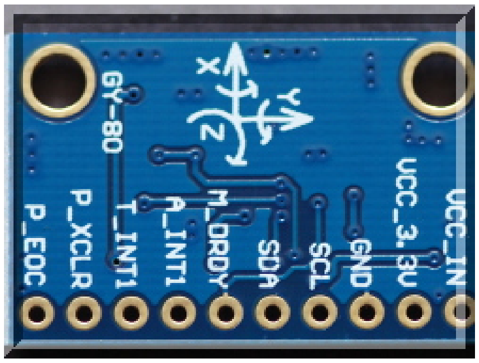
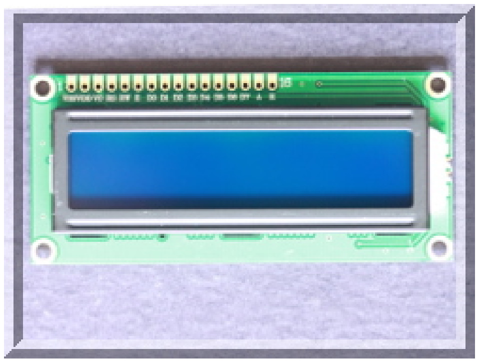

pigpio C Interface

pigpio is a C library for the Raspberry which allows control of the GPIO.

Features

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 callbacks at timed intervals

o reading/writing all of the GPIO in a bank as one operation

o individually setting GPIO modes, reading and writing

o notifications when any of GPIO 0-31 change state

o the construction of output waveforms with microsecond timing

o rudimentary permission control over GPIO

o a simple interface to start and stop new threads

o I2C, SPI, and serial link wrappers

o creating and running scripts

GPIO

ALL GPIO are identified by their Broadcom number.

Credits

The PWM and servo pulses are timed using the DMA and PWM peripherals.

This use was inspired by Richard Hirst's servoblaster kernel module.

Usage

Include <pigpio.h> in your source files.

Assuming your source is in prog.c use the following command to build and run the executable.

```
gcc -Wall -pthread -o prog prog.c -lpigpio -lrt
sudo ./prog
```

For examples of usage see the C programs within the pigpio archive file.

Notes

All the functions which return an int return < 0 on error.

gpioInitialise must be called before all other library functions with the following exceptions:

```
gpioCfg*
gpioVersion
gpioHardwareRevision
```

If the library is not initialised all but the gpioCfg*, gpioVersion, and gpioHardwareRevision functions will return error PI_NOT_INITIALISED.

If the library is initialised the gpioCfg* functions will return error PI_INITIALISED.

If you intend to rely on signals sent to your application, you should turn off the internal signal handling as shown in this example:

```
int cfg = gpioCfgGetInternals();
cfg |= PI_CFG_NOSIGHANDLER;  // (1<<10)
gpioCfgSetInternals(cfg);
int status = gpioInitialise();
```

OVERVIEW

ESSENTIAL	
gpioInitialise	Initialise library
gpioTerminate	Stop library
BASIC	
gpioSetMode	Set a GPIO mode
gpioGetMode	Get a GPIO mode
gpioSetPullUpDown	Set/clear GPIO pull up/down resistor

gpioRead	Read a GPIO
gpioWrite	Write a GPIO
PWM (overrides servo commands on same GPIO)	
gpioPWM	Start/stop PWM pulses on a GPIO
gpioSetPWMfrequency	Configure PWM frequency for a GPIO
gpioSetPWMrange	Configure PWM range for a GPIO
gpioGetPWMdutycycle	Get dutycycle setting on a GPIO
gpioGetPWMfrequency	Get configured PWM frequency for a GPIO
gpioGetPWMrange	Get configured PWM range for a GPIO
gpioGetPWMrealRange	Get underlying PWM range for a GPIO
Servo (overrides PWM commands on same GPIO)	
gpioServo	Start/stop servo pulses on a GPIO
gpioGetServoPulsewidth	Get pulsewidth setting on a GPIO
INTERMEDIATE	
gpioTrigger	Send a trigger pulse to a GPIO
gpioSetWatchdog	Set a watchdog on a GPIO
gpioRead_Bits_0_31	Read all GPIO in bank 1
gpioRead_Bits_32_53	Read all GPIO in bank 2
gpioWrite_Bits_0_31_Clear	Clear selected GPIO in bank 1
gpioWrite_Bits_32_53_Clear	Clear selected GPIO in bank 2
gpioWrite_Bits_0_31_Set	Set selected GPIO in bank 1
gpioWrite_Bits_32_53_Set	Set selected GPIO in bank 2
gpioSetAlertFunc	Request a GPIO level change callback
gpioSetAlertFuncEx	Request a GPIO change callback, extended
gpioSetTimerFunc	Request a regular timed callback
gpioSetTimerFuncEx	Request a regular timed callback, extended
gpioStartThread	Start a new thread
gpioStopThread	Stop a previously started thread
ADVANCED	
gpioNotifyOpen	Request a notification handle
gpioNotifyClose	Close a notification
gpioNotifyOpenWithSize	Request a notification with sized pipe
gpioNotifyBegin	Start notifications for selected GPIO
gpioNotifyPause	Pause notifications

gpioHardwareClock	Start hardware clock on supported GPIO
gpioHardwarePWM	Start hardware PWM on supported GPIO
gpioGlitchFilter	Set a glitch filter on a GPIO
gpioNoiseFilter	Set a noise filter on a GPIO
gpioSetPad	Sets a pads drive strength
gpioGetPad	Gets a pads drive strength
shell	Executes a shell command
gpioSetISRFunc	Request a GPIO interrupt callback
gpioSetISRFuncEx	Request a GPIO interrupt callback, extended
gpioSetSignalFunc	Request a signal callback
gpioSetSignalFuncEx	Request a signal callback, extended
gpioSetGetSamplesFunc	Requests a GPIO samples callback
gpioSetGetSamplesFuncEx	Requests a GPIO samples callback, extended
Custom	
gpioCustom1	User custom function 1
gpioCustom2	User custom function 2
Events	
eventMonitor	Sets the events to monitor
eventSetFunc	Request an event callback
eventSetFuncEx	Request an event callback, extended
eventTrigger	Trigger an event
Scripts	
gpioStoreScript	Store a script
gpioRunScript	Run a stored script
gpioUpdateScript	Set a scripts parameters
gpioScriptStatus	Get script status and parameters
gpioStopScript	Stop a running script
gpioDeleteScript	Delete a stored script
I2C	
i2cOpen	Opens an I2C device
i2cClose	Closes an I2C device
i2cWriteQuick	SMBus write quick
i2cReadByte	SMBus read byte
i2cWriteByte	SMBus write byte
i2cReadByteData	SMBus read byte data

i2cWriteByteData	SMBus write byte data
i2cReadWordData	SMBus read word data
i2cWriteWordData	SMBus write word data
i2cReadBlockData	SMBus read block data
i2cWriteBlockData	SMBus write block data
i2cReadI2CBlockData	SMBus read I2C block data
i2cWriteI2CBlockData	SMBus write I2C block data
i2cReadDevice	Reads the raw I2C device
i2cWriteDevice	Writes the raw I2C device
i2cProcessCall	SMBus process call
i2cBlockProcessCall	SMBus block process call
i2cSwitchCombined	Sets or clears the combined flag
i2cSegments	Performs multiple I2C transactions
i2cZip	Performs multiple I2C transactions
I2C BIT BANG	
bbI2COpen	Opens GPIO for bit banging I2C
bbI2CClose	Closes GPIO for bit banging I2C
bbI2CZip	Performs bit banged I2C transactions
I2C/SPI SLAVE	
bscXfer	I2C/SPI as slave transfer
SERIAL	
serOpen	Opens a serial device
serClose	Closes a serial device
serReadByte	Reads a byte from a serial device
serWriteByte	Writes a byte to a serial device
serRead	Reads bytes from a serial device
serWrite	Writes bytes to a serial device
serDataAvailable	Returns number of bytes ready to be read
SERIAL BIT BANG (read only)	
gpioSerialReadOpen	Opens a GPIO for bit bang serial reads
gpioSerialReadClose	Closes a GPIO for bit bang serial reads
gpioSerialReadInvert	Configures normal/inverted for serial reads
gpioSerialRead	Reads bit bang serial data from a GPIO
SPI	
spiOpen	Opens a SPI device
spiClose	Closes a SPI device

spiRead	Reads bytes from a SPI device
spiWrite	Writes bytes to a SPI device
spiXfer	Transfers bytes with a SPI device

SPI BIT BANG

bbSPIOpen	Opens GPIO for bit banging SPI
bbSPIClose	Closes GPIO for bit banging SPI
bbSPIXfer	Performs bit banged SPI transactions

FILES

fileOpen	Opens a file
fileClose	Closes a file
fileRead	Reads bytes from a file
fileWrite	Writes bytes to a file
fileSeek	Seeks to a position within a file
fileList	List files which match a pattern

WAVES

gpioWaveClear	Deletes all waveforms
gpioWaveAddNew	Starts a new waveform
gpioWaveAddGeneric	Adds a series of pulses to the waveform
gpioWaveAddSerial	Adds serial data to the waveform
gpioWaveCreate	Creates a waveform from added data
gpioWaveCreatePad	Creates a waveform of fixed size from added data
gpioWaveDelete	Deletes a waveform
gpioWaveTxSend	Transmits a waveform
gpioWaveChain	Transmits a chain of waveforms
gpioWaveTxAt	Returns the current transmitting waveform
gpioWaveTxBusy	Checks to see if the waveform has ended
gpioWaveTxStop	Aborts the current waveform
gpioWaveGetCbs	Length in CBs of the current waveform
gpioWaveGetHighCbs	Length of longest waveform so far
gpioWaveGetMaxCbs	Absolute maximum allowed CBs
gpioWaveGetMicros	Length in micros of the current waveform
gpioWaveGetHighMicros	Length of longest waveform so far
gpioWaveGetMaxMicros	Absolute maximum allowed micros
gpioWaveGetPulses	Length in pulses of the current waveform

gpioWaveGetHighPulses	Length of longest waveform so far
gpioWaveGetMaxPulses	Absolute maximum allowed pulses

