ADDR,
           "{}*".format(time.asctime()[:10]))

pi = pigpio.pi()

if not pi.connected:
   exit()

# Respond to BSC slave activity

e = pi.event_callback(pigpio.EVENT_BSC, i2c)

pi.bsc_i2c(I2C_ADDR) # Configure BSC as I2C slave
```

```
time.sleep(600)

e.cancel()

pi.bsc_i2c(0) # Disable BSC peripheral

pi.stop()
```

While running the above.

```
$ i2cdetect -y 1
   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- 13 -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --

$ pigs i2co 1 0x13 0
0

$ pigs i2cwd 0 116
$ pigs i2crd 0 9 -a
9 10:13:58*

$ pigs i2cwd 0 116
$ pigs i2crd 0 9 -a
9 10:14:29*

$ pigs i2cwd 0 100
$ pigs i2crd 0 11 -a
11 Sun Oct 30*

$ pigs i2cwd 0 100
$ pigs i2crd 0 11 -a
11 Sun Oct 30*

$ pigs i2cwd 0 116
$ pigs i2crd 0 9 -a
9 10:23:16*

$ pigs i2cwd 0 100
$ pigs i2crd 0 11 -a
11 Sun Oct 30*
```

bsc_xfer(bsc_control, data)

This function provides a low-level interface to the SPI/I2C Slave peripheral on the BCM chip.

This peripheral allows the Pi to act as a hardware slave device on an I2C or SPI bus.

This is not a bit bang version and as such is OS timing independent. The bus timing is handled directly by the chip.

The output process is simple. You simply append data to the FIFO buffer on the chip. This works like a queue, you add data to the queue and the master removes it.

The function sets the BSC mode, writes any data in the transmit buffer to the BSC transmit FIFO, and copies any data in the BSC receive FIFO to the receive buffer.

Parameters

bsc_control:= see below
    data:= the data bytes to place in the transmit FIFO.

The returned value is a tuple of the status (see below), the number of bytes read, and a bytearray containing the read bytes. If there was an error the status will be less than zero (and will contain the error code).

Note that the control word sets the BSC mode. The BSC will stay in that mode until a different control word is sent.

GPIO used for models other than those based on the BCM2711.

	SDA	SCL	MOSI	SCLK	MISO	CE
I2C	18	19	-	-	-	-
SPI	-	-	20	19	18	21

GPIO used for models based on the BCM2711 (e.g. the Pi4B).

	SDA	SCL	MOSI	SCLK	MISO	CE
I2C	10	11	-	-	-	-
SPI	-	-	9	11	10	8

When a zero control word is received the used GPIO will be reset to INPUT mode.

bsc_control consists of the following bits:

22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
a a a a a a a - - IT HC TF IR RE TE BK EC ES PL PH I2 SP EN

Bits 0-13 are copied unchanged to the BSC CR register. See pages 163-165 of the Broadcom

peripherals document for full details.

aaaaaaa	defines the I2C slave address (only relevant in I2C mode)
IT	invert transmit status flags
HC	enable host control
TF	enable test FIFO
IR	invert receive status flags
RE	enable receive
TE	enable transmit
BK	abort operation and clear FIFOs
EC	send control register as first I2C byte
ES	send status register as first I2C byte
PL	set SPI polarity high
PH	set SPI phase high
I2	enable I2C mode
SP	enable SPI mode
EN	enable BSC peripheral

The status has the following format:

20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
S S S S S R R R R R T T T T T RB TE RF TF RE TB

Bits 0-15 are copied unchanged from the BSC FR register. See pages 165-166 of the Broadcom peripherals document for full details.

SSSSS	number of bytes successfully copied to transmit FIFO
RRRRR	number of bytes in receive FIFO
TTTTT	number of bytes in transmit FIFO
RB	receive busy
TE	transmit FIFO empty
RF	receive FIFO full
TF	transmit FIFO full
RE	receive FIFO empty
TB	transmit busy

Example

(status, count, data) = pi.bsc_xfer(0x330305, "Hello!")

The BSC slave in SPI mode deserializes data from the MOSI pin into its receiver/FIFO when the LSB of the first byte is a 0. No data is output on the MISO pin. When the LSB of the first byte on MOSI is a 1, the transmitter/FIFO data is serialized onto the MISO pin while all other data on the MOSI pin is ignored.

The BK bit of the BSC control register is non-functional when in the SPI mode. The transmitter along with its FIFO can be dequeued by successively disabling and re-enabling the TE bit on the BSC control register while in SPI mode.

This example demonstrates a SPI master talking to the BSC as SPI slave: Requires SPI master SCLK / MOSI / MISO / CE GPIO are connected to BSC peripheral GPIO 11 / 9 / 10 / 8 respectively, on a Pi4B (BCM2711).

Example

```
#!/usr/bin/env python

import pigpio

# Choose some random GPIO for the bit-bang SPI master
CE=15
MISO=26
MOSI=13
SCLK=14

pi = pigpio.pi()
if not pi.connected:
  exit()

pi.bb_spi_open(CE, MISO, MOSI, SCLK, 10000, 0) # open SPI master
pi.bsc_xfer(0x303, []) # start BSC as SPI slave
pi.bb_spi_xfer(CE, '' + 'hello') # write 'hello' to BSC
status, count, bsc_data = pi.bsc_xfer(0x303, 'world')
print bsc_data # hello
count, spi_data = pi.bb_spi_xfer(CE, [1,0,0,0,0,0])
print spi_data # world

pi.bsc_xfer(0, [])
pi.bb_spi_close(CE)

pi.stop()
```

callback(user_gpio, edge, func)
Calls a user supplied function (a callback) whenever the specified GPIO edge is detected.

Parameters

user_gpio:= 0-31.

edge:= EITHER_EDGE, RISING_EDGE (default), or FALLING_EDGE.
func:= user supplied callback function.

The user supplied callback receives three parameters, the GPIO, the level, and the tick.

Parameter	Value	Meaning
GPIO	0-31	The GPIO which has changed state
level	0-2	0 = change to low (a falling edge) 1 = change to high (a rising edge) 2 = no level change (a watchdog timeout)
tick	32 bit	The number of microseconds since boot WARNING: this wraps around from 4294967295 to 0 roughly every 72 minutes

If a user callback is not specified a default tally callback is provided which simply counts edges. The count may be retrieved by calling the tally function. The count may be reset to zero by calling the reset_tally function.

The callback may be cancelled by calling the cancel function.

A GPIO may have multiple callbacks (although I can't think of a reason to do so).

The GPIO are sampled at a rate set when the pigpio daemon is started (default 5 us).

The number of samples per second is given in the following table.

	samples per sec
	1 1,000,000
	2 500,000
sample	4 250,000
rate	5 200,000
(us)	8 125,000
	10 100,000

GPIO level changes shorter than the sample rate may be missed.

The daemon software which generates the callbacks is triggered 1000 times per second. The callbacks will be called once per level change since the last time they were called. i.e. The callbacks will get all level changes but there will be a latency.

If you want to track the level of more than one GPIO do so by maintaining the state in the callback. Do not use read. Remember the event that triggered the callback may have happened several milliseconds before and the GPIO may have changed level many times since then.

Example

```
def cbf(gpio, level, tick):
   print(gpio, level, tick)

cb1 = pi.callback(22, pigpio.EITHER_EDGE, cbf)

cb2 = pi.callback(4, pigpio.EITHER_EDGE)

cb3 = pi.callback(17)

print(cb3.tally())

cb3.reset_tally()

cb1.cancel() # To cancel callback cb1.
```

clear_bank_1(bits)
Clears GPIO 0-31 if the corresponding bit in bits is set.

Parameters

bits:= a 32 bit mask with 1 set if the corresponding GPIO is
to be cleared.

A returned status of PI_SOME_PERMITTED indicates that the user is not allowed to write to one or more of the GPIO.

Example

```
pi.clear_bank_1(int("111110010000",2))
```

clear_bank_2(bits)
Clears GPIO 32-53 if the corresponding bit (0-21) in bits is set.

Parameters

bits:= a 32 bit mask with 1 set if the corresponding GPIO is
to be cleared.

A returned status of PI_SOME_PERMITTED indicates that the user is not allowed to write to one or more of the GPIO.

Example

```
pi.clear_bank_2(0x1010)
```

custom_1(arg1, arg2, argx)
Calls a pigpio function customised by the user.

Parameters

arg1:= >=0, default 0.
arg2:= >=0, default 0.
argx:= extra arguments (each 0-255), default empty.

The returned value is an integer which by convention should be >=0 for OK and <0 for error.

Example

value = pi.custom_1()

value = pi.custom_1(23)

value = pi.custom_1(0, 55)

value = pi.custom_1(23, 56, [1, 5, 7])

value = pi.custom_1(23, 56, b"hello")

value = pi.custom_1(23, 56, "hello")

custom_2(arg1, argx, retMax)
Calls a pigpio function customised by the user.

Parameters

arg1:= >=0, default 0.
argx:= extra arguments (each 0-255), default empty.
retMax:= >=0, maximum number of bytes to return, default 8192.

The returned value is a tuple of the number of bytes returned and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

Example

(count, data) = pi.custom_2()

(count, data) = pi.custom_2(23)

(count, data) = pi.custom_2(23, [1, 5, 7])

(count, data) = pi.custom_2(23, b"hello")

(count, data) = pi.custom_2(23, "hello", 128)

delete_script(script_id)
Deletes a stored script.

Parameters

script_id:= id of stored script.

Example

```
status = pi.delete_script(sid)
```

event_callback(event, func)
Calls a user supplied function (a callback) whenever the specified event is signalled.

Parameters

event:= 0-31.
func:= user supplied callback function.

The user supplied callback receives two parameters, the event id, and the tick.

If a user callback is not specified a default tally callback is provided which simply counts events. The count may be retrieved by calling the tally function. The count may be reset to zero by calling the reset_tally function.

The callback may be canceled by calling the cancel function.

An event may have multiple callbacks (although I can't think of a reason to do so).

Example