UTILITIES

gpioDelay	Delay for a number of microseconds
gpioTick	Get current tick (microseconds)
gpioHardwareRevision	Get hardware revision
gpioVersion	Get the pigpio version
getBitInBytes	Get the value of a bit
putBitInBytes	Set the value of a bit
gpioTime	Get current time
gpioSleep	Sleep for specified time
time_sleep	Sleeps for a float number of seconds
time_time	Float number of seconds since the epoch

CONFIGURATION

gpioCfgBufferSize	Configure the GPIO sample buffer size
gpioCfgClock	Configure the GPIO sample rate
gpioCfgDMAchannel	Configure the DMA channel (DEPRECATED)
gpioCfgDMAchannels	Configure the DMA channels
gpioCfgPermissions	Configure the GPIO access permissions
gpioCfgInterfaces	Configure user interfaces
gpioCfgSocketPort	Configure socket port
gpioCfgMemAlloc	Configure DMA memory allocation mode
gpioCfgNetAddr	Configure allowed network addresses
gpioCfgGetInternals	Get internal configuration settings
gpioCfgSetInternals	Set internal configuration settings

EXPERT

rawWaveAddSPI	Not intended for general use
rawWaveAddGeneric	Not intended for general use
rawWaveCB	Not intended for general use
rawWaveCBAdr	Not intended for general use
rawWaveGetOOL	Not intended for general use
rawWaveSetOOL	Not intended for general use
rawWaveGetOut	Not intended for general use
rawWaveSetOut	Not intended for general use
rawWaveGetIn	Not intended for general use
rawWaveSetIn	Not intended for general use
rawWaveInfo	Not intended for general use
rawDumpWave	Not intended for general use

rawDumpScript	Not intended for general use

FUNCTIONS

int gpioInitialise(void)

Initialises the library.

Returns the pigpio version number if OK, otherwise PI_INIT_FAILED.

gpioInitialise must be called before using the other library functions with the following exceptions:

```
gpioCfg*
gpioVersion
gpioHardwareRevision
```

Example

```
if (gpioInitialise() < 0)
{
   // pigpio initialisation failed.
}
else
{
   // pigpio initialised okay.
}
```

void gpioTerminate(void)

Terminates the library.

Returns nothing.

Call before program exit.

This function resets the used DMA channels, releases memory, and terminates any running threads.

Example

```
gpioTerminate();
```

int gpioSetMode(unsigned gpio, unsigned mode)

Sets the GPIO mode, typically input or output.

```
gpio: 0-53
mode: 0-7
```

Returns 0 if OK, otherwise PI_BAD_GPIO or PI_BAD_MODE.

Arduino style: pinMode.

Example

```
gpioSetMode(17, PI_INPUT);  // Set GPIO17 as input.

gpioSetMode(18, PI_OUTPUT); // Set GPIO18 as output.

gpioSetMode(22,PI_ALT0);    // Set GPIO22 to alternative mode 0.
```

See
http://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf page 102 for an overview of the modes.

int gpioGetMode(unsigned gpio)

Gets the GPIO mode.

```
gpio: 0-53
```

Returns the GPIO mode if OK, otherwise PI_BAD_GPIO.

Example

```
if (gpioGetMode(17) != PI_ALT0)
{
   gpioSetMode(17, PI_ALT0);  // set GPIO17 to ALT0
}
```

int gpioSetPullUpDown(unsigned gpio, unsigned pud)

Sets or clears resistor pull ups or downs on the GPIO.

```
gpio: 0-53
 pud: 0-2
```

Returns 0 if OK, otherwise PI_BAD_GPIO or PI_BAD_PUD.

Example

```
gpioSetPullUpDown(17, PI_PUD_UP);   // Sets a pull-up.

gpioSetPullUpDown(18, PI_PUD_DOWN); // Sets a pull-down.

gpioSetPullUpDown(23, PI_PUD_OFF);  // Clear any pull-ups/downs.
```

int gpioRead(unsigned gpio)

Reads the GPIO level, on or off.

```
gpio: 0-53
```

Returns the GPIO level if OK, otherwise PI_BAD_GPIO.

Arduino style: digitalRead.

Example

```
printf("GPIO24 is level %d", gpioRead(24));
```

int gpioWrite(unsigned gpio, unsigned level)

Sets the GPIO level, on or off.

```
 gpio: 0-53
level: 0-1
```

Returns 0 if OK, otherwise PI_BAD_GPIO or PI_BAD_LEVEL.

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

Arduino style: digitalWrite

Example

```
gpioWrite(24, 1); // Set GPIO24 high.
```

int gpioPWM(unsigned user_gpio, unsigned dutycycle)

Starts PWM on the GPIO, dutycycle between 0 (off) and range (fully on). Range defaults to 255.

```
user_gpio: 0-31
dutycycle: 0-range
```

Returns 0 if OK, otherwise PI_BAD_USER_GPIO or PI_BAD_DUTYCYCLE.

Arduino style: analogWrite

This and the servo functionality use the DMA and PWM or PCM peripherals to control and schedule the pulse lengths and dutycycles.

The gpioSetPWMrange function may be used to change the default range of 255.

Example

```
gpioPWM(17, 255); // Sets GPIO17 full on.

gpioPWM(18, 128); // Sets GPIO18 half on.

gpioPWM(23, 0);   // Sets GPIO23 full off.
```

int gpioGetPWMdutycycle(unsigned user_gpio)

Returns the PWM dutycycle setting for the GPIO.

```
user_gpio: 0-31
```

Returns between 0 (off) and range (fully on) if OK, otherwise PI_BAD_USER_GPIO or

PI_NOT_PWM_GPIO.

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

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).

Normal PWM range defaults to 255.

int gpioSetPWMrange(unsigned user_gpio, unsigned range)

Selects the dutycycle range to be used for the GPIO. Subsequent calls to gpioPWM will use a dutycycle between 0 (off) and range (fully on).

```
user_gpio: 0-31
    range: 25-40000
```

Returns the real range for the given GPIO's frequency if OK, otherwise PI_BAD_USER_GPIO or PI_BAD_DUTYRANGE.

If PWM is currently active on the GPIO its dutycycle will be scaled to reflect the new range.

The real range, the number of steps between fully off and fully on for each frequency, is given in the following table.

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

The real value set by gpioPWM is (dutycycle * real range) / range.

Example

```
gpioSetPWMrange(24, 2000); // Now 2000 is fully on
                           //     1000 is half on
                           //      500 is quarter on, etc.
```

int gpioGetPWMrange(unsigned user_gpio)

Returns the dutycycle range used for the GPIO if OK, otherwise PI_BAD_USER_GPIO.

```
user_gpio: 0-31
```

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

Example

```
r = gpioGetPWMrange(23);
```

int gpioGetPWMrealRange(unsigned user_gpio)

Returns the real range used for the GPIO if OK, otherwise PI_BAD_USER_GPIO.

```
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

```
rr = gpioGetPWMrealRange(17);
```

int gpioSetPWMfrequency(unsigned user_gpio, unsigned frequency)

Sets the frequency in hertz to be used for the GPIO.

```
user_gpio: 0-31
frequency: >=0
```

Returns the numerically closest frequency if OK, otherwise PI_BAD_USER_GPIO.

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 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

```
gpioSetPWMfrequency(23, 0); // Set GPIO23 to lowest frequency.

gpioSetPWMfrequency(24, 500); // Set GPIO24 to 500Hz.

gpioSetPWMfrequency(25, 100000); // Set GPIO25 to highest frequency.
```

int gpioGetPWMfrequency(unsigned user_gpio)

Returns the frequency (in hertz) used for the GPIO if OK, otherwise PI_BAD_USER_GPIO.

```
user_gpio: 0-31
```

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

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

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

Example

```
f = gpioGetPWMfrequency(23); // Get frequency used for GPIO23.
```

int gpioServo(unsigned user_gpio, unsigned pulsewidth)

Starts servo pulses on the GPIO, 0 (off), 500 (most anti-clockwise) to 2500 (most clockwise).

```
 user_gpio: 0-31
pulsewidth: 0, 500-2500
```

Returns 0 if OK, otherwise PI_BAD_USER_GPIO or PI_BAD_PULSEWIDTH.

The range 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.

The following causes an on pulse of 1500 microseconds duration to be transmitted on GPIO 17 at a rate of 50 times per second. This will command a servo connected to GPIO 17 to rotate to its mid-point.

Example

```
gpioServo(17, 1000); // Move servo to safe position anti-clockwise.

gpioServo(23, 1500); // Move servo to centre position.

gpioServo(25, 2000); // Move servo to safe position clockwise.
```

OTHER UPDATE RATES:

This function updates servos at 50Hz. If you wish to use a different update frequency you will have to use the PWM functions.

```
PWM Hz    50   100  200  400  500
1E6/Hz 20000 10000 5000 2500 2000
```

Firstly set the desired PWM frequency using gpioSetPWMfrequency.

Then set the PWM range using gpioSetPWMrange to 1E6/frequency. Doing this allows you to use units of microseconds when setting the servo pulsewidth.

E.g. If you want to update a servo connected to GPIO25 at 400Hz

```
gpioSetPWMfrequency(25, 400);

gpioSetPWMrange(25, 2500);
```

Thereafter use the PWM command to move the servo, e.g. gpioPWM(25, 1500) will set a 1500 us pulse.

int gpioGetServoPulsewidth(unsigned user_gpio)

Returns the servo pulsewidth setting for the GPIO.

```
user_gpio: 0-31
```

Returns 0 (off), 500 (most anti-clockwise) to 2500 (most clockwise) if OK, otherwise PI_BAD_USER_GPIO or PI_NOT_SERVO_GPIO.

int gpioSetAlertFunc(unsigned user_gpio, gpioAlertFunc_t f)

Registers a function to be called (a callback) when the specified GPIO changes state.