```
def cbf(event, tick):
   print(event, tick)

cb1 = pi.event_callback(22, cbf)

cb2 = pi.event_callback(4)

print(cb2.tally())

cb2.reset_tally()

cb1.cancel() # To cancel callback cb1.
```

event_trigger(event)
This function signals the occurrence of an event.

Parameters

event:= 0-31, the event

Returns 0 if OK, otherwise PI_BAD_EVENT_ID.

An event is a signal used to inform one or more consumers to start an action. Each consumer which has registered an interest in the event (e.g. by calling event_callback) will be informed by a callback.

One event, EVENT_BSC (31) is predefined. This event is auto generated on BSC slave activity.

The meaning of other events is arbitrary.

Note that other than its id and its tick there is no data associated with an event.

Example

pi.event_trigger(23)

file_close(handle)
Closes the file associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to file_open).

Example

pi.file_close(handle)

file_list(fpattern)
Returns a list of files which match a pattern.

Parameters

fpattern:= file pattern to match.

Returns the number of returned bytes if OK, otherwise PI_NO_FILE_ACCESS, or PI_NO_FILE_MATCH.

The pattern must match an entry in /opt/pigpio/access. The pattern may contain wildcards.
See file_open.

NOTE

The returned value is not the number of files, it is the number of bytes in the buffer. The file names are separated by newline characters.

Example

```
#!/usr/bin/env python

import pigpio

pi = pigpio.pi()

if not pi.connected:
```

```
  exit()

# Assumes /opt/pigpio/access contains the following line:
# /ram/*.c r

c, d = pi.file_list("/ram/p*.c")
if c > 0:
  print(d)

pi.stop()
```

file_open(file_name, file_mode)
This function returns a handle to a file opened in a specified mode.

Parameters

file_name:= the file to open.
file_mode:= the file open mode.

Returns a handle (>=0) if OK, otherwise PI_NO_HANDLE, PI_NO_FILE_ACCESS, PI_BAD_FILE_MODE, PI_FILE_OPEN_FAILED, or PI_FILE_IS_A_DIR.

Example

```
h = pi.file_open("/home/pi/shared/dir_3/file.txt",
      pigpio.FILE_WRITE | pigpio.FILE_CREATE)

pi.file_write(h, "Hello world")

pi.file_close(h)
```

File

A file may only be opened if permission is granted by an entry in /opt/pigpio/access. This is intended to allow remote access to files in a more or less controlled manner.

Each entry in /opt/pigpio/access takes the form of a file path which may contain wildcards followed by a single letter permission. The permission may be R for read, W for write, U for read/write, and N for no access.

Where more than one entry matches a file the most specific rule applies. If no entry matches a file then access is denied.

Suppose /opt/pigpio/access contains the following entries:

```
/home/* n
/home/pi/shared/dir_1/* w
/home/pi/shared/dir_2/* r
/home/pi/shared/dir_3/* u
/home/pi/shared/dir_1/file.txt n
```

Files may be written in directory dir_1 with the exception of file.txt.

Files may be read in directory dir_2.

Files may be read and written in directory dir_3.

If a directory allows read, write, or read/write access then files may be created in that directory.

In an attempt to prevent risky permissions the following paths are ignored in /opt/pigpio/access:

a path containing ..
a path containing only wildcards (*?)
a path containing less than two non-wildcard parts

Mode

The mode may have the following values:

Constant	Value	Meaning
FILE_READ	1	open file for reading
FILE_WRITE	2	open file for writing
FILE_RW	3	open file for reading and writing

The following values may be or'd into the mode:

Name	Value	Meaning
FILE_APPEND	4	All writes append data to the end of the file
FILE_CREATE	8	The file is created if it doesn't exist
FILE_TRUNC	16	The file is truncated

Newly created files are owned by root with permissions owner read and write.

Example

```
#!/usr/bin/env python

import pigpio

pi = pigpio.pi()

if not pi.connected:
   exit()
```

```
# Assumes /opt/pigpio/access contains the following line:
# /ram/*.c r

handle = pi.file_open("/ram/pigpio.c", pigpio.FILE_READ)

done = False

while not done:
   c, d = pi.file_read(handle, 60000)
   if c > 0:
      print(d)
   else:
      done = True

pi.file_close(handle)

pi.stop()
```

file_read(handle, count)
Reads up to count bytes from the file associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to file_open).
count:= >0, the number of bytes to read.

The returned value is a tuple of the number of bytes read and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

Example

```
(b, d) = pi.file_read(h2, 100)
if b > 0:
   # process read data
```

file_seek(handle, seek_offset, seek_from)
Seeks to a position relative to the start, current position, or end of the file. Returns the new position.

Parameters

handle:= >=0 (as returned by a prior call to file_open).
seek_offset:= byte offset.
seek_from:= FROM_START, FROM_CURRENT, or FROM_END.

Example

```
new_pos = pi.file_seek(h, 100, pigpio.FROM_START)

cur_pos = pi.file_seek(h, 0, pigpio.FROM_CURRENT)
```

file_size = pi.file_seek(h, 0, pigpio.FROM_END)

file_write(handle, data)
Writes the data bytes to the file associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to file_open).
data:= the bytes to write.

Example

pi.file_write(h1, b'\x02\x03\x04')

pi.file_write(h2, b'help')

pi.file_write(h2, "hello")

pi.file_write(h1, [2, 3, 4])

get_PWM_dutycycle(user_gpio)
Returns the PWM dutycycle being used on the GPIO.

Parameters

user_gpio:= 0-31.

Returns the PWM dutycycle.

For normal PWM the dutycycle will be out of the defined range for the GPIO (see get_PWM_range).

If a hardware clock is active on the GPIO the reported dutycycle will be 500000 (500k) out of 1000000 (1M).

If hardware PWM is active on the GPIO the reported dutycycle will be out of a 1000000 (1M).

Example

```
pi.set_PWM_dutycycle(4, 25)
print(pi.get_PWM_dutycycle(4))
25

pi.set_PWM_dutycycle(4, 203)
print(pi.get_PWM_dutycycle(4))
203
```

get_PWM_frequency(user_gpio)
Returns the frequency of PWM being used on the GPIO.

Parameters

user_gpio:= 0-31.

Returns the frequency (in Hz) used for the GPIO.

For normal PWM the frequency will be that defined for the GPIO by set_PWM_frequency.

If a hardware clock is active on the GPIO the reported frequency will be that set by hardware_clock.

If hardware PWM is active on the GPIO the reported frequency will be that set by hardware_PWM.

Example

```
pi.set_PWM_frequency(4,0)
print(pi.get_PWM_frequency(4))
10

pi.set_PWM_frequency(4, 800)
print(pi.get_PWM_frequency(4))
800
```

get_PWM_range(user_gpio)
Returns the range of PWM values being used on the GPIO.

Parameters

user_gpio:= 0-31.

If a hardware clock or hardware PWM is active on the GPIO the reported range will be 1000000 (1M).

Example

```
pi.set_PWM_range(9, 500)
print(pi.get_PWM_range(9))
500
```

get_PWM_real_range(user_gpio)
Returns the real (underlying) range of PWM values being used on the GPIO.

Parameters

user_gpio:= 0-31.

If a hardware clock is active on the GPIO the reported real range will be 1000000 (1M).

If hardware PWM is active on the GPIO the reported real range will be approximately 250M divided by the set PWM frequency.

Example

```
pi.set_PWM_frequency(4, 800)
print(pi.get_PWM_real_range(4))
250
```

get_current_tick()
Returns the current system tick.

Tick is the number of microseconds since system boot. As an unsigned 32 bit quantity tick wraps around approximately every 71.6 minutes.

Example

```
t1 = pi.get_current_tick()
time.sleep(1)
t2 = pi.get_current_tick()
```

get_hardware_revision()
Returns the Pi's hardware revision number.

The hardware revision is the last few characters on the Revision line of /proc/cpuinfo.

The revision number can be used to determine the assignment of GPIO to pins (see gpio).

There are at least three types of board.

Type 1 boards have hardware revision numbers of 2 and 3.

Type 2 boards have hardware revision numbers of 4, 5, 6, and 15.

Type 3 boards have hardware revision numbers of 16 or greater.

If the hardware revision can not be found or is not a valid hexadecimal number the function returns 0.

Example

```
print(pi.get_hardware_revision())
2
```

get_mode(gpio)
Returns the GPIO mode.

Parameters

gpio:= 0-53.

Returns a value as follows

0 = INPUT
1 = OUTPUT

2 = ALT5
3 = ALT4
4 = ALT0
5 = ALT1
6 = ALT2
7 = ALT3

Example

print(pi.get_mode(0))
4

get_pad_strength(pad)
This function returns the pad drive strength in mA.

Parameters

pad:= 0-2, the pad to get.

Returns the pad drive strength if OK, otherwise PI_BAD_PAD.

Pad	GPIO
0	0-27
1	28-45
2	46-53

Example

strength = pi.get_pad_strength(0) # Get pad 0 strength.

get_pigpio_version()
Returns the pigpio software version.

Example

v = pi.get_pigpio_version()

get_servo_pulsewidth(user_gpio)
Returns the servo pulsewidth being used on the GPIO.

Parameters

user_gpio:= 0-31.

Returns the servo pulsewidth.

Example

```
pi.set_servo_pulsewidth(4, 525)
print(pi.get_servo_pulsewidth(4))
525

pi.set_servo_pulsewidth(4, 2130)
print(pi.get_servo_pulsewidth(4))
2130
```

gpio_trigger(user_gpio, pulse_len, level)
Send a trigger pulse to a GPIO. The GPIO is set to level for pulse_len microseconds and then reset to not level.

Parameters

user_gpio:= 0-31
pulse_len:= 1-100
level:= 0-1

Example