```
user_gpio: 0-31
        f: the callback function
```

Returns 0 if OK, otherwise PI_BAD_USER_GPIO.

One callback may be registered per GPIO.

The callback is passed the GPIO, the new 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
```

The alert may be cancelled by passing NULL as the function.

The GPIO are sampled at a rate set when the library is started.

If a value isn't specifically set the default of 5 us is used.

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

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

Level changes shorter than the sample rate may be missed.

The thread which calls the alert functions is triggered nominally 1000 times per second. The active alert functions will be called once per level change since the last time the thread was activated. i.e. The active alert functions 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 gpioRead. Remember the event that triggered the callback may have happened several milliseconds before and the GPIO may have changed level many times since then.

The tick value is the time stamp of the sample in microseconds, see gpioTick for more details.

Example

```
void aFunction(int gpio, int level, uint32_t tick)
{
   printf("GPIO %d became %d at %d", gpio, level, tick);
}

// call aFunction whenever GPIO 4 changes state

gpioSetAlertFunc(4, aFunction);
```

int gpioSetAlertFuncEx(unsigned user_gpio, gpioAlertFuncEx_t f, void *userdata)

Registers a function to be called (a callback) when the specified GPIO changes state.

```
user_gpio: 0-31
        f: the callback function
 userdata: pointer to arbitrary user data
```

Returns 0 if OK, otherwise PI_BAD_USER_GPIO.

One callback may be registered per GPIO.

The callback is passed the GPIO, the new level, the tick, and the userdata pointer.

```
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

userdata    pointer  Pointer to an arbitrary object
```

See gpioSetAlertFunc for further details.

Only one of gpioSetAlertFunc or gpioSetAlertFuncEx can be registered per GPIO.

int gpioSetISRFunc(unsigned gpio, unsigned edge, int timeout, gpioISRFunc_t f)

Registers a function to be called (a callback) whenever the specified GPIO interrupt occurs.

```
   gpio: 0-53
   edge: RISING_EDGE, FALLING_EDGE, or EITHER_EDGE
timeout: interrupt timeout in milliseconds (<=0 to cancel)
      f: the callback function
```

Returns 0 if OK, otherwise PI_BAD_GPIO, PI_BAD_EDGE, or PI_BAD_ISR_INIT.

One function may be registered per GPIO.

The function is passed the GPIO, the current level, and the current tick. The level will be PI_TIMEOUT if the optional interrupt timeout expires.

```
Parameter   Value    Meaning

GPIO        0-53     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 (interrupt timeout)

tick        32 bit   The number of microseconds since boot
                     WARNING: this wraps around from
                     4294967295 to 0 roughly every 72 minutes
```

The underlying Linux sysfs GPIO interface is used to provide the interrupt services.

The first time the function is called, with a non-NULL f, the GPIO is exported, set to be an input, and set to interrupt on the given edge and timeout.

Subsequent calls, with a non-NULL f, can vary one or more of the edge, timeout, or function.

The ISR may be cancelled by passing a NULL f, in which case the GPIO is unexported.

The tick is that read at the time the process was informed of the interrupt. This will be a variable number of microseconds after the interrupt occurred. Typically the latency will be of the order of 50 microseconds. The latency is not guaranteed and will vary with system load.

The level is that read at the time the process was informed of the interrupt, or PI_TIMEOUT if the optional interrupt timeout expired. It may not be the same as the expected edge as interrupts happening in rapid succession may be missed by the kernel (i.e. this mechanism can not be used to capture several interrupts only a few microseconds apart).

int gpioSetISRFuncEx(unsigned gpio, unsigned edge, int timeout, gpioISRFuncEx_t f, void *userdata)

Registers a function to be called (a callback) whenever the specified GPIO interrupt occurs.

```
    gpio: 0-53
    edge: RISING_EDGE, FALLING_EDGE, or EITHER_EDGE
 timeout: interrupt timeout in milliseconds (<=0 to cancel)
       f: the callback function
userdata: pointer to arbitrary user data
```

Returns 0 if OK, otherwise PI_BAD_GPIO, PI_BAD_EDGE, or PI_BAD_ISR_INIT.

The function is passed the GPIO, the current level, the current tick, and the userdata pointer.

```
Parameter   Value    Meaning

GPIO        0-53     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 (interrupt timeout)

tick        32 bit   The number of microseconds since boot
                     WARNING: this wraps around from
                     4294967295 to 0 roughly every 72 minutes

userdata    pointer  Pointer to an arbitrary object
```

Only one of gpioSetISRFunc or gpioSetISRFuncEx can be registered per GPIO.

See gpioSetISRFunc for further details.

int gpioNotifyOpen(void)

This function requests a free notification handle.

Returns a handle greater than or equal to zero if OK, otherwise PI_NO_HANDLE.

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

Pipe 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.

Socket notifications are returned to the socket which requested the handle.

Example

```
h = gpioNotifyOpen();

if (h >= 0)
{
   sprintf(str, "/dev/pigpio%d", h);

   fd = open(str, O_RDONLY);

   if (fd >= 0)
   {
      // Okay.
   }
   else
   {
      // Error.
   }
}
else
{
   // Error.
}
```

int gpioNotifyOpenWithSize(int bufSize)

This function requests a free notification handle.

It differs from gpioNotifyOpen in that the pipe size may be specified, whereas gpioNotifyOpen uses the default pipe size.

See gpioNotifyOpen for further details.

int gpioNotifyBegin(unsigned handle, uint32_t bits)

This function starts notifications on a previously opened handle.

```
handle: >=0, as returned by gpioNotifyOpen
  bits: a bit mask indicating the GPIO of interest
```

Returns 0 if OK, otherwise PI_BAD_HANDLE.

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

Each notification occupies 12 bytes in the fifo and has the following structure.

```
typedef struct
{
   uint16_t seqno;
   uint16_t flags;
   uint32_t tick;
   uint32_t level;
} gpioReport_t;
```

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

```
// Start notifications for GPIO 1, 4, 6, 7, 10.

//                         1
//                         0  76 4  1
// (1234 = 0x04D2 = 0b0000010011010010)

gpioNotifyBegin(h, 1234);
```

int gpioNotifyPause(unsigned handle)

This function pauses notifications on a previously opened handle.

```
handle: >=0, as returned by gpioNotifyOpen
```

Returns 0 if OK, otherwise PI_BAD_HANDLE.

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

Example

```
gpioNotifyPause(h);
```

int gpioNotifyClose(unsigned handle)

This function stops notifications on a previously opened handle and releases the handle for reuse.

```
handle: >=0, as returned by gpioNotifyOpen
```

Returns 0 if OK, otherwise PI_BAD_HANDLE.

Example

```
gpioNotifyClose(h);
```

int gpioWaveClear(void)

This function clears all waveforms and any data added by calls to the gpioWaveAdd* functions.

Returns 0 if OK.

Example

```
gpioWaveClear();
```

int gpioWaveAddNew(void)

This function starts a new empty waveform.

You wouldn't normally need to call this function as it is automatically called after a waveform is created with the gpioWaveCreate function.

Returns 0 if OK.

Example

```
gpioWaveAddNew();
```

int gpioWaveAddGeneric(unsigned numPulses, gpioPulse_t *pulses)

This function adds a number of pulses to the current waveform.

```
numPulses: the number of pulses
   pulses: an array of pulses
```

Returns the new total number of pulses in the current waveform if OK, otherwise PI_TOO_MANY_PULSES.

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 of a delay.

Example

```
// Construct and send a 30 microsecond square wave.

gpioSetMode(gpio, PI_OUTPUT);

pulse[0].gpioOn = (1<<gpio);
```

```
pulse[0].gpioOff = 0;
pulse[0].usDelay = 15;

pulse[1].gpioOn = 0;
pulse[1].gpioOff = (1<<gpio);
pulse[1].usDelay = 15;

gpioWaveAddNew();

gpioWaveAddGeneric(2, pulse);

wave_id = gpioWaveCreate();

if (wave_id >= 0)
{
   gpioWaveTxSend(wave_id, PI_WAVE_MODE_REPEAT);

   // Transmit for 30 seconds.

   sleep(30);

   gpioWaveTxStop();
}
else
{
   // Wave create failed.
}
```

int gpioWaveAddSerial(unsigned user_gpio, unsigned baud, unsigned data_bits, unsigned stop_bits, unsigned offset, unsigned numBytes, char *str)

This function adds a waveform representing serial data to the existing waveform (if any). The serial data starts offset microseconds from the start of the waveform.

```
user_gpio: 0-31
     baud: 50-1000000
data_bits: 1-32
stop_bits: 2-8
   offset: >=0
 numBytes: >=1
      str: an array of chars (which may contain nulls)
```

Returns the new total number of pulses in the current waveform if OK, otherwise PI_BAD_USER_GPIO, PI_BAD_WAVE_BAUD, PI_BAD_DATABITS, PI_BAD_STOPBITS, PI_TOO_MANY_CHARS, PI_BAD_SER_OFFSET, or PI_TOO_MANY_PULSES.

NOTES:

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

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

numBytes is the number of bytes of data in str.

The bytes required for each character depend upon data_bits.

For data_bits 1-8 there will be one byte per character.

For data_bits 9-16 there will be two bytes per character.
For data_bits 17-32 there will be four bytes per character.

Example

```
#define MSG_LEN 8

int i;
char *str;
char data[MSG_LEN];

str = "Hello world!";

gpioWaveAddSerial(4, 9600, 8, 2, 0, strlen(str), str);

for (i=0; i<MSG_LEN; i++) data[i] = i;

// Data added is offset 1 second from the waveform start.
gpioWaveAddSerial(4, 9600, 8, 2, 1000000, MSG_LEN, data);
```

int gpioWaveCreate(void)

This function creates a waveform from the data provided by the prior calls to the gpioWaveAdd* functions. 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.

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

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

Normal usage would be

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

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

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

Repeat steps 2 and 3 as needed.

Step 4. gpioWaveTxSend with the id of the waveform to transmit.

A waveform comprises one of more pulses. Each pulse consists of a gpioPulse_t structure.

```
typedef struct
{
   uint32_t gpioOn;
   uint32_t gpioOff;
   uint32_t usDelay;
} gpioPulse_t;
```

The fields specify

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.

Returns the new waveform id if OK, otherwise PI_EMPTY_WAVEFORM, PI_NO_WAVEFORM_ID, PI_TOO_MANY_CBS, or PI_TOO_MANY_OOL.

int gpioWaveCreatePad(int pctCB, int pctBOOL, int pctTOOL)

Similar to gpioWaveCreate, this function creates a waveform but pads the consumed resources. Padded waves of equal dimension can be re-cycled efficiently allowing newly created waves to re-use the resources of deleted waves of the same dimension.

```
pctCB: 0-100, the percent of all DMA control blocks to consume.
pctBOOL: 0-100, percent On-Off-Level (OOL) buffer to consume for wave output.
pctTOOL: 0-100, the percent of OOL buffer to consume for wave input (flags).
```

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.

Waveform data provided by gpioWaveAdd* and rawWaveAdd* functions are consumed by this function.

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.

Example

```
  // get firstWaveChunk, somehow
  gpioWaveAddGeneric(firstWaveChunk);
  wid = gpioWaveCreatePad(50, 50, 0);
  gpioWaveTxSend(wid, PI_WAVE_MODE_ONE_SHOT);
  // get nextWaveChunk

  while (nextWaveChunk) {
     gpioWaveAddGeneric(nextWaveChunk);
     nextWid = gpioWaveCreatePad(50, 50, 0);
     gpioWaveTxSend(nextWid, PI_WAVE_MODE_ONE_SHOT_SYNC);
     while(gpioWaveTxAt() == wid) time_sleep(0.1);
     gpioWaveDelete(wid);
     wid = nextWid;
     // get nextWaveChunk
  }
```

int gpioWaveDelete(unsigned wave_id)

This function deletes the waveform with id wave_id.

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).

```
wave_id: >=0, as returned by gpioWaveCreate
```

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

Returns 0 if OK, otherwise PI_BAD_WAVE_ID.

int gpioWaveTxSend(unsigned wave_id, unsigned wave_mode)

This function transmits the waveform with id wave_id. The mode determines whether the waveform is sent once or cycles endlessly. The SYNC variants wait for the current waveform to reach the end of a cycle or finish before starting the new 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 gpioHardwarePWM will be cancelled.

```
  wave_id: >=0, as returned by gpioWaveCreate
wave_mode: PI_WAVE_MODE_ONE_SHOT, PI_WAVE_MODE_REPEAT,
           PI_WAVE_MODE_ONE_SHOT_SYNC, PI_WAVE_MODE_REPEAT_SYNC
```

Returns the number of DMA control blocks in the waveform if OK, otherwise PI_BAD_WAVE_ID, or PI_BAD_WAVE_MODE.

int gpioWaveChain(char *buf, unsigned bufSize)

This function transmits a chain of waveforms.

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

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

```
    buf: pointer to the wave_ids and optional command codes
bufSize: the number of bytes in buf
```

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

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

#define WAVES 5
#define GPIO 4

int main(int argc, char *argv[])
{
   int i, wid[WAVES];

   if (gpioInitialise()<0) return -1;

   gpioSetMode(GPIO, PI_OUTPUT);

   printf("start piscope, press return"); getchar();

   for (i=0; i<WAVES; i++)
   {
      gpioWaveAddGeneric(2, (gpioPulse_t[])
         {{1<<GPIO, 0,        20},
          {0, 1<<GPIO, (i+1)*200}});

      wid[i] = gpioWaveCreate();
   }

   gpioWaveChain((char []) {
      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

      }, 46);

   while (gpioWaveTxBusy()) time_sleep(0.1);

   for (i=0; i<WAVES; i++) gpioWaveDelete(wid[i]);

   printf("stop piscope, press return"); getchar();
   gpioTerminate();
}
```

int gpioWaveTxAt(void)

This function returns the id of the waveform currently being transmitted using gpioWaveTxSend. Chained waves are not supported.

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

PI_WAVE_NOT_FOUND (9998) - transmitted wave not found.
PI_NO_TX_WAVE (9999) - no wave being transmitted.

int gpioWaveTxBusy(void)

This function checks to see if a waveform is currently being transmitted.

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

int gpioWaveTxStop(void)

This function aborts the transmission of the current waveform.

Returns 0 if OK.

This function is intended to stop a waveform started in repeat mode.

int gpioWaveGetMicros(void)

This function returns the length in microseconds of the current waveform.

int gpioWaveGetHighMicros(void)

This function returns the length in microseconds of the longest waveform created since gpioInitialise was called.

int gpioWaveGetMaxMicros(void)

This function returns the maximum possible size of a waveform in microseconds.

int gpioWaveGetPulses(void)

This function returns the length in pulses of the current waveform.

int gpioWaveGetHighPulses(void)

This function returns the length in pulses of the longest waveform created since gpioInitialise was called.

int gpioWaveGetMaxPulses(void)

This function returns the maximum possible size of a waveform in pulses.

int gpioWaveGetCbs(void)

This function returns the length in DMA control blocks of the current waveform.

int gpioWaveGetHighCbs(void)

This function returns the length in DMA control blocks of the longest waveform created since gpioInitialise was called.

int gpioWaveGetMaxCbs(void)

This function returns the maximum possible size of a waveform in DMA control blocks.

int gpioSerialReadOpen(unsigned user_gpio, unsigned baud, unsigned data_bits)

This function opens a GPIO for bit bang reading of serial data.

```
user_gpio: 0-31
     baud: 50-250000
data_bits: 1-32
```

Returns 0 if OK, otherwise PI_BAD_USER_GPIO, PI_BAD_WAVE_BAUD, PI_BAD_DATABITS, or PI_GPIO_IN_USE.

The serial data is returned in a cyclic buffer and is read using gpioSerialRead.

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

int gpioSerialReadInvert(unsigned user_gpio, unsigned invert)

This function configures the level logic for bit bang serial reads.

Use PI_BB_SER_INVERT to invert the serial logic and PI_BB_SER_NORMAL for normal logic. Default is PI_BB_SER_NORMAL.

```
user_gpio: 0-31
   invert: 0-1
```

Returns 0 if OK, otherwise PI_BAD_USER_GPIO, PI_GPIO_IN_USE, PI_NOT_SERIAL_GPIO, or PI_BAD_SER_INVERT.

The GPIO must be opened for bit bang reading of serial data using gpioSerialReadOpen prior to calling this function.

int gpioSerialRead(unsigned user_gpio, void *buf, size_t bufSize)

This function copies up to bufSize bytes of data read from the bit bang serial cyclic buffer to the buffer starting at buf.

```
user_gpio: 0-31, previously opened with gpioSerialReadOpen
      buf: an array to receive the read bytes
  bufSize: >=0
```

Returns the number of bytes copied if OK, otherwise PI_BAD_USER_GPIO or PI_NOT_SERIAL_GPIO.

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

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

int gpioSerialReadClose(unsigned user_gpio)

This function closes a GPIO for bit bang reading of serial data.

```
user_gpio: 0-31, previously opened with gpioSerialReadOpen
```

Returns 0 if OK, otherwise PI_BAD_USER_GPIO, or PI_NOT_SERIAL_GPIO.

int i2cOpen(unsigned i2cBus, unsigned i2cAddr, unsigned i2cFlags)

This returns a handle for the device at the address on the I2C bus.

```
  i2cBus: >=0
 i2cAddr: 0-0x7F
i2cFlags: 0
```

No flags are currently defined. This parameter should be set to zero.

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

Returns a handle (>=0) if OK, otherwise PI_BAD_I2C_BUS, PI_BAD_I2C_ADDR, PI_BAD_FLAGS, PI_NO_HANDLE, or PI_I2C_OPEN_FAILED.

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.
i2cReg  (8 bits): Command byte, a 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.
```

int i2cClose(unsigned handle)

This closes the I2C device associated with the handle.

```
handle: >=0, as returned by a call to i2cOpen
```

Returns 0 if OK, otherwise PI_BAD_HANDLE.

int i2cWriteQuick(unsigned handle, unsigned bit)

This sends a single bit (in the Rd/Wr bit) to the device associated with handle.

```
handle: >=0, as returned by a call to i2cOpen
   bit: 0-1, the value to write
```

Returns 0 if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_WRITE_FAILED.

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

int i2cWriteByte(unsigned handle, unsigned bVal)

This sends a single byte to the device associated with handle.

```
handle: >=0, as returned by a call to i2cOpen
  bVal: 0-0xFF, the value to write
```

Returns 0 if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_WRITE_FAILED.

Send byte. SMBus 2.0 5.5.2
```
S Addr Wr [A] bVal [A] P
```

int i2cReadByte(unsigned handle)

This reads a single byte from the device associated with handle.

```
handle: >=0, as returned by a call to i2cOpen
```

Returns the byte read (>=0) if OK, otherwise PI_BAD_HANDLE, or

PI_I2C_READ_FAILED.

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

int i2cWriteByteData(unsigned handle, unsigned i2cReg, unsigned bVal)

This writes a single byte to the specified register of the device associated with handle.

```
handle: >=0, as returned by a call to i2cOpen
i2cReg: 0-255, the register to write
  bVal: 0-0xFF, the value to write
```

Returns 0 if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_WRITE_FAILED.