```
pi.gpio_trigger(23, 10, 1)
```

hardware_PWM(gpio, PWMfreq, PWMduty)
Starts hardware PWM on a GPIO at the specified frequency and dutycycle. Frequencies above 30MHz are unlikely to work.

NOTE: Any waveform started by wave_send_once, wave_send_repeat, or wave_chain will be cancelled.

This function is only valid if the pigpio main clock is PCM. The main clock defaults to PCM but may be overridden when the pigpio daemon is started (option -t).

Parameters

gpio:= see descripton
PWMfreq:= 0 (off) or 1-125M (1-187.5M for the BCM2711).
PWMduty:= 0 (off) to 1000000 (1M)(fully on).

Returns 0 if OK, otherwise PI_NOT_PERMITTED, PI_BAD_GPIO, PI_NOT_HPWM_GPIO, PI_BAD_HPWM_DUTY, PI_BAD_HPWM_FREQ.

The same PWM channel is available on multiple GPIO. The latest frequency and dutycycle setting will be used by all GPIO which share a PWM channel.

The GPIO must be one of the following:

12 PWM channel 0 All models but A and B
13 PWM channel 1 All models but A and B

18 PWM channel 0 All models
19 PWM channel 1 All models but A and B

40 PWM channel 0 Compute module only
41 PWM channel 1 Compute module only
45 PWM channel 1 Compute module only
52 PWM channel 0 Compute module only
53 PWM channel 1 Compute module only

The actual number of steps beween off and fully on is the integral part of 250M/PWMfreq (375M/PWMfreq for the BCM2711).

The actual frequency set is 250M/steps (375M/steps for the BCM2711).

There will only be a million steps for a PWMfreq of 250 (375 for the BCM2711). Lower frequencies will have more steps and higher frequencies will have fewer steps. PWMduty is automatically scaled to take this into account.

Example

pi.hardware_PWM(18, 800, 250000) # 800Hz 25% dutycycle

pi.hardware_PWM(18, 2000, 750000) # 2000Hz 75% dutycycle

hardware_clock(gpio, clkfreq)
Starts a hardware clock on a GPIO at the specified frequency. Frequencies above 30MHz are unlikely to work.

Parameters

  gpio:= see description
clkfreq:= 0 (off) or 4689-250M (13184-375M for the BCM2711)

Returns 0 if OK, otherwise PI_NOT_PERMITTED, PI_BAD_GPIO, PI_NOT_HCLK_GPIO, PI_BAD_HCLK_FREQ,or PI_BAD_HCLK_PASS.

The same clock is available on multiple GPIO. The latest frequency setting will be used by all GPIO which share a clock.

The GPIO must be one of the following:

4   clock 0 All models
5   clock 1 All models but A and B (reserved for system use)
6   clock 2 All models but A and B
20 clock 0 All models but A and B
21 clock 1 All models but A and Rev.2 B (reserved for system use)

32 clock 0 Compute module only
34 clock 0 Compute module only
42 clock 1 Compute module only (reserved for system use)

43 clock 2 Compute module only
44 clock 1 Compute module only (reserved for system use)

Access to clock 1 is protected by a password as its use will likely crash the Pi. The password is given by or'ing 0x5A000000 with the GPIO number.

Example

pi.hardware_clock(4, 5000) # 5 KHz clock on GPIO 4

pi.hardware_clock(4, 40000000) # 40 MHz clock on GPIO 4

i2c_block_process_call(handle, reg, data)
Writes data bytes to the specified register of the device associated with handle and reads a device specified number of bytes of data in return.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
  reg:= >=0, the device register.
 data:= the bytes to write.

The SMBus 2.0 documentation states that a minimum of 1 byte may be sent and a minimum of 1 byte may be received. The total number of bytes sent/received must be 32 or less.

SMBus 2.0 5.5.8 - Block write-block read.S Addr Wr [A] reg [A] len(data) [A] data0 [A] ... datan [A]
  S Addr Rd [A] [Count] A [Data] ... A P

The returned value is a tuple of the number of bytes read and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

Example

(b, d) = pi.i2c_block_process_call(h, 10, b'\x02\x05\x00')

(b, d) = pi.i2c_block_process_call(h, 10, b'abcdr')

(b, d) = pi.i2c_block_process_call(h, 10, "abracad")

(b, d) = pi.i2c_block_process_call(h, 10, [2, 5, 16])

i2c_close(handle)
Closes the I2C device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).

Example

pi.i2c_close(h)

i2c_open(i2c_bus, i2c_address, i2c_flags)
Returns a handle (>=0) for the device at the I2C bus address.

Parameters

i2c_bus:= >=0.
i2c_address:= 0-0x7F.
i2c_flags:= 0, no flags are currently defined.

Physically buses 0 and 1 are available on the Pi. Higher numbered buses will be available if a kernel supported bus multiplexor is being used.

The GPIO used are given in the following table.

	SDA	SCL
I2C 0	0	1
I2C 1	2	3

For the SMBus commands the low level transactions are shown at the end of the function description. The following abbreviations are used:

S (1 bit) : Start bit
P (1 bit) : Stop bit
Rd/Wr (1 bit) : Read/Write bit. Rd equals 1, Wr equals 0.
A, NA (1 bit) : Accept and not accept bit.
Addr (7 bits): I2C 7 bit address.
reg (8 bits): Command byte, which often selects a register.
Data (8 bits): A data byte.
Count (8 bits): A byte defining the length of a block operation.

[..]: Data sent by the device.

Example

h = pi.i2c_open(1, 0x53) # open device at address 0x53 on bus 1

i2c_process_call(handle, reg, word_val)
Writes 16 bits of data to the specified register of the device associated with handle and reads 16 bits of data in return.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).

reg:= >=0, the device register.
word_val:= 0-65535, the value to write.

SMBus 2.0 5.5.6 - Process call.S Addr Wr [A] reg [A] word_val_Low [A] word_val_High [A]
S Addr Rd [A] [DataLow] A [DataHigh] NA P

Example

```
r = pi.i2c_process_call(h, 4, 0x1231)
r = pi.i2c_process_call(h, 6, 0)
```

i2c_read_block_data(handle, reg)
Reads a block of up to 32 bytes from the specified register of the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
reg:= >=0, the device register.

SMBus 2.0 5.5.7 - Block read.S Addr Wr [A] reg [A]
S Addr Rd [A] [Count] A [Data] A [Data] A ... A [Data] NA P

The amount of returned data is set by the device.

The returned value is a tuple of the number of bytes read and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

Example

```
(b, d) = pi.i2c_read_block_data(h, 10)
if b >= 0:
  # process data
else:
  # process read failure
```

i2c_read_byte(handle)
Reads a single byte from the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).

SMBus 2.0 5.5.3 - Receive byte.S Addr Rd [A] [Data] NA P

Example

b = pi.i2c_read_byte(2) # read a byte from device 2

i2c_read_byte_data(handle, reg)
Reads a single byte from the specified register of the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
reg:= >=0, the device register.

SMBus 2.0 5.5.5 - Read byte.S Addr Wr [A] reg [A] S Addr Rd [A] [Data] NA P

Example

```
# read byte from reg 17 of device 2
b = pi.i2c_read_byte_data(2, 17)

# read byte from reg  1 of device 0
b = pi.i2c_read_byte_data(0, 1)
```

i2c_read_device(handle, count)
Returns count bytes read from the raw device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
count:= >0, the number of bytes to read.

S Addr Rd [A] [Data] A [Data] A ... A [Data] NA P

The returned value is a tuple of the number of bytes read and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

Example

```
(count, data) = pi.i2c_read_device(h, 12)
```

i2c_read_i2c_block_data(handle, reg, count)
Reads count bytes from the specified register of the device associated with handle . The count may be 1-32.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
reg:= >=0, the device register.
count:= >0, the number of bytes to read.

S Addr Wr [A] reg [A]
  S Addr Rd [A] [Data] A [Data] A ... A [Data] NA P

The returned value is a tuple of the number of bytes read and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

Example

```
(b, d) = pi.i2c_read_i2c_block_data(h, 4, 32)
if b >= 0:
  # process data
else:
  # process read failure
```

i2c_read_word_data(handle, reg)
Reads a single 16 bit word from the specified register of the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
  reg:= >=0, the device register.

SMBus 2.0 5.5.5 - Read word.S Addr Wr [A] reg [A] S Addr Rd [A] [DataLow] A [DataHigh] NA P

Example

```
# read word from reg 2 of device 3
w = pi.i2c_read_word_data(3, 2)

# read word from reg 7 of device 2
w = pi.i2c_read_word_data(2, 7)
```

i2c_write_block_data(handle, reg, data)
Writes up to 32 bytes to the specified register of the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
  reg:= >=0, the device register.
 data:= the bytes to write.

SMBus 2.0 5.5.7 - Block write.S Addr Wr [A] reg [A] len(data) [A] data0 [A] data1 [A] ... [A]
  datan [A] P

Example

```
pi.i2c_write_block_data(4, 5, b'hello')
```

```
pi.i2c_write_block_data(4, 5, "data bytes")

pi.i2c_write_block_data(5, 0, b'\x00\x01\x22')

pi.i2c_write_block_data(6, 2, [0, 1, 0x22])
```

i2c_write_byte(handle, byte_val)
Sends a single byte to the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
byte_val:= 0-255, the value to write.

SMBus 2.0 5.5.2 - Send byte.S Addr Wr [A] byte_val [A] P

Example

```
pi.i2c_write_byte(1, 17)   # send byte   17 to device 1
pi.i2c_write_byte(2, 0x23) # send byte 0x23 to device 2
```

i2c_write_byte_data(handle, reg, byte_val)
Writes a single byte to the specified register of the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
reg:= >=0, the device register.
byte_val:= 0-255, the value to write.

SMBus 2.0 5.5.4 - Write byte.S Addr Wr [A] reg [A] byte_val [A] P

Example

```
# send byte 0xC5 to reg 2 of device 1
pi.i2c_write_byte_data(1, 2, 0xC5)

# send byte 9 to reg 4 of device 2
pi.i2c_write_byte_data(2, 4, 9)
```

i2c_write_device(handle, data)
Writes the data bytes to the raw device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
data:= the bytes to write.

S Addr Wr [A] data0 [A] data1 [A] ... [A] datan [A] P

Example

pi.i2c_write_device(h, b"\x12\x34\xA8")

pi.i2c_write_device(h, b"help")

pi.i2c_write_device(h, 'help')

pi.i2c_write_device(h, [23, 56, 231])

i2c_write_i2c_block_data(handle, reg, data)
Writes data bytes to the specified register of the device associated with handle . 1-32 bytes may be written.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
reg:= >=0, the device register.
data:= the bytes to write.

S Addr Wr [A] reg [A] data0 [A] data1 [A] ... [A] datan [NA] P

Example

pi.i2c_write_i2c_block_data(4, 5, 'hello')

pi.i2c_write_i2c_block_data(4, 5, b'hello')

pi.i2c_write_i2c_block_data(5, 0, b'\x00\x01\x22')

pi.i2c_write_i2c_block_data(6, 2, [0, 1, 0x22])

i2c_write_quick(handle, bit)
Sends a single bit to the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
bit:= 0 or 1, the value to write.

SMBus 2.0 5.5.1 - Quick command.S Addr bit [A] P

Example

pi.i2c_write_quick(0, 1) # send 1 to device 0
pi.i2c_write_quick(3, 0) # send 0 to device 3

i2c_write_word_data(handle, reg, word_val)
Writes a single 16 bit word to the specified register of the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
reg:= >=0, the device register.
word_val:= 0-65535, the value to write.

SMBus 2.0 5.5.4 - Write word.S Addr Wr [A] reg [A] word_val_Low [A] word_val_High [A] P

Example