Write byte. SMBus 2.0 5.5.4
```
S Addr Wr [A] i2cReg [A] bVal [A] P
```

int i2cWriteWordData(unsigned handle, unsigned i2cReg, unsigned wVal)

This writes a single 16 bit word to the specified register of the device associated with handle.

```
handle: >=0, as returned by a call to i2cOpen
i2cReg: 0-255, the register to write
  wVal: 0-0xFFFF, the value to write
```

Returns 0 if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_WRITE_FAILED.

Write word. SMBus 2.0 5.5.4
```
S Addr Wr [A] i2cReg [A] wValLow [A] wValHigh [A] P
```

int i2cReadByteData(unsigned handle, unsigned i2cReg)

This reads a single byte from the specified register of the device associated with handle.

```
handle: >=0, as returned by a call to i2cOpen
i2cReg: 0-255, the register to read
```

Returns the byte read (>=0) if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_READ_FAILED.

Read byte. SMBus 2.0 5.5.5
```
S Addr Wr [A] i2cReg [A] S Addr Rd [A] [Data] NA P
```

int i2cReadWordData(unsigned handle, unsigned i2cReg)

This reads a single 16 bit word from the specified register of the device associated with handle.

```
handle: >=0, as returned by a call to i2cOpen
i2cReg: 0-255, the register to read
```

Returns the word read (>=0) if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_READ_FAILED.

Read word. SMBus 2.0 5.5.5
```
S Addr Wr [A] i2cReg [A] S Addr Rd [A] [DataLow] A [DataHigh] NA P
```

int i2cProcessCall(unsigned handle, unsigned i2cReg, unsigned wVal)

This writes 16 bits of data to the specified register of the device associated with handle and reads 16 bits of data in return.

```
handle: >=0, as returned by a call to i2cOpen
i2cReg: 0-255, the register to write/read
  wVal: 0-0xFFFF, the value to write
```

Returns the word read (>=0) if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_READ_FAILED.

Process call. SMBus 2.0 5.5.6
```
S Addr Wr [A] i2cReg [A] wValLow [A] wValHigh [A]
   S Addr Rd [A] [DataLow] A [DataHigh] NA P
```

int i2cWriteBlockData(unsigned handle, unsigned i2cReg, char *buf, unsigned count)

This writes up to 32 bytes to the specified register of the device associated with handle.

```
handle: >=0, as returned by a call to i2cOpen
i2cReg: 0-255, the register to write
   buf: an array with the data to send
 count: 1-32, the number of bytes to write
```

Returns 0 if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_WRITE_FAILED.

Block write. SMBus 2.0 5.5.7
```
S Addr Wr [A] i2cReg [A] count [A]
   buf0 [A] buf1 [A] ... [A] bufn [A] P
```

int i2cReadBlockData(unsigned handle, unsigned i2cReg, char *buf)

This reads a block of up to 32 bytes from the specified register of the device associated with handle.

```
handle: >=0, as returned by a call to i2cOpen
i2cReg: 0-255, the register to read
   buf: an array to receive the read data
```

The amount of returned data is set by the device.

Returns the number of bytes read (>=0) if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_READ_FAILED.

Block read. SMBus 2.0 5.5.7

```
S Addr Wr [A] i2cReg [A]
   S Addr Rd [A] [Count] A [buf0] A [buf1] A ... A [bufn] NA P
```

int i2cBlockProcessCall(unsigned handle, unsigned i2cReg, char *buf, unsigned count)

This 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.

```
handle: >=0, as returned by a call to i2cOpen
i2cReg: 0-255, the register to write/read
   buf: an array with the data to send and to receive the read data
 count: 1-32, the number of bytes to write
```

Returns the number of bytes read (>=0) if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_READ_FAILED.

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.

Block write-block read. SMBus 2.0 5.5.8

```
S Addr Wr [A] i2cReg [A] count [A] buf0 [A] ... bufn [A]
   S Addr Rd [A] [Count] A [buf0] A ... [bufn] A P
```

int i2cReadI2CBlockData(unsigned handle, unsigned i2cReg, char *buf, unsigned count)

This reads count bytes from the specified register of the device associated with handle . The count may be 1-32.

```
handle: >=0, as returned by a call to i2cOpen
i2cReg: 0-255, the register to read
   buf: an array to receive the read data
 count: 1-32, the number of bytes to read
```

Returns the number of bytes read (>0) if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_READ_FAILED.

```
S Addr Wr [A] i2cReg [A]
   S Addr Rd [A] [buf0] A [buf1] A ... A [bufn] NA P
```

int i2cWriteI2CBlockData(unsigned handle, unsigned i2cReg, char *buf, unsigned count)

This writes 1 to 32 bytes to the specified register of the device associated with handle.

```
handle: >=0, as returned by a call to i2cOpen
i2cReg: 0-255, the register to write
   buf: the data to write
 count: 1-32, the number of bytes to write
```

Returns 0 if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_WRITE_FAILED.

```
S Addr Wr [A] i2cReg [A] buf0 [A] buf1 [A] ... [A] bufn [A] P
```

int i2cReadDevice(unsigned handle, char *buf, unsigned count)

This reads count bytes from the raw device into buf.

```
handle: >=0, as returned by a call to i2cOpen
   buf: an array to receive the read data bytes
 count: >0, the number of bytes to read
```

Returns count (>0) if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_READ_FAILED.

```
S Addr Rd [A] [buf0] A [buf1] A ... A [bufn] NA P
```

int i2cWriteDevice(unsigned handle, char *buf, unsigned count)

This writes count bytes from buf to the raw device.

```
handle: >=0, as returned by a call to i2cOpen
   buf: an array containing the data bytes to write
 count: >0, the number of bytes to write
```

Returns 0 if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_I2C_WRITE_FAILED.

```
S Addr Wr [A] buf0 [A] buf1 [A] ... [A] bufn [A] P
```

void i2cSwitchCombined(int setting)

This sets the I2C (i2c-bcm2708) module "use combined transactions" parameter on or off.

```
setting: 0 to set the parameter off, non-zero to set it on
```

NOTE: when the flag is on a write followed by a read to the same slave address will use a repeated start (rather than a stop/start).

int i2cSegments(unsigned handle, pi_i2c_msg_t *segs, unsigned numSegs)

This function executes multiple I2C segments in one transaction by calling the I2C_RDWR ioctl.

```
 handle: >=0, as returned by a call to i2cOpen
   segs: an array of I2C segments
numSegs: >0, the number of I2C segments
```

Returns the number of segments if OK, otherwise PI_BAD_I2C_SEG.

int i2cZip(unsigned handle, char *inBuf, unsigned inLen, char *outBuf, unsigned outLen)

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

```
handle: >=0, as returned by a call to i2cOpen
 inBuf: pointer to the concatenated I2C commands, see below
 inLen: size of command buffer
outBuf: pointer to buffer to hold returned data
outLen: size of output buffer
```

Returns >= 0 if OK (the number of bytes read), otherwise PI_BAD_HANDLE, PI_BAD_POINTER, PI_BAD_I2C_CMD, PI_BAD_I2C_RLEN. PI_BAD_I2C_WLEN, or PI_BAD_I2C_SEG.

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.

The returned I2C data is stored in consecutive locations of outBuf.

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
```

int bbI2COpen(unsigned SDA, unsigned SCL, unsigned 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

```
 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.

int bbI2CClose(unsigned SDA)

This function stops bit banging I2C on a pair of GPIO previously opened with bbI2COpen.

```
SDA: 0-31, the SDA GPIO used in a prior call to bbI2COpen
```

Returns 0 if OK, otherwise PI_BAD_USER_GPIO, or PI_NOT_I2C_GPIO.

int bbI2CZip(unsigned SDA, char *inBuf, unsigned inLen, char *outBuf, unsigned outLen)

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

```
   SDA: 0-31 (as used in a prior call to bbI2COpen)
 inBuf: pointer to the concatenated I2C commands, see below
 inLen: size of command buffer
outBuf: pointer to buffer to hold returned data
outLen: size of output buffer
```

Returns >= 0 if OK (the number of bytes read), otherwise PI_BAD_USER_GPIO, PI_NOT_I2C_GPIO, PI_BAD_POINTER, PI_BAD_I2C_CMD, PI_BAD_I2C_RLEN, PI_BAD_I2C_WLEN, PI_I2C_READ_FAILED, or PI_I2C_WRITE_FAILED.

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.

The returned I2C data is stored in consecutive locations of outBuf.

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
```

int bscXfer(bsc_xfer_t *bsc_xfer)

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.

```
bsc_xfer:= a structure defining the transfer

typedef struct
{
   uint32_t control;          // Write
   int rxCnt;                 // Read only
   char rxBuf[BSC_FIFO_SIZE]; // Read only
   int txCnt;                 // Write
   char txBuf[BSC_FIFO_SIZE]; // Write
} bsc_xfer_t;
```

To start a transfer set control (see below), copy the bytes to be added to the transmit FIFO (if any) to txBuf and set txCnt to the number of copied bytes.

Upon return rxCnt will be set to the number of received bytes placed in rxBuf.

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.

The returned function value is the status of the transfer (see below).

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

The most significant word of the returned status contains the number of bytes actually copied from txBuf to the BSC transmit FIFO (may be less than requested if the FIFO already contained untransmitted data).

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 returned 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

The following example shows how to configure the BSC peripheral as an I2C slave with address 0x13 and send four bytes.

Example

```
bsc_xfer_t xfer;

xfer.control = (0x13<<16) | 0x305;

memcpy(xfer.txBuf, "ABCD", 4);
xfer.txCnt = 4;

status = bscXfer(&xfer);

if (status >= 0)
{
   // process transfer
}
```

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.

int bbSPIOpen(unsigned CS, unsigned MISO, unsigned MOSI, unsigned SCLK, unsigned baud, unsigned spiFlags)

This function selects a set of GPIO for bit banging SPI with a specified baud rate and mode.

```
      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  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
```

p is 0 if CS is active low (default) and 1 for active high.

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

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

The other bits in flags 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

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

int bbSPIClose(unsigned CS)

This function stops bit banging SPI on a set of GPIO opened with bbSPIOpen.