```
# send word 0xA0C5 to reg 5 of device 4
pi.i2c_write_word_data(4, 5, 0xA0C5)

# send word 2 to reg 2 of device 5
pi.i2c_write_word_data(5, 2, 23)
```

i2c_zip(handle, data)
This function executes a sequence of I2C operations. The operations to be performed are specified by the contents of data which contains the concatenated command codes and associated data.

Parameters

handle:= >=0 (as returned by a prior call to i2c_open).
data:= the concatenated I2C commands, see below

The returned value is a tuple of the number of bytes read and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

Example

```
(count, data) = pi.i2c_zip(h, [4, 0x53, 7, 1, 0x32, 6, 6, 0])
```

The following command codes are supported:

Name	Cmd & Data	Meaning
End	0	No more commands
Escape	1	Next P is two bytes
On	2	Switch combined flag on

Off	3	Switch combined flag off
Address	4 P	Set I2C address to P
Flags	5 lsb msb	Set I2C flags to lsb + (msb << 8)
Read	6 P	Read P bytes of data
Write	7 P ...	Write P bytes of data

The address, read, and write commands take a parameter P. Normally P is one byte (0-255). If the command is preceded by the Escape command then P is two bytes (0-65535, least significant byte first).

The address defaults to that associated with the handle. The flags default to 0. The address and flags maintain their previous value until updated.

Any read I2C data is concatenated in the returned bytearray.

Example

Set address 0x53, write 0x32, read 6 bytes
Set address 0x1E, write 0x03, read 6 bytes
Set address 0x68, write 0x1B, read 8 bytes
End

0x04 0x53 0x07 0x01 0x32 0x06 0x06
0x04 0x1E 0x07 0x01 0x03 0x06 0x06
0x04 0x68 0x07 0x01 0x1B 0x06 0x08
0x00

notify_begin(handle, bits)
Starts notifications on a handle.

Parameters

handle:= >=0 (as returned by a prior call to notify_open)
bits:= a 32 bit mask indicating the GPIO to be notified.

The notification sends state changes for each GPIO whose corresponding bit in bits is set.

The following code starts notifications for GPIO 1, 4, 6, 7, and 10 (1234 = 0x04D2 = 0b0000010011010010).

Example

```
h = pi.notify_open()
if h >= 0:
  pi.notify_begin(h, 1234)
```

notify_close(handle)

Stops notifications on a handle and releases the handle for reuse.

Parameters

handle:= >=0 (as returned by a prior call to notify_open)

Example

```
h = pi.notify_open()
if h >= 0:
  pi.notify_begin(h, 1234)
  ...
  pi.notify_close(h)
  ...
```

notify_open()
Returns a notification handle (>=0).

A notification is a method for being notified of GPIO state changes via a pipe.

Pipes are only accessible from the local machine so this function serves no purpose if you are using Python from a remote machine. The in-built (socket) notifications provided by callback should be used instead.

Notifications for handle x will be available at the pipe named /dev/pigpiox (where x is the handle number).

E.g. if the function returns 15 then the notifications must be read from /dev/pigpio15.

Notifications have the following structure:

```
H seqno
H flags
I tick
I level
```

seqno: starts at 0 each time the handle is opened and then increments by one for each report.

flags: three flags are defined, PI_NTFY_FLAGS_WDOG, PI_NTFY_FLAGS_ALIVE, and PI_NTFY_FLAGS_EVENT.

If bit 5 is set (PI_NTFY_FLAGS_WDOG) then bits 0-4 of the flags indicate a GPIO which has had a watchdog timeout.

If bit 6 is set (PI_NTFY_FLAGS_ALIVE) this indicates a keep alive signal on the pipe/socket and is sent once a minute in the absence of other notification activity.

If bit 7 is set (PI_NTFY_FLAGS_EVENT) then bits 0-4 of the flags indicate an event which has been triggered.

tick: the number of microseconds since system boot. It wraps around after 1h12m.

level: indicates the level of each GPIO. If bit 1<<x is set then GPIO x is high.

Example

```
h = pi.notify_open()
if h >= 0:
  pi.notify_begin(h, 1234)
```

notify_pause(handle)
Pauses notifications on a handle.

Parameters

handle:= >=0 (as returned by a prior call to notify_open)

Notifications for the handle are suspended until notify_begin is called again.

Example

```
h = pi.notify_open()
if h >= 0:
  pi.notify_begin(h, 1234)
  ...
  pi.notify_pause(h)
  ...
  pi.notify_begin(h, 1234)
  ...
```

read(gpio)
Returns the GPIO level.

Parameters

gpio:= 0-53.

Example

```
pi.set_mode(23, pigpio.INPUT)

pi.set_pull_up_down(23, pigpio.PUD_DOWN)
print(pi.read(23))
0

pi.set_pull_up_down(23, pigpio.PUD_UP)
print(pi.read(23))
1
```

read_bank_1()

Returns the levels of the bank 1 GPIO (GPIO 0-31).

The returned 32 bit integer has a bit set if the corresponding GPIO is high. GPIO n has bit value (1<<n).

Example

print(bin(pi.read_bank_1()))
0b10010100000011100100001001111

read_bank_2()
Returns the levels of the bank 2 GPIO (GPIO 32-53).

The returned 32 bit integer has a bit set if the corresponding GPIO is high. GPIO n has bit value (1<<(n-32)).

Example

print(bin(pi.read_bank_2()))
0b111111000000000000000000

run_script(script_id, params)
Runs a stored script.

Parameters

script_id:= id of stored script.
params:= up to 10 parameters required by the script.

Example

s = pi.run_script(sid, [par1, par2])

s = pi.run_script(sid)

s = pi.run_script(sid, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

script_status(script_id)
Returns the run status of a stored script as well as the current values of parameters 0 to 9.

Parameters

script_id:= id of stored script.

The run status may be

PI_SCRIPT_INITING
PI_SCRIPT_HALTED
PI_SCRIPT_RUNNING
PI_SCRIPT_WAITING

PI_SCRIPT_FAILED

The return value is a tuple of run status and a list of the 10 parameters. On error the run status will be negative and the parameter list will be empty.

Example

(s, pars) = pi.script_status(sid)

serial_close(handle)
Closes the serial device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to serial_open).

Example

pi.serial_close(h1)

serial_data_available(handle)
Returns the number of bytes available to be read from the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to serial_open).

Example

rdy = pi.serial_data_available(h1)

if rdy > 0:
  (b, d) = pi.serial_read(h1, rdy)

serial_open(tty, baud, ser_flags)
Returns a handle for the serial tty device opened at baud bits per second. The device name must start with /dev/tty or /dev/serial.

Parameters

tty:= the serial device to open.
baud:= baud rate in bits per second, see below.
ser_flags:= 0, no flags are currently defined.

Normally you would only use the serial_* functions if you are or will be connecting to the Pi over a network. If you will always run on the local Pi use the standard serial module instead.

The baud rate must be one of 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600,

19200, 38400, 57600, 115200, or 230400.

Example

```
h1 = pi.serial_open("/dev/ttyAMA0", 300)

h2 = pi.serial_open("/dev/ttyUSB1", 19200, 0)

h3 = pi.serial_open("/dev/serial0", 9600)
```

serial_read(handle, count)
Reads up to count bytes from the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to serial_open).
count:= >0, the number of bytes to read (defaults to 1000).

The returned value is a tuple of the number of bytes read and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

If no data is ready a bytes read of zero is returned. **Example**

```
(b, d) = pi.serial_read(h2, 100)
if b > 0:
  # process read data
```

serial_read_byte(handle)
Returns a single byte from the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to serial_open).

If no data is ready a negative error code will be returned.

Example

```
b = pi.serial_read_byte(h1)
```

serial_write(handle, data)
Writes the data bytes to the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to serial_open).
data:= the bytes to write.

Example

pi.serial_write(h1, b'\x02\x03\x04')

pi.serial_write(h2, b'help')

pi.serial_write(h2, "hello")

pi.serial_write(h1, [2, 3, 4])

serial_write_byte(handle, byte_val)
Writes a single byte to the device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to serial_open).
byte_val:= 0-255, the value to write.

Example

pi.serial_write_byte(h1, 23)

pi.serial_write_byte(h1, ord('Z'))

set_PWM_dutycycle(user_gpio, dutycycle)
Starts (non-zero dutycycle) or stops (0) PWM pulses on the GPIO.

Parameters

user_gpio:= 0-31.
dutycycle:= 0-range (range defaults to 255).

The set_PWM_range function can change the default range of 255.

Example

```
pi.set_PWM_dutycycle(4,   0) # PWM off
pi.set_PWM_dutycycle(4,  64) # PWM 1/4 on
pi.set_PWM_dutycycle(4, 128) # PWM 1/2 on
pi.set_PWM_dutycycle(4, 192) # PWM 3/4 on
pi.set_PWM_dutycycle(4, 255) # PWM full on
```

set_PWM_frequency(user_gpio, frequency)
Sets the frequency (in Hz) of the PWM to be used on the GPIO.

Parameters

user_gpio:= 0-31.
frequency:= >=0 Hz

Returns the numerically closest frequency if OK, otherwise PI_BAD_USER_GPIO or PI_NOT_PERMITTED.

If PWM is currently active on the GPIO it will be switched off and then back on at the new frequency.

Each GPIO can be independently set to one of 18 different PWM frequencies.

The selectable frequencies depend upon the sample rate which may be 1, 2, 4, 5, 8, or 10 microseconds (default 5). The sample rate is set when the pigpio daemon is started.

The frequencies for each sample rate are:

```
                Hertz

     1: 40000 20000 10000 8000 5000 4000 2500 2000 1600
        1250  1000   800  500  400  250  200  100   50

     2: 20000 10000  5000 4000 2500 2000 1250 1000  800
        625   500   400  250  200  125  100   50   25

     4: 10000  5000  2500 2000 1250 1000  625  500  400
        313   250   200  125  100   63   50   25   13
sample
 rate
 (us)  5:  8000  4000  2000 1600 1000  800  500  400  320
        250   200   160  100   80   50   40   20   10

     8:  5000  2500  1250 1000  625  500  313  250  200
        156   125   100   63   50   31   25   13    6

    10:  4000  2000  1000  800  500  400  250  200  160
        125   100    80   50   40   25   20   10    5
```

Example

```
pi.set_PWM_frequency(4,0)
print(pi.get_PWM_frequency(4))
10

pi.set_PWM_frequency(4,100000)
print(pi.get_PWM_frequency(4))
8000
```

set_PWM_range(user_gpio, range_)
Sets the range of PWM values to be used on the GPIO.

Parameters

user_gpio:= 0-31.
range_:= 25-40000.

Example

```
pi.set_PWM_range(9, 100)  # now  25 1/4,   50 1/2,   75 3/4 on
pi.set_PWM_range(9, 500)  # now 125 1/4,  250 1/2,  375 3/4 on
pi.set_PWM_range(9, 3000) # now 750 1/4, 1500 1/2, 2250 3/4 on
```

set_bank_1(bits)
Sets GPIO 0-31 if the corresponding bit in bits is set.

Parameters

bits:= a 32 bit mask with 1 set if the corresponding GPIO is
to be set.

A returned status of PI_SOME_PERMITTED indicates that the user is not allowed to write to one or more of the GPIO.

Example

```
pi.set_bank_1(int("111110010000",2))
```

set_bank_2(bits)
Sets GPIO 32-53 if the corresponding bit (0-21) in bits is set.

Parameters

bits:= a 32 bit mask with 1 set if the corresponding GPIO is
to be set.

A returned status of PI_SOME_PERMITTED indicates that the user is not allowed to write to one or more of the GPIO.

Example

```
pi.set_bank_2(0x303)
```

set_glitch_filter(user_gpio, steady)
Sets a glitch filter on a GPIO.

Level changes on the GPIO are not reported unless the level has been stable for at least steady microseconds. The level is then reported. Level changes of less than steady microseconds are ignored.

Parameters

user_gpio:= 0-31
steady:= 0-300000

Returns 0 if OK, otherwise PI_BAD_USER_GPIO, or PI_BAD_FILTER.

This filter affects the GPIO samples returned to callbacks set up with callback and wait_for_edge.

It does not affect levels read by read, read_bank_1, or read_bank_2.

Each (stable) edge will be timestamped steady microseconds after it was first detected.

Example

```
pi.set_glitch_filter(23, 100)
```

set_mode(gpio, mode)
Sets the GPIO mode.

Parameters

gpio:= 0-53.
mode:= INPUT, OUTPUT, ALT0, ALT1, ALT2, ALT3, ALT4, ALT5.

Example

```
pi.set_mode( 4, pigpio.INPUT)  # GPIO  4 as input
pi.set_mode(17, pigpio.OUTPUT) # GPIO 17 as output
pi.set_mode(24, pigpio.ALT2)   # GPIO 24 as ALT2
```

set_noise_filter(user_gpio, steady, active)
Sets a noise filter on a GPIO.

Level changes on the GPIO are ignored until a level which has been stable for steady microseconds is detected. Level changes on the GPIO are then reported for active microseconds after which the process repeats.

Parameters

user_gpio:= 0-31
  steady:= 0-300000
  active:= 0-1000000

Returns 0 if OK, otherwise PI_BAD_USER_GPIO, or PI_BAD_FILTER.

This filter affects the GPIO samples returned to callbacks set up with callback and wait_for_edge.

It does not affect levels read by read, read_bank_1, or read_bank_2.

Level changes before and after the active period may be reported. Your software must be designed to cope with such reports.

Example

pi.set_noise_filter(23, 1000, 5000)

set_pad_strength(pad, pad_strength)
This function sets the pad drive strength in mA.

Parameters

pad:= 0-2, the pad to set.
pad_strength:= 1-16 mA.

Returns 0 if OK, otherwise PI_BAD_PAD, or PI_BAD_STRENGTH.

Pad	GPIO
0	0-27
1	28-45
2	46-53

Example

pi.set_pad_strength(2, 14) # Set pad 2 to 14 mA.

set_pull_up_down(gpio, pud)
Sets or clears the internal GPIO pull-up/down resistor.

Parameters

gpio:= 0-53.
pud:= PUD_UP, PUD_DOWN, PUD_OFF.

Example

pi.set_pull_up_down(17, pigpio.PUD_OFF)
pi.set_pull_up_down(23, pigpio.PUD_UP)
pi.set_pull_up_down(24, pigpio.PUD_DOWN)

set_servo_pulsewidth(user_gpio, pulsewidth)
Starts (500-2500) or stops (0) servo pulses on the GPIO.

Parameters

user_gpio:= 0-31.
pulsewidth:= 0 (off),
500 (most anti-clockwise) - 2500 (most clockwise).

The selected pulsewidth will continue to be transmitted until changed by a subsequent call to set_servo_pulsewidth.

The pulsewidths supported by servos varies and should probably be determined by experiment. A value of 1500 should always be safe and represents the mid-point of rotation.

You can DAMAGE a servo if you command it to move beyond its limits.

Example

```
pi.set_servo_pulsewidth(17, 0)    # off
pi.set_servo_pulsewidth(17, 1000) # safe anti-clockwise
pi.set_servo_pulsewidth(17, 1500) # centre
pi.set_servo_pulsewidth(17, 2000) # safe clockwise
```

set_watchdog(user_gpio, wdog_timeout)
Sets a watchdog timeout for a GPIO.

Parameters

```
  user_gpio:= 0-31.
wdog_timeout:= 0-60000.
```

The watchdog is nominally in milliseconds.

Only one watchdog may be registered per GPIO.

The watchdog may be cancelled by setting timeout to 0.

Once a watchdog has been started callbacks for the GPIO will be triggered every timeout interval after the last GPIO activity.

The callback will receive the special level TIMEOUT.

Example

```
pi.set_watchdog(23, 1000) # 1000 ms watchdog on GPIO 23
pi.set_watchdog(23, 0)    # cancel watchdog on GPIO 23
```

shell(shellscr, pstring)
This function uses the system call to execute a shell script with the given string as its parameter.

Parameters

```
shellscr:= the name of the script, only alphanumerics,
           '-' and '_' are allowed in the name
pstring := the parameter string to pass to the script
```

The exit status of the system call is returned if OK, otherwise PI_BAD_SHELL_STATUS.

shellscr must exist in /opt/pigpio/cgi and must be executable.

The returned exit status is normally 256 times that set by the shell script exit function. If the script can't be found 32512 will be returned.

The following table gives some example returned statuses:

Script exit status	Returned system call status
1	256
5	1280
10	2560
200	51200
script not found	32512

Example

```
// pass two parameters, hello and world
status = pi.shell("scr1", "hello world");

// pass three parameters, hello, string with spaces, and world
status = pi.shell("scr1", "hello 'string with spaces' world");

// pass one parameter, hello string with spaces world
status = pi.shell("scr1", "\"hello string with spaces world\"");
```

spi_close(handle)
Closes the SPI device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to spi_open).

Example

```
pi.spi_close(h)
```

spi_open(spi_channel, baud, spi_flags)
Returns a handle for the SPI device on the channel. Data will be transferred at baud bits per second. The flags may be used to modify the default behaviour of 4-wire operation, mode 0, active low chip select.

The Pi has two SPI peripherals: main and auxiliary.

The main SPI has two chip selects (channels), the auxiliary has three.

The auxiliary SPI is available on all models but the A and B.

The GPIO used are given in the following table.

	MISO	MOSI	SCLK	CE0	CE1	CE2
Main SPI	9	10	11	8	7	-
Aux SPI	19	20	21	18	17	16

Parameters

spi_channel:= 0-1 (0-2 for the auxiliary SPI).
baud:= 32K-125M (values above 30M are unlikely to work).
spi_flags:= see below.

spi_flags consists of the least significant 22 bits.

21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
b b b b b b R T n n n n W A u2 u1 u0 p2 p1 p0 m m

mm defines the SPI mode.

WARNING: modes 1 and 3 do not appear to work on the auxiliary SPI.

Mode POL PHA
0 0 0
1 0 1
2 1 0
3 1 1

px is 0 if CEx is active low (default) and 1 for active high.

ux is 0 if the CEx GPIO is reserved for SPI (default) and 1 otherwise.

A is 0 for the main SPI, 1 for the auxiliary SPI.

W is 0 if the device is not 3-wire, 1 if the device is 3-wire. Main SPI only.

nnnn defines the number of bytes (0-15) to write before switching the MOSI line to MISO to read data. This field is ignored if W is not set. Main SPI only.

T is 1 if the least significant bit is transmitted on MOSI first, the default (0) shifts the most significant bit out first. Auxiliary SPI only.

R is 1 if the least significant bit is received on MISO first, the default (0) receives the most significant bit first. Auxiliary SPI only.

bbbbbb defines the word size in bits (0-32). The default (0) sets 8 bits per word. Auxiliary SPI only.

The spi_read, spi_write, and spi_xfer functions transfer data packed into 1, 2, or 4 bytes according to the word size in bits.

For bits 1-8 there will be one byte per character. For bits 9-16 there will be two bytes per character. For bits 17-32 there will be four bytes per character.

Multi-byte transfers are made in least significant byte first order.

E.g. to transfer 32 11-bit words data should contain 64 bytes.

E.g. to transfer the 14 bit value 0x1ABC send the bytes 0xBC followed by 0x1A.

The other bits in flags should be set to zero.

Example

```
# open SPI device on channel 1 in mode 3 at 50000 bits per second

h = pi.spi_open(1, 50000, 3)
```

spi_read(handle, count)
Reads count bytes from the SPI device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to spi_open).
count:= >0, the number of bytes to read.

The returned value is a tuple of the number of bytes read and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

Example

```
(b, d) = pi.spi_read(h, 60) # read 60 bytes from device h
if b == 60:
   # process read data
else:
   # error path
```

spi_write(handle, data)
Writes the data bytes to the SPI device associated with handle.

Parameters

handle:= >=0 (as returned by a prior call to spi_open).
data:= the bytes to write.

Example