```
CS: 0-31, the CS GPIO used in a prior call to bbSPIOpen
```

Returns 0 if OK, otherwise PI_BAD_USER_GPIO, or PI_NOT_SPI_GPIO.

int bbSPIXfer(unsigned CS, char *inBuf, char *outBuf, unsigned count)

This function executes a bit banged SPI transfer.

```
    CS: 0-31 (as used in a prior call to bbSPIOpen)
 inBuf: pointer to buffer to hold data to be sent
outBuf: pointer to buffer to hold returned data
 count: size of data transfer
```

Returns >= 0 if OK (the number of bytes read), otherwise PI_BAD_USER_GPIO, PI_NOT_SPI_GPIO or PI_BAD_POINTER.

Example

```
// gcc -Wall -pthread -o bbSPIx_test bbSPIx_test.c -lpigpio
// sudo ./bbSPIx_test

#include <stdio.h>

#include "pigpio.h"

#define CE0 5
#define CE1 6
#define MISO 13
#define MOSI 19
#define SCLK 12
```

```
int main(int argc, char *argv[])
{
   int i, count, set_val, read_val;
   unsigned char inBuf[3];
   char cmd1[] = {0, 0};
   char cmd2[] = {12, 0};
   char cmd3[] = {1, 128, 0};

   if (gpioInitialise() < 0)
   {
      fprintf(stderr, "pigpio initialisation failed.\n");
      return 1;
   }

   bbSPIOpen(CE0, MISO, MOSI, SCLK, 10000, 0); // MCP4251 DAC
   bbSPIOpen(CE1, MISO, MOSI, SCLK, 20000, 3); // MCP3008 ADC

   for (i=0; i<256; i++)
   {
      cmd1[1] = i;

      count = bbSPIXfer(CE0, cmd1, (char *)inBuf, 2); // > DAC

      if (count == 2)
      {
         count = bbSPIXfer(CE0, cmd2, (char *)inBuf, 2); // < DAC

         if (count == 2)
         {
            set_val = inBuf[1];

            count = bbSPIXfer(CE1, cmd3, (char *)inBuf, 3); // < ADC

            if (count == 3)
            {
               read_val = ((inBuf[1]&3)<<8) | inBuf[2];
               printf("%d %d\n", set_val, read_val);
            }
         }
      }
   }

   bbSPIClose(CE0);
   bbSPIClose(CE1);

   gpioTerminate();

   return 0;
}
```

int spiOpen(unsigned spiChan, unsigned baud, unsigned spiFlags)

This function 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

```
 spiChan: 0-1 (0-2 for the auxiliary SPI)
    baud: 32K-125M (values above 30M are unlikely to work)
spiFlags: see below
```

Returns a handle (>=0) if OK, otherwise PI_BAD_SPI_CHANNEL, PI_BAD_SPI_SPEED, PI_BAD_FLAGS, PI_NO_AUX_SPI, or PI_SPI_OPEN_FAILED.

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
 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 spiRead, spiWrite, and spiXfer 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 word.
For bits 9-16 there will be two bytes per word.
For bits 17-32 there will be four bytes per word.

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

E.g. to transfer 32 11-bit words buf should contain 64 bytes and count should be 64.

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.

int spiClose(unsigned handle)

This functions closes the SPI device identified by the handle.

```
handle: >=0, as returned by a call to spiOpen
```

Returns 0 if OK, otherwise PI_BAD_HANDLE.

int spiRead(unsigned handle, char *buf, unsigned count)

This function reads count bytes of data from the SPI device associated with the handle.

```
handle: >=0, as returned by a call to spiOpen
   buf: an array to receive the read data bytes
 count: the number of bytes to read
```

Returns the number of bytes transferred if OK, otherwise PI_BAD_HANDLE, PI_BAD_SPI_COUNT, or PI_SPI_XFER_FAILED.

int spiWrite(unsigned handle, char *buf, unsigned count)

This function writes count bytes of data from buf to the SPI device associated with the handle.

```
handle: >=0, as returned by a call to spiOpen
   buf: the data bytes to write
 count: the number of bytes to write
```

Returns the number of bytes transferred if OK, otherwise PI_BAD_HANDLE, PI_BAD_SPI_COUNT, or PI_SPI_XFER_FAILED.

int spiXfer(unsigned handle, char *txBuf, char *rxBuf, unsigned count)

This function transfers count bytes of data from txBuf to the SPI device associated with the handle. Simultaneously count bytes of data are read from the device and placed in rxBuf.

```
handle: >=0, as returned by a call to spiOpen
 txBuf: the data bytes to write
 rxBuf: the received data bytes
 count: the number of bytes to transfer
```

Returns the number of bytes transferred if OK, otherwise PI_BAD_HANDLE, PI_BAD_SPI_COUNT, or PI_SPI_XFER_FAILED.

int serOpen(char *sertty, unsigned baud, unsigned serFlags)

This function opens a serial device at a specified baud rate and with specified flags. The device name must start with /dev/tty or /dev/serial.

```
  sertty: the serial device to open
    baud: the baud rate in bits per second, see below
serFlags: 0
```

Returns a handle (>=0) if OK, otherwise PI_NO_HANDLE, or PI_SER_OPEN_FAILED.

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.

No flags are currently defined. This parameter should be set to zero.

int serClose(unsigned handle)

This function closes the serial device associated with handle.

```
handle: >=0, as returned by a call to serOpen
```

Returns 0 if OK, otherwise PI_BAD_HANDLE.

int serWriteByte(unsigned handle, unsigned bVal)

This function writes bVal to the serial port associated with handle.

```
handle: >=0, as returned by a call to serOpen
```

Returns 0 if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_SER_WRITE_FAILED.

int serReadByte(unsigned handle)

This function reads a byte from the serial port associated with handle.

```
handle: >=0, as returned by a call to serOpen
```

Returns the read byte (>=0) if OK, otherwise PI_BAD_HANDLE, PI_SER_READ_NO_DATA, or PI_SER_READ_FAILED.

If no data is ready PI_SER_READ_NO_DATA is returned.

int serWrite(unsigned handle, char *buf, unsigned count)

This function writes count bytes from buf to the the serial port associated with handle.

```
handle: >=0, as returned by a call to serOpen
   buf: the array of bytes to write
 count: the number of bytes to write
```

Returns 0 if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_SER_WRITE_FAILED.

int serRead(unsigned handle, char *buf, unsigned count)

This function reads up count bytes from the the serial port associated with handle and writes them to buf.

```
handle: >=0, as returned by a call to serOpen
   buf: an array to receive the read data
 count: the maximum number of bytes to read
```

Returns the number of bytes read (>0=) if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, or PI_SER_READ_NO_DATA.

If no data is ready zero is returned.

int serDataAvailable(unsigned handle)

This function returns the number of bytes available to be read from the device associated with handle.

```
handle: >=0, as returned by a call to serOpen
```

Returns the number of bytes of data available (>=0) if OK, otherwise PI_BAD_HANDLE.

int gpioTrigger(unsigned user_gpio, unsigned pulseLen, unsigned level)

This function sends a trigger pulse to a GPIO. The GPIO is set to level for pulseLen microseconds and then reset to not level.

```
user_gpio: 0-31
 pulseLen: 1-100
    level: 0,1
```

Returns 0 if OK, otherwise PI_BAD_USER_GPIO, PI_BAD_LEVEL, or PI_BAD_PULSELEN.

int gpioSetWatchdog(unsigned user_gpio, unsigned timeout)

Sets a watchdog for a GPIO.

```
user_gpio: 0-31
  timeout: 0-60000
```

Returns 0 if OK, otherwise PI_BAD_USER_GPIO or PI_BAD_WDOG_TIMEOUT.

The watchdog is nominally in milliseconds.

One watchdog may be registered per GPIO.

The watchdog may be cancelled by setting timeout to 0.

Until cancelled a timeout will be reported every timeout milliseconds after the last GPIO activity.

In particular:

1) any registered alert function for the GPIO will be called with the level set to PI_TIMEOUT.

2) any notification for the GPIO will have a report written to the fifo with the flags set to indicate a watchdog timeout.

Example

```
void aFunction(int gpio, int level, uint32_t tick)
{
   printf("GPIO %d became %d at %d", gpio, level, tick);
}

// call aFunction whenever GPIO 4 changes state
gpioSetAlertFunc(4, aFunction);

//  or approximately every 5 millis
gpioSetWatchdog(4, 5);
```

int gpioNoiseFilter(unsigned user_gpio, unsigned steady, unsigned 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.

```
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 gpioSetAlertFunc, gpioSetAlertFuncEx, gpioSetGetSamplesFunc, and gpioSetGetSamplesFuncEx.

It does not affect interrupts set up with gpioSetISRFunc, gpioSetISRFuncEx, or levels read by gpioRead, gpioRead_Bits_0_31, or gpioRead_Bits_32_53.

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

int gpioGlitchFilter(unsigned user_gpio, unsigned 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.

```
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 gpioSetAlertFunc, gpioSetAlertFuncEx, gpioSetGetSamplesFunc, and gpioSetGetSamplesFuncEx.

It does not affect interrupts set up with gpioSetISRFunc, gpioSetISRFuncEx, or levels read by gpioRead, gpioRead_Bits_0_31, or gpioRead_Bits_32_53.

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

int gpioSetGetSamplesFunc(gpioGetSamplesFunc_t f, uint32_t bits)

Registers a function to be called (a callback) every millisecond with the latest GPIO samples.

```
   f: the function to call
bits: the GPIO of interest
```

Returns 0 if OK.

The function is passed a pointer to the samples (an array of gpioSample_t), and the number of samples.

Only one function can be registered.

The callback may be cancelled by passing NULL as the function.

The samples returned will be the union of bits, plus any active alerts, plus any active notifications.

e.g. if there are alerts for GPIO 7, 8, and 9, notifications for GPIO 8, 10, 23, 24, and bits is (1<<23)|(1<<17) then samples for GPIO 7, 8, 9, 10, 17, 23, and 24 will be reported.

int gpioSetGetSamplesFuncEx(gpioGetSamplesFuncEx_t f, uint32_t bits, void *userdata)

Registers a function to be called (a callback) every millisecond with the latest GPIO samples.