```
pi.spi_write(0, b'\x02\xc0\x80') # write 3 bytes to device 0

pi.spi_write(0, b'defgh')       # write 5 bytes to device 0
```

```
pi.spi_write(0, "def")         # write 3 bytes to device 0

pi.spi_write(1, [2, 192, 128])   # write 3 bytes to device 1
```

spi_xfer(handle, data)
Writes the data bytes to the SPI device associated with handle, returning the data bytes read from the device.

Parameters

handle:= >=0 (as returned by a prior call to spi_open).
data:= the bytes to write.

The returned value is a tuple of the number of bytes read and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

Example

```
(count, rx_data) = pi.spi_xfer(h, b'\x01\x80\x00')

(count, rx_data) = pi.spi_xfer(h, [1, 128, 0])

(count, rx_data) = pi.spi_xfer(h, b"hello")

(count, rx_data) = pi.spi_xfer(h, "hello")
```

stop()
Release pigpio resources.

Example

```
pi.stop()
```

stop_script(script_id)
Stops a running script.

Parameters

script_id:= id of stored script.

Example

```
status = pi.stop_script(sid)
```

store_script(script)
Store a script for later execution.

See http://abyz.me.uk/rpi/pigpio/pigs.html#Scripts for details.

Parameters

script:= the script text as a series of bytes.

Returns a >=0 script id if OK.

Example

```
sid = pi.store_script(
  b'tag 0 w 22 1 mils 100 w 22 0 mils 100 dcr p0 jp 0')
```

update_script(script_id, params)
Sets the parameters of a script. The script may or may not be running. The first parameters of the script are overwritten with the new values.

Parameters

script_id:= id of stored script.
  params:= up to 10 parameters required by the script.

Example

```
s = pi.update_script(sid, [par1, par2])

s = pi.update_script(sid, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

wait_for_edge(user_gpio, edge, wait_timeout)
Wait for an edge event on a GPIO.

Parameters

```
  user_gpio:= 0-31.
      edge:= EITHER_EDGE, RISING_EDGE (default), or
           FALLING_EDGE.
wait_timeout:= >=0.0 (default 60.0).
```

The function returns when the edge is detected or after the number of seconds specified by timeout has expired.

Do not use this function for precise timing purposes, the edge is only checked 20 times a second. Whenever you need to know the accurate time of GPIO events use a callback function.

The function returns True if the edge is detected, otherwise False.

Example

```
if pi.wait_for_edge(23):
  print("Rising edge detected")
else:
```

```
  print("wait for edge timed out")

if pi.wait_for_edge(23, pigpio.FALLING_EDGE, 5.0):
  print("Falling edge detected")
else:
  print("wait for falling edge timed out")
```

wait_for_event(event, wait_timeout)
Wait for an event.

Parameters

```
     event:= 0-31.
wait_timeout:= >=0.0 (default 60.0).
```

The function returns when the event is signalled or after the number of seconds specified by timeout has expired.

The function returns True if the event is detected, otherwise False.

Example

```
if pi.wait_for_event(23):
  print("event detected")
else:
  print("wait for event timed out")
```

wave_add_generic(pulses)
Adds a list of pulses to the current waveform.

Parameters

pulses:= list of pulses to add to the waveform.

Returns the new total number of pulses in the current waveform.

The pulses are interleaved in time order within the existing waveform (if any).

Merging allows the waveform to be built in parts, that is the settings for GPIO#1 can be added, and then GPIO#2 etc.

If the added waveform is intended to start after or within the existing waveform then the first pulse should consist solely of a delay.

Example

```
G1=4
G2=24

pi.set_mode(G1, pigpio.OUTPUT)
```

```
pi.set_mode(G2, pigpio.OUTPUT)

flash_500=[] # flash every 500 ms
flash_100=[] # flash every 100 ms

#                       ON     OFF  DELAY

flash_500.append(pigpio.pulse(1<<G1, 1<<G2, 500000))
flash_500.append(pigpio.pulse(1<<G2, 1<<G1, 500000))

flash_100.append(pigpio.pulse(1<<G1, 1<<G2, 100000))
flash_100.append(pigpio.pulse(1<<G2, 1<<G1, 100000))

pi.wave_clear() # clear any existing waveforms

pi.wave_add_generic(flash_500) # 500 ms flashes
f500 = pi.wave_create() # create and save id

pi.wave_add_generic(flash_100) # 100 ms flashes
f100 = pi.wave_create() # create and save id

pi.wave_send_repeat(f500)

time.sleep(4)

pi.wave_send_repeat(f100)

time.sleep(4)

pi.wave_send_repeat(f500)

time.sleep(4)

pi.wave_tx_stop() # stop waveform

pi.wave_clear() # clear all waveforms
```

wave_add_new()
Starts a new empty waveform.

You would not normally need to call this function as it is automatically called after a waveform is created with the wave_create function.

Example

```
pi.wave_add_new()
```

wave_add_serial(user_gpio, baud, data, offset, bb_bits, bb_stop)
Adds a waveform representing serial data to the existing waveform (if any). The serial data starts offset microseconds from the start of the waveform.

Parameters

user_gpio:= GPIO to transmit data. You must set the GPIO mode
        to output.
    baud:= 50-1000000 bits per second.
    data:= the bytes to write.
  offset:= number of microseconds from the start of the
        waveform, default 0.
 bb_bits:= number of data bits, default 8.
 bb_stop:= number of stop half bits, default 2.

Returns the new total number of pulses in the current waveform.

The serial data is formatted as one start bit, bb_bits data bits, and bb_stop/2 stop bits.

It is legal to add serial data streams with different baud rates to the same waveform.

The bytes required for each character depend upon bb_bits.

For bb_bits 1-8 there will be one byte per character. For bb_bits 9-16 there will be two bytes per character. For bb_bits 17-32 there will be four bytes per character.

Example

pi.wave_add_serial(4, 300, 'Hello world')

pi.wave_add_serial(4, 300, b"Hello world")

pi.wave_add_serial(4, 300, b'\x23\x01\x00\x45')

pi.wave_add_serial(17, 38400, [23, 128, 234], 5000)

wave_chain(data)
This function transmits a chain of waveforms.

NOTE: Any hardware PWM started by hardware_PWM will be cancelled.

The waves to be transmitted are specified by the contents of data which contains an ordered list of wave_ids and optional command codes and related data.

Returns 0 if OK, otherwise PI_CHAIN_NESTING, PI_CHAIN_LOOP_CNT, PI_BAD_CHAIN_LOOP, PI_BAD_CHAIN_CMD, PI_CHAIN_COUNTER, PI_BAD_CHAIN_DELAY, PI_CHAIN_TOO_BIG, or PI_BAD_WAVE_ID.

Each wave is transmitted in the order specified. A wave may occur multiple times per chain.

A blocks of waves may be transmitted multiple times by using the loop commands. The block is bracketed by loop start and end commands. Loops may be nested.

Delays between waves may be added with the delay command.

The following command codes are supported:

Name	Cmd & Data	Meaning
Loop Start	255 0	Identify start of a wave block
Loop Repeat	255 1 x y	loop x + y*256 times
Delay	255 2 x y	delay x + y*256 microseconds
Loop Forever	255 3	loop forever

If present Loop Forever must be the last entry in the chain.

The code is currently dimensioned to support a chain with roughly 600 entries and 20 loop counters.

Example

```
#!/usr/bin/env python

import time
import pigpio

WAVES=5
GPIO=4

wid=[0]*WAVES

pi = pigpio.pi() # Connect to local Pi.

pi.set_mode(GPIO, pigpio.OUTPUT);

for i in range(WAVES):
  pi.wave_add_generic([
    pigpio.pulse(1<<GPIO, 0, 20),
    pigpio.pulse(0, 1<<GPIO, (i+1)*200)]);

  wid[i] = pi.wave_create();

pi.wave_chain([
  wid[4], wid[3], wid[2],      # transmit waves 4+3+2
  255, 0,                  # loop start
    wid[0], wid[0], wid[0],    # transmit waves 0+0+0
    255, 0,                  # loop start
      wid[0], wid[1],        # transmit waves 0+1
      255, 2, 0x88, 0x13,    # delay 5000us
    255, 1, 30, 0,           # loop end (repeat 30 times)
    255, 0,                  # loop start
      wid[2], wid[3], wid[0], # transmit waves 2+3+0
      wid[3], wid[1], wid[2], # transmit waves 3+1+2
    255, 1, 10, 0,           # loop end (repeat 10 times)
  255, 1, 5, 0,              # loop end (repeat 5 times)
```

```
  wid[4], wid[4], wid[4],      # transmit waves 4+4+4
  255, 2, 0x20, 0x4E,          # delay 20000us
  wid[0], wid[0], wid[0],      # transmit waves 0+0+0
  ])

while pi.wave_tx_busy():
  time.sleep(0.1);

for i in range(WAVES):
  pi.wave_delete(wid[i])

pi.stop()
```

wave_clear()
Clears all waveforms and any data added by calls to the wave_add_* functions.

Example

```
pi.wave_clear()
```

wave_create()
Creates a waveform from the data provided by the prior calls to the wave_add_* functions.

Returns a wave id (>=0) if OK, otherwise PI_EMPTY_WAVEFORM, PI_TOO_MANY_CBS, PI_TOO_MANY_OOL, or PI_NO_WAVEFORM_ID.

The data provided by the wave_add_* functions is consumed by this function.

As many waveforms may be created as there is space available. The wave id is passed to wave_send_* to specify the waveform to transmit.

Normal usage would be

Step 1. wave_clear to clear all waveforms and added data.

Step 2. wave_add_* calls to supply the waveform data.

Step 3. wave_create to create the waveform and get a unique id

Repeat steps 2 and 3 as needed.

Step 4. wave_send_* with the id of the waveform to transmit.

A waveform comprises one or more pulses.

A pulse specifies

1) the GPIO to be switched on at the start of the pulse. 2) the GPIO to be switched off at the start of the pulse. 3) the delay in microseconds before the next pulse.

Any or all the fields can be zero. It doesn't make any sense to set all the fields to zero (the pulse will be ignored).

When a waveform is started each pulse is executed in order with the specified delay between the pulse and the next.

Example

```
wid = pi.wave_create()
```

wave_create_and_pad(percent)
This function creates a waveform like wave_create but pads the consumed resources. Where percent gives the percentage of the resources to use (in terms of the theoretical maximum, not the current amount free). This allows the reuse of deleted waves while a transmission is active.

Upon success a wave id greater than or equal to 0 is returned, otherwise PI_EMPTY_WAVEFORM, PI_TOO_MANY_CBS, PI_TOO_MANY_OOL, or PI_NO_WAVEFORM_ID.

percent: 0-100, size of waveform as percentage of maximum available.