```
       f: the function to call
    bits: the GPIO of interest
```

```
userdata: a pointer to arbitrary user data
```

Returns 0 if OK.

The function is passed a pointer to the samples (an array of gpioSample_t), the number of samples, and the userdata pointer.

Only one of gpioGetSamplesFunc or gpioGetSamplesFuncEx can be registered.

See gpioSetGetSamplesFunc for further details.

int gpioSetTimerFunc(unsigned timer, unsigned millis, gpioTimerFunc_t f)

Registers a function to be called (a callback) every millis milliseconds.

```
 timer: 0-9
millis: 10-60000
     f: the function to call
```

Returns 0 if OK, otherwise PI_BAD_TIMER, PI_BAD_MS, or PI_TIMER_FAILED.

10 timers are supported numbered 0 to 9.

One function may be registered per timer.

The timer may be cancelled by passing NULL as the function.

Example

```
void bFunction(void)
{
   printf("two seconds have elapsed");
}

// call bFunction every 2000 milliseconds
gpioSetTimerFunc(0, 2000, bFunction);
```

int gpioSetTimerFuncEx(unsigned timer, unsigned millis, gpioTimerFuncEx_t f, void *userdata)

Registers a function to be called (a callback) every millis milliseconds.

```
   timer: 0-9.
  millis: 10-60000
       f: the function to call
userdata: a pointer to arbitrary user data
```

Returns 0 if OK, otherwise PI_BAD_TIMER, PI_BAD_MS, or PI_TIMER_FAILED.

The function is passed the userdata pointer.

Only one of gpioSetTimerFunc or gpioSetTimerFuncEx can be registered per timer.

See gpioSetTimerFunc for further details.

pthread_t *gpioStartThread(gpioThreadFunc_t f, void *userdata)

Starts a new thread of execution with f as the main routine.

```
       f: the main function for the new thread
userdata: a pointer to arbitrary user data
```

Returns a pointer to pthread_t if OK, otherwise NULL.

The function is passed the single argument arg.

The thread can be cancelled by passing the pointer to pthread_t to gpioStopThread.

Example

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

void *myfunc(void *arg)
{
   while (1)
   {
      printf("%s", arg);
      sleep(1);
   }
}

int main(int argc, char *argv[])
{
   pthread_t *p1, *p2, *p3;

   if (gpioInitialise() < 0) return 1;

   p1 = gpioStartThread(myfunc, "thread 1"); sleep(3);

   p2 = gpioStartThread(myfunc, "thread 2"); sleep(3);

   p3 = gpioStartThread(myfunc, "thread 3"); sleep(3);

   gpioStopThread(p3); sleep(3);

   gpioStopThread(p2); sleep(3);

   gpioStopThread(p1); sleep(3);

   gpioTerminate();
}
```

void gpioStopThread(pthread_t *pth)

Cancels the thread pointed at by pth.

```
pth: a thread pointer returned by gpioStartThread
```

No value is returned.

The thread to be stopped should have been started with gpioStartThread.

int gpioStoreScript(char *script)

This function stores a null terminated script for later execution.

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

```
script: the text of the script
```

The function returns a script id if the script is valid, otherwise PI_BAD_SCRIPT.

int gpioRunScript(unsigned script_id, unsigned numPar, uint32_t *param)

This function runs a stored script.

```
script_id: >=0, as returned by gpioStoreScript
   numPar: 0-10, the number of parameters
    param: an array of parameters
```

The function returns 0 if OK, otherwise PI_BAD_SCRIPT_ID, or PI_TOO_MANY_PARAM.

param is an array of up to 10 parameters which may be referenced in the script as p0 to p9.

int gpioUpdateScript(unsigned script_id, unsigned numPar, uint32_t *param)

This function sets the parameters of a script. The script may or may not be running. The first numPar parameters of the script are overwritten with the new values.

```
script_id: >=0, as returned by gpioStoreScript
   numPar: 0-10, the number of parameters
    param: an array of parameters
```

The function returns 0 if OK, otherwise PI_BAD_SCRIPT_ID, or PI_TOO_MANY_PARAM.

param is an array of up to 10 parameters which may be referenced in the script as p0 to p9.

int gpioScriptStatus(unsigned script_id, uint32_t *param)

This function returns the run status of a stored script as well as the current values of parameters 0 to 9.

```
script_id: >=0, as returned by gpioStoreScript
    param: an array to hold the returned 10 parameters
```

The function returns greater than or equal to 0 if OK, otherwise PI_BAD_SCRIPT_ID.

The run status may be

```
PI_SCRIPT_INITING
PI_SCRIPT_HALTED
PI_SCRIPT_RUNNING
PI_SCRIPT_WAITING
PI_SCRIPT_FAILED
```

The current value of script parameters 0 to 9 are returned in param.

int gpioStopScript(unsigned script_id)

This function stops a running script.

```
script_id: >=0, as returned by gpioStoreScript
```

The function returns 0 if OK, otherwise PI_BAD_SCRIPT_ID.

int gpioDeleteScript(unsigned script_id)

This function deletes a stored script.

```
script_id: >=0, as returned by gpioStoreScript
```

The function returns 0 if OK, otherwise PI_BAD_SCRIPT_ID.

int gpioSetSignalFunc(unsigned signum, gpioSignalFunc_t f)

Registers a function to be called (a callback) when a signal occurs.

```
signum: 0-63
     f: the callback function
```

Returns 0 if OK, otherwise PI_BAD_SIGNUM.

The function is passed the signal number.

One function may be registered per signal.

The callback may be cancelled by passing NULL.

By default all signals are treated as fatal and cause the library to call gpioTerminate and then exit.

int gpioSetSignalFuncEx(unsigned signum, gpioSignalFuncEx_t f, void *userdata)

Registers a function to be called (a callback) when a signal occurs.

```
  signum: 0-63
       f: the callback function
userdata: a pointer to arbitrary user data
```

Returns 0 if OK, otherwise PI_BAD_SIGNUM.

The function is passed the signal number and the userdata pointer.

Only one of gpioSetSignalFunc or gpioSetSignalFuncEx can be registered per signal.

See gpioSetSignalFunc for further details.

uint32_t gpioRead_Bits_0_31(void)

Returns the current level of GPIO 0-31.

uint32_t gpioRead_Bits_32_53(void)

Returns the current level of GPIO 32-53.

int gpioWrite_Bits_0_31_Clear(uint32_t bits)

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

```
bits: a bit mask of GPIO to clear
```

Returns 0 if OK.

Example

```
// To clear (set to 0) GPIO 4, 7, and 15
gpioWrite_Bits_0_31_Clear( (1<<4) | (1<<7) | (1<<15) );
```

int gpioWrite_Bits_32_53_Clear(uint32_t bits)

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

```
bits: a bit mask of GPIO to clear
```

Returns 0 if OK.

int gpioWrite_Bits_0_31_Set(uint32_t bits)

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

```
bits: a bit mask of GPIO to set
```

Returns 0 if OK.

int gpioWrite_Bits_32_53_Set(uint32_t bits)

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

```
bits: a bit mask of GPIO to set
```

Returns 0 if OK.

Example

```
// To set (set to 1) GPIO 32, 40, and 53
gpioWrite_Bits_32_53_Set((1<<(32-32)) | (1<<(40-32)) | (1<<(53-32)));
```

int gpioHardwareClock(unsigned gpio, unsigned clkfreq)

Starts a hardware clock on a GPIO at the specified frequency. Frequencies above 30MHz are unlikely to work.

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

Returns 0 if OK, otherwise 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.

int gpioHardwarePWM(unsigned gpio, unsigned PWMfreq, unsigned 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 gpioWaveTxSend, or gpioWaveChain 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 by a call to gpioCfgClock.

```
   gpio: see description
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_BAD_GPIO, PI_NOT_HPWM_GPIO, PI_BAD_HPWM_DUTY, PI_BAD_HPWM_FREQ, or PI_HPWM_ILLEGAL.

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.

int gpioTime(unsigned timetype, int *seconds, int *micros)

Updates the seconds and micros variables with the current time.

```
timetype: 0 (relative), 1 (absolute)
 seconds: a pointer to an int to hold seconds
  micros: a pointer to an int to hold microseconds
```

Returns 0 if OK, otherwise PI_BAD_TIMETYPE.

If timetype is PI_TIME_ABSOLUTE updates seconds and micros with the number of seconds and microseconds since the epoch (1st January 1970).

If timetype is PI_TIME_RELATIVE updates seconds and micros with the number of seconds and microseconds since the library was initialised.

Example

```
int secs, mics;

// print the number of seconds since the library was started
gpioTime(PI_TIME_RELATIVE, &secs, &mics);
printf("library started %d.%03d seconds ago", secs, mics/1000);
```

int gpioSleep(unsigned timetype, int seconds, int micros)

Sleeps for the number of seconds and microseconds specified by seconds and micros.

```
timetype: 0 (relative), 1 (absolute)
 seconds: seconds to sleep
  micros: microseconds to sleep
```

Returns 0 if OK, otherwise PI_BAD_TIMETYPE, PI_BAD_SECONDS, or PI_BAD_MICROS.

If timetype is PI_TIME_ABSOLUTE the sleep ends when the number of seconds and microseconds since the epoch (1st January 1970) has elapsed. System clock changes are taken into account.

If timetype is PI_TIME_RELATIVE the sleep is for the specified number of seconds and microseconds. System clock changes do not effect the sleep length.

For short delays (say, 50 microseonds or less) use gpioDelay.

Example

```
gpioSleep(PI_TIME_RELATIVE, 2, 500000); // sleep for 2.5 seconds

gpioSleep(PI_TIME_RELATIVE, 0, 100000); // sleep for 0.1 seconds

gpioSleep(PI_TIME_RELATIVE, 60, 0);     // sleep for one minute
```

uint32_t gpioDelay(uint32_t micros)

Delays for at least the number of microseconds specified by micros.