The data provided by the wave_add_* functions are consumed by this function.

As many waveforms may be created as there is space available. The wave id is passed to wave_send_* to specify the waveform to transmit.

A usage would be the creation of two waves where one is filled while the other is being transmitted. Each wave is assigned 50% of the resources. This buffer structure allows the transmission of infinite wave sequences.

Normal usage:

Step 1. wave_clear to clear all waveforms and added data.

Step 2. wave_add_* calls to supply the waveform data.

Step 3. wave_create_and_pad to create a waveform of uniform size.

Step 4. wave_send_* with the id of the waveform to transmit.

Repeat steps 2-4 as needed.

Step 5. Any wave id can now be deleted and another wave of the same size can be created in its place.

Example

```
wid = pi.wave_create_and_pad(50)
```

wave_delete(wave_id)
This function deletes the waveform with id wave_id.

Parameters

wave_id:= >=0 (as returned by a prior call to wave_create).

Wave ids are allocated in order, 0, 1, 2, etc.

The wave is flagged for deletion. The resources used by the wave will only be reused when either of the following apply.

- all waves with higher numbered wave ids have been deleted or have been flagged for deletion.

- a new wave is created which uses exactly the same resources as the current wave (see the C source for gpioWaveCreate for details).

Example

pi.wave_delete(6) # delete waveform with id 6

pi.wave_delete(0) # delete waveform with id 0

wave_get_cbs()
Returns the length in DMA control blocks of the current waveform.

Example

cbs = pi.wave_get_cbs()

wave_get_max_cbs()
Returns the maximum possible size of a waveform in DMA control blocks.

Example

cbs = pi.wave_get_max_cbs()

wave_get_max_micros()
Returns the maximum possible size of a waveform in microseconds.

Example

micros = pi.wave_get_max_micros()

wave_get_max_pulses()
Returns the maximum possible size of a waveform in pulses.

Example

pulses = pi.wave_get_max_pulses()

wave_get_micros()
Returns the length in microseconds of the current waveform.

Example

micros = pi.wave_get_micros()

wave_get_pulses()
Returns the length in pulses of the current waveform.

Example

pulses = pi.wave_get_pulses()

wave_send_once(wave_id)
Transmits the waveform with id wave_id. The waveform is sent once.

NOTE: Any hardware PWM started by hardware_PWM will be cancelled.

Parameters

wave_id:= >=0 (as returned by a prior call to wave_create).

Returns the number of DMA control blocks used in the waveform.

Example

cbs = pi.wave_send_once(wid)

wave_send_repeat(wave_id)
Transmits the waveform with id wave_id. The waveform repeats until wave_tx_stop is called or another call to wave_send_* is made.

NOTE: Any hardware PWM started by hardware_PWM will be cancelled.

Parameters

wave_id:= >=0 (as returned by a prior call to wave_create).

Returns the number of DMA control blocks used in the waveform.

Example

cbs = pi.wave_send_repeat(wid)

wave_send_using_mode(wave_id, mode)
Transmits the waveform with id wave_id using mode mode.

Parameters

wave_id:= >=0 (as returned by a prior call to wave_create).
mode:= WAVE_MODE_ONE_SHOT, WAVE_MODE_REPEAT,
WAVE_MODE_ONE_SHOT_SYNC, or WAVE_MODE_REPEAT_SYNC.

WAVE_MODE_ONE_SHOT: same as wave_send_once.

WAVE_MODE_REPEAT same as wave_send_repeat.

WAVE_MODE_ONE_SHOT_SYNC same as wave_send_once but tries to sync with the previous waveform.

WAVE_MODE_REPEAT_SYNC same as wave_send_repeat but tries to sync with the previous waveform.

WARNING: bad things may happen if you delete the previous waveform before it has been synced to the new waveform.

NOTE: Any hardware PWM started by hardware_PWM will be cancelled.

Parameters

wave_id:= >=0 (as returned by a prior call to wave_create).

Returns the number of DMA control blocks used in the waveform.

Example

```
cbs = pi.wave_send_using_mode(wid, WAVE_MODE_REPEAT_SYNC)
```

wave_tx_at()
Returns the id of the waveform currently being transmitted using wave_send*. Chained waves are not supported.

Returns the waveform id or one of the following special values:

WAVE_NOT_FOUND (9998) - transmitted wave not found. NO_TX_WAVE (9999) - no wave being transmitted.

Example

```
wid = pi.wave_tx_at()
```

wave_tx_busy()
Returns 1 if a waveform is currently being transmitted, otherwise 0.

Example

```
pi.wave_send_once(0) # send first waveform

while pi.wave_tx_busy(): # wait for waveform to be sent
  time.sleep(0.1)

pi.wave_send_once(1) # send next waveform
```

wave_tx_repeat()

This function is deprecated and has beeen removed.

Use wave_create/wave_send_* instead.
wave_tx_start()
This function is deprecated and has been removed.

Use wave_create/wave_send_* instead.
wave_tx_stop()
Stops the transmission of the current waveform.

This function is intended to stop a waveform started with wave_send_repeat.

Example

```
pi.wave_send_repeat(3)

time.sleep(5)

pi.wave_tx_stop()
```

write(gpio, level)
Sets the GPIO level.

Parameters

GPIO:= 0-53.
level:= 0, 1.

If PWM or servo pulses are active on the GPIO they are switched off.

Example

```
pi.set_mode(17, pigpio.OUTPUT)

pi.write(17,0)
print(pi.read(17))
0

pi.write(17,1)
print(pi.read(17))
1
```

---------- descriptors defined here:

ct__ dictionary for instance variables (if defined)

akref__ list of weak references to the object (if defined)
class pulse(builtins.object)
pulse(gpio_on, gpio_off, delay)

ass to store pulse information.

ods defined here:
pigpio.pulse(builtins.object)(gpio_on, gpio_off, delay)
Initialises a pulse.

Parameters

gpio_on:= the GPIO to switch on at the start of the pulse.
gpio_off:= the GPIO to switch off at the start of the pulse.
delay:= the delay in microseconds before the next pulse.

---------- descriptors defined here:

ct__ dictionary for instance variables (if defined)

akref__ list of weak references to the object (if defined)
FUNCTIONS
pigpio.error_text(errnum)
Returns a text description of a pigpio error.

Parameters

errnum:= <0, the error number

Example

print(pigpio.error_text(-5))
level not 0-1

pigpio.tickDiff(t1, t2)
Returns the microsecond difference between two ticks.

Parameters

t1:= the earlier tick
t2:= the later tick

Example

print(pigpio.tickDiff(4294967272, 12))
36

pigpio.u2i(uint32)
Converts a 32 bit unsigned number to signed.

Parameters

uint32:= an unsigned 32 bit number

Example

```
print(u2i(4294967272))
-24
print(u2i(37))
37
```

PARAMETERS

active: 0-1000000
The number of microseconds level changes are reported for once a noise filter has been triggered (by steady microseconds of a stable level).

arg1:
An unsigned argument passed to a user customised function. Its meaning is defined by the customiser.

arg2:
An unsigned argument passed to a user customised function. Its meaning is defined by the customiser.

argx:
An array of bytes passed to a user customised function. Its meaning and content is defined by the customiser.

baud:
The speed of serial communication (I2C, SPI, serial link, waves) in bits per second.

bb_bits: 1-32
The number of data bits to be used when adding serial data to a waveform.

bb_stop: 2-8
The number of (half) stop bits to be used when adding serial data to a waveform.

bit: 0-1
A value of 0 or 1.

bits: 32 bit number
A mask used to select GPIO to be operated on. If bit n is set then GPIO n is selected. A convenient way of setting bit n is to bit or in the value (1<<n).

To select GPIO 1, 7, 23

```
bits = (1<<1) | (1<<7) | (1<<23)
```

bsc_control:

```
22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 a  a  a  a  a  a  a  -  - IT HC TF IR RE TE BK EC ES PL PH I2 SP EN
```

aaaaaaa defines the I2C slave address (only relevant in I2C mode)

Bits 0-13 are copied unchanged to the BSC CR register. See pages 163-165 of the Broadcom peripherals document.

byte_val: 0-255
A whole number.

clkfreq: 4689-250M (13184-375M for the BCM2711)
The hardware clock frequency.

connected:
True if a connection was established, False otherwise.

count:
The number of bytes of data to be transferred.
CS:
The GPIO used for the slave select signal when bit banging SPI.
data:
Data to be transmitted, a series of bytes.
delay: >=1
The length of a pulse in microseconds.
dutycycle: 0-range_
A number between 0 and range_.

The dutycycle sets the proportion of time on versus time off during each PWM cycle.

Dutycycle	On time
0	Off
range_ * 0.25	25% On
range_ * 0.50	50% On
range_ * 0.75	75% On
range_	Fully On

edge: 0-2
EITHER_EDGE = 2
FALLING_EDGE = 1
RISING_EDGE = 0

errnum: <0
PI_BAD_USER_GPIO = -2
PI_BAD_GPIO = -3
PI_BAD_MODE = -4
PI_BAD_LEVEL = -5
PI_BAD_PUD = -6
PI_BAD_PULSEWIDTH = -7
PI_BAD_DUTYCYCLE = -8
PI_BAD_WDOG_TIMEOUT = -15
PI_BAD_DUTYRANGE = -21
PI_NO_HANDLE = -24
PI_BAD_HANDLE = -25
PI_BAD_WAVE_BAUD = -35
PI_TOO_MANY_PULSES = -36
PI_TOO_MANY_CHARS = -37
PI_NOT_SERIAL_GPIO = -38
PI_NOT_PERMITTED = -41
PI_SOME_PERMITTED = -42
PI_BAD_WVSC_COMMND = -43
PI_BAD_WVSM_COMMND = -44
PI_BAD_WVSP_COMMND = -45
PI_BAD_PULSELEN = -46
PI_BAD_SCRIPT = -47
PI_BAD_SCRIPT_ID = -48
PI_BAD_SER_OFFSET = -49

PI_GPIO_IN_USE = -50
PI_BAD_SERIAL_COUNT = -51
PI_BAD_PARAM_NUM = -52
PI_DUP_TAG = -53
PI_TOO_MANY_TAGS = -54
PI_BAD_SCRIPT_CMD = -55
PI_BAD_VAR_NUM = -56
PI_NO_SCRIPT_ROOM = -57
PI_NO_MEMORY = -58
PI_SOCK_READ_FAILED = -59
PI_SOCK_WRIT_FAILED = -60
PI_TOO_MANY_PARAM = -61
PI_SCRIPT_NOT_READY = -62
PI_BAD_TAG = -63
PI_BAD_MICS_DELAY = -64
PI_BAD_MILS_DELAY = -65
PI_BAD_WAVE_ID = -66
PI_TOO_MANY_CBS = -67
PI_TOO_MANY_OOL = -68
PI_EMPTY_WAVEFORM = -69
PI_NO_WAVEFORM_ID = -70
PI_I2C_OPEN_FAILED = -71
PI_SER_OPEN_FAILED = -72
PI_SPI_OPEN_FAILED = -73
PI_BAD_I2C_BUS = -74
PI_BAD_I2C_ADDR = -75
PI_BAD_SPI_CHANNEL = -76
PI_BAD_FLAGS = -77
PI_BAD_SPI_SPEED = -78
PI_BAD_SER_DEVICE = -79
PI_BAD_SER_SPEED = -80
PI_BAD_PARAM = -81
PI_I2C_WRITE_FAILED = -82
PI_I2C_READ_FAILED = -83
PI_BAD_SPI_COUNT = -84
PI_SER_WRITE_FAILED = -85
PI_SER_READ_FAILED = -86
PI_SER_READ_NO_DATA = -87
PI_UNKNOWN_COMMAND = -88
PI_SPI_XFER_FAILED = -89
PI_NO_AUX_SPI = -91
PI_NOT_PWM_GPIO = -92
PI_NOT_SERVO_GPIO = -93
PI_NOT_HCLK_GPIO = -94
PI_NOT_HPWM_GPIO = -95
PI_BAD_HPWM_FREQ = -96
PI_BAD_HPWM_DUTY = -97
PI_BAD_HCLK_FREQ = -98
PI_BAD_HCLK_PASS = -99
PI_HPWM_ILLEGAL = -100
PI_BAD_DATABITS = -101
PI_BAD_STOPBITS = -102

PI_MSG_TOOBIG = -103
PI_BAD_MALLOC_MODE = -104
PI_BAD_SMBUS_CMD = -107
PI_NOT_I2C_GPIO = -108
PI_BAD_I2C_WLEN = -109
PI_BAD_I2C_RLEN = -110
PI_BAD_I2C_CMD = -111
PI_BAD_I2C_BAUD = -112
PI_CHAIN_LOOP_CNT = -113
PI_BAD_CHAIN_LOOP = -114
PI_CHAIN_COUNTER = -115
PI_BAD_CHAIN_CMD = -116
PI_BAD_CHAIN_DELAY = -117
PI_CHAIN_NESTING = -118
PI_CHAIN_TOO_BIG = -119
PI_DEPRECATED = -120
PI_BAD_SER_INVERT = -121
PI_BAD_FOREVER = -124
PI_BAD_FILTER = -125
PI_BAD_PAD = -126
PI_BAD_STRENGTH = -127
PI_FIL_OPEN_FAILED = -128
PI_BAD_FILE_MODE = -129
PI_BAD_FILE_FLAG = -130
PI_BAD_FILE_READ = -131
PI_BAD_FILE_WRITE = -132
PI_FILE_NOT_ROPEN = -133
PI_FILE_NOT_WOPEN = -134
PI_BAD_FILE_SEEK = -135
PI_NO_FILE_MATCH = -136
PI_NO_FILE_ACCESS = -137
PI_FILE_IS_A_DIR = -138
PI_BAD_SHELL_STATUS = -139
PI_BAD_SCRIPT_NAME = -140
PI_BAD_SPI_BAUD = -141
PI_NOT_SPI_GPIO = -142
PI_BAD_EVENT_ID = -143
PI_CMD_INTERRUPTED = -144
PI_NOT_ON_BCM2711 = -145
PI_ONLY_ON_BCM2711 = -146

event: 0-31
An event is a signal used to inform one or more consumers to start an action.
file_mode:
The mode may have the following values

FILE_READ 1
FILE_WRITE 2
FILE_RW 3

The following values can be or'd into the file open mode

FILE_APPEND 4
FILE_CREATE 8
FILE_TRUNC 16

file_name:
A full file path. To be accessible the path must match an entry in /opt/pigpio/access.
fpattern:
A file path which may contain wildcards. To be accessible the path must match an entry in /opt/pigpio/access.
frequency: 0-40000
Defines the frequency to be used for PWM on a GPIO. The closest permitted frequency will be used.
func:
A user supplied callback function.
gpio: 0-53
A Broadcom numbered GPIO. All the user GPIO are in the range 0-31.

There are 54 General Purpose Input Outputs (GPIO) named GPIO0 through GPIO53.

They are split into two banks. Bank 1 consists of GPIO0 through GPIO31. Bank 2 consists of GPIO32 through GPIO53.

All the GPIO which are safe for the user to read and write are in bank 1. Not all GPIO in bank 1 are safe though. Type 1 boards have 17 safe GPIO. Type 2 boards have 21. Type 3 boards have 26.

See get_hardware_revision.

The user GPIO are marked with an X in the following table

```
      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Type 1  X X - - X - - X X X X X - - X X
Type 2  - - X X X - - X X X X X - - X X
Type 3      X X X X X X X X X X X X X X

     16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Type 1  - X X - - X X X X X - - - - - -
Type 2  - X X - - - X X X X - X X X X X
Type 3  X X X X X X X X X X X X - - - -
```

gpio_off:
A mask used to select GPIO to be operated on. See bits.

This mask selects the GPIO to be switched off at the start of a pulse.
gpio_on:
A mask used to select GPIO to be operated on. See bits.

This mask selects the GPIO to be switched on at the start of a pulse.
handle: >=0
A number referencing an object opened by one of the following

file_open i2c_open notify_open serial_open spi_open
host:

The name or IP address of the Pi running the pigpio daemon.
i2c_address: 0-0x7F
The address of a device on the I2C bus.
i2c_bus: >=0
An I2C bus number.
i2c_flags: 0
No I2C flags are currently defined.
invert: 0-1
A flag used to set normal or inverted bit bang serial data level logic.
level: 0-1 (2)
CLEAR = 0
HIGH = 1
LOW = 0
OFF = 0
ON = 1
SET = 1
TIMEOUT = 2 # only returned for a watchdog timeout

MISO:
The GPIO used for the MISO signal when bit banging SPI.
mode:
1.The operational mode of a GPIO, normally INPUT or OUTPUT.

ALT0 = 4
ALT1 = 5
ALT2 = 6
ALT3 = 7
ALT4 = 3
ALT5 = 2
INPUT = 0
OUTPUT = 1

2. The mode of waveform transmission.

WAVE_MODE_ONE_SHOT = 0
WAVE_MODE_REPEAT = 1
WAVE_MODE_ONE_SHOT_SYNC = 2
WAVE_MODE_REPEAT_SYNC = 3

MOSI:
The GPIO used for the MOSI signal when bit banging SPI.
offset: >=0
The offset wave data starts from the beginning of the waveform being currently defined.
pad: 0-2
A set of GPIO which share common drivers.

Pad	GPIO
0	0-27
1	28-45

2	46-53

pad_strength: 1-16
The mA which may be drawn from each GPIO whilst still guaranteeing the high and low levels.
params: 32 bit number
When scripts are started they can receive up to 10 parameters to define their operation.
percent: : 0-100
The size of waveform as percentage of maximum available.
port:
The port used by the pigpio daemon, defaults to 8888.
pstring:
The string to be passed to a shell script to be executed.
pud: 0-2
PUD_DOWN = 1
PUD_OFF = 0
PUD_UP = 2

pulse_len: 1-100
The length of the trigger pulse in microseconds.
pulses:
A list of class pulse objects defining the characteristics of a waveform.
pulsewidth:
The servo pulsewidth in microseconds. 0 switches pulses off.
PWMduty: 0-1000000 (1M)
The hardware PWM dutycycle.
PWMfreq: 1-125M (1-187.5M for the BCM2711)
The hardware PWM frequency.
range_: 25-40000
Defines the limits for the dutycycle parameter.

range_ defaults to 255.
reg: 0-255
An I2C device register. The usable registers depend on the actual device.
retMax: >=0
The maximum number of bytes a user customised function should return, default 8192.
SCL:
The user GPIO to use for the clock when bit banging I2C.
SCLK: :
The GPIO used for the SCLK signal when bit banging SPI.
script:
The text of a script to store on the pigpio daemon.
script_id: >=0
A number referencing a script created by store_script.
SDA:
The user GPIO to use for data when bit banging I2C.
seek_from: 0-2
Direction to seek for file_seek.

FROM_START=0
FROM_CURRENT=1
FROM_END=2

seek_offset:
The number of bytes to move forward (positive) or backwards (negative) from the seek position (start, current, or end of file).
ser_flags: 32 bit
No serial flags are currently defined.
serial_*:
One of the serial_ functions.
shellscr:
The name of a shell script. The script must exist in /opt/pigpio/cgi and must be executable.
show_errors:
Controls the display of pigpio daemon connection failures. The default of True prints the probable failure reasons to standard output.
spi_channel: 0-2
A SPI channel.
spi_flags: 32 bit
See spi_open.
steady: 0-300000
The number of microseconds level changes must be stable for before reporting the level changed (set_glitch_filter) or triggering the active part of a noise filter (set_noise_filter).
t1:
A tick (earlier).
t2:
A tick (later).
tty:
A Pi serial tty device, e.g. /dev/ttyAMA0, /dev/ttyUSB0
uint32:
An unsigned 32 bit number.
user_gpio: 0-31
A Broadcom numbered GPIO.

All the user GPIO are in the range 0-31.

Not all the GPIO within this range are usable, some are reserved for system use.

See gpio.
wait_timeout: 0.0 -
The number of seconds to wait in wait_for_edge before timing out.
wave_add_*:
One of the following

wave_add_new wave_add_generic wave_add_serial
wave_id: >=0
A number referencing a wave created by wave_create.
wave_send_*:
One of the following

wave_send_once wave_send_repeat
wdog_timeout: 0-60000
Defines a GPIO watchdog timeout in milliseconds. If no level change is detected on the GPIO for timeout millisecond a watchdog timeout report is issued (with level TIMEOUT).
word_val: 0-65535
A whole number.