```
micros: the number of microseconds to sleep
```

Returns the actual length of the delay in microseconds.

Delays of 100 microseconds or less use busy waits.

uint32_t gpioTick(void)

Returns the current system tick.

Tick is the number of microseconds since system boot.

As tick is an unsigned 32 bit quantity it wraps around after 2^32 microseconds, which is approximately 1 hour 12 minutes.

You don't need to worry about the wrap around as long as you take a tick (uint32_t) from another tick, i.e. the following code will always provide the correct difference.

Example

```
uint32_t startTick, endTick;
int diffTick;
```

```
startTick = gpioTick();

// do some processing

endTick = gpioTick();

diffTick = endTick - startTick;

printf("some processing took %d microseconds", diffTick);
```

unsigned gpioHardwareRevision(void)

Returns the hardware revision.

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

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.

for "Revision : 0002" the function returns 2.
for "Revision : 000f" the function returns 15.
for "Revision : 000g" the function returns 0.

unsigned gpioVersion(void)

Returns the pigpio version.

int gpioGetPad(unsigned pad)

This function returns the pad drive strength in mA.

```
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 = gpioGetPad(1); // get pad 1 strength
```

int gpioSetPad(unsigned pad, unsigned padStrength)

This function sets the pad drive strength in mA.

```
        pad: 0-2, the pad to set
padStrength: 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

```
gpioSetPad(0, 16); // set pad 0 strength to 16 mA
```

int eventMonitor(unsigned handle, uint32_t bits)

This function selects the events to be reported on a previously opened handle.

```
handle: >=0, as returned by gpioNotifyOpen
  bits: a bit mask indicating the events of interest
```

Returns 0 if OK, otherwise PI_BAD_HANDLE.

A report is sent each time an event is triggered providing the corresponding bit in bits is set.

See gpioNotifyBegin for the notification format.

Example

```
// Start reporting events 3, 6, and 7.

//  bit      76543210
// (0xC8 = 0b11001000)

eventMonitor(h, 0xC8);
```

int eventSetFunc(unsigned event, eventFunc_t f)

Registers a function to be called (a callback) when the specified event occurs.

```
event: 0-31
    f: the callback function
```

Returns 0 if OK, otherwise PI_BAD_EVENT_ID.

One function may be registered per event.

The function is passed the event, and the tick.

The callback may be cancelled by passing NULL as the function.

int eventSetFuncEx(unsigned event, eventFuncEx_t f, void *userdata)

Registers a function to be called (a callback) when the specified event occurs.

```
   event: 0-31
       f: the callback function
userdata: pointer to arbitrary user data
```

Returns 0 if OK, otherwise PI_BAD_EVENT_ID.

One function may be registered per event.

The function is passed the event, the tick, and the ueserdata pointer.

The callback may be cancelled by passing NULL as the function.

Only one of eventSetFunc or eventSetFuncEx can be registered per event.

int eventTrigger(unsigned event)

This function signals the occurrence of an event.

```
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 eventSetFunc) will be informed by a callback.

One event, PI_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.

int shell(char *scriptName, char *scriptString)

This function uses the system call to execute a shell script with the given string as its parameter.

```
  scriptName: the name of the script, only alphanumeric characters,
              '-' and '_' are allowed in the name
scriptString: the string to pass to the script
```

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

scriptName 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 = shell("scr1", "hello world");

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

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

int fileOpen(char *file, unsigned mode)

This function returns a handle to a file opened in a specified mode.

```
file: the file to open
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.

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.

Macro	Value	Meaning
PI_FILE_READ	1	open file for reading
PI_FILE_WRITE	2	open file for writing
PI_FILE_RW	3	open file for reading and writing

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

Macro	Value	Meaning
PI_FILE_APPEND	4	Writes append data to the end of the file
PI_FILE_CREATE	8	The file is created if it doesn't exist
PI_FILE_TRUNC	16	The file is truncated

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

Example

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

int main(int argc, char *argv[])
{
   int handle, c;
   char buf[60000];
   if (gpioInitialise() < 0) return 1;

   // assumes /opt/pigpio/access contains the following line
   // /ram/*.c r

   handle = fileOpen("/ram/pigpio.c", PI_FILE_READ);

   if (handle >= 0)
   {
      while ((c=fileRead(handle, buf, sizeof(buf)-1)))
      {
         buf[c] = 0;
         printf("%s", buf);
      }

      fileClose(handle);
   }

   gpioTerminate();
}
```

int fileClose(unsigned handle)

This function closes the file associated with handle.

```
handle: >=0, as returned by a call to fileOpen
```

Returns 0 if OK, otherwise PI_BAD_HANDLE.

Example

```
fileClose(h);
```

int fileWrite(unsigned handle, char *buf, unsigned count)

This function writes count bytes from buf to the the file associated with handle.

```
handle: >=0, as returned by a call to fileOpen
   buf: the array of bytes to write
 count: the number of bytes to write
```

Returns 0 if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, PI_FILE_NOT_WOPEN, or PI_BAD_FILE_WRITE.

Example

```
status = fileWrite(h, buf, count);
if (status == 0)
{
   // okay
}
else
{
   // error
}
```

int fileRead(unsigned handle, char *buf, unsigned count)

This function reads up to count bytes from the the file associated with handle and writes them to buf.

```
handle: >=0, as returned by a call to fileOpen
   buf: an array to receive the read data
 count: the maximum number of bytes to read
```

Returns the number of bytes read (>=0) if OK, otherwise PI_BAD_HANDLE, PI_BAD_PARAM, PI_FILE_NOT_ROPEN, or PI_BAD_FILE_WRITE.

Example

```
if (fileRead(h, buf, sizeof(buf)) > 0)
{
   // process read data
}
```

int fileSeek(unsigned handle, int32_t seekOffset, int seekFrom)

This function seeks to a position within the file associated with handle.

```
    handle: >=0, as returned by a call to fileOpen
seekOffset: the number of bytes to move.  Positive offsets
            move forward, negative offsets backwards.
  seekFrom: one of PI_FROM_START (0), PI_FROM_CURRENT (1),
            or PI_FROM_END (2)
```

Returns the new byte position within the file (>=0) if OK, otherwise PI_BAD_HANDLE, or PI_BAD_FILE_SEEK.

Example

```
fileSeek(0, 20, PI_FROM_START); // Seek to start plus 20

size = fileSeek(0, 0, PI_FROM_END); // Seek to end, return size

pos = fileSeek(0, 0, PI_FROM_CURRENT); // Return current position
```

int fileList(char *fpat, char *buf, unsigned count)

This function returns a list of files which match a pattern. The pattern may contain wildcards.

```
 fpat: file pattern to match
  buf: an array to receive the matching file names
count: the maximum number of bytes to read
```

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 fileOpen.

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

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

int main(int argc, char *argv[])
{
   int c;
   char buf[1000];

   if (gpioInitialise() < 0) return 1;

   // assumes /opt/pigpio/access contains the following line
   // /ram/*.c r

   c = fileList("/ram/p*.c", buf, sizeof(buf));

   if (c >= 0)
   {
      // terminate string
      buf[c] = 0;
      printf("%s", buf);
   }

   gpioTerminate();
}
```

int gpioCfgBufferSize(unsigned cfgMillis)

Configures pigpio to buffer cfgMillis milliseconds of GPIO samples.

This function is only effective if called before gpioInitialise.

```
cfgMillis: 100-10000
```

The default setting is 120 milliseconds.

The intention is to allow for bursts of data and protection against other processes hogging cpu time.

I haven't seen a process locked out for more than 100 milliseconds.

Making the buffer bigger uses a LOT of memory at the more frequent sampling rates as shown in the following table in MBs.

		buffer milliseconds						
		120	250	500	1sec	2sec	4sec	8sec
	1	16	31	55	107	---	---	---
	2	10	18	31	55	107	---	---
sample	4	8	12	18	31	55	107	---
rate	5	8	10	14	24	45	87	---
(us)	8	6	8	12	18	31	55	107
	10	6	8	10	14	24	45	87

int gpioCfgClock(unsigned cfgMicros, unsigned cfgPeripheral, unsigned cfgSource)

Configures pigpio to use a particular sample rate timed by a specified peripheral.

This function is only effective if called before gpioInitialise.

```
    cfgMicros: 1, 2, 4, 5, 8, 10
cfgPeripheral: 0 (PWM), 1 (PCM)
    cfgSource: deprecated, value is ignored
```

The timings are provided by the specified peripheral (PWM or PCM).

The default setting is 5 microseconds using the PCM peripheral.

The approximate CPU percentage used for each sample rate is:

```
sample  cpu
 rate    %

  1     25
  2     16
  4     11
  5     10
  8     15
 10     14
```

A sample rate of 5 microseconds seeems to be the sweet spot.

int gpioCfgDMAchannel(unsigned DMAchannel)

Configures pigpio to use the specified DMA channel.

This function is only effective if called before gpioInitialise.

```
DMAchannel: 0-14
```

The default setting is to use channel 14.

int gpioCfgDMAchannels(unsigned primaryChannel, unsigned secondaryChannel)

Configures pigpio to use the specified DMA channels.

This function is only effective if called before gpioInitialise.

```
  primaryChannel: 0-14
secondaryChannel: 0-14
```

The default setting depends on whether the Pi has a BCM2711 chip or not (currently only the Pi4B has a BCM2711).

The default setting for a non-BCM2711 is to use channel 14 for the primary channel and channel 6 for the secondary channel.

The default setting for a BCM2711 is to use channel 7 for the primary channel and channel 6 for the secondary channel.

The secondary channel is only used for the transmission of waves.

If possible use one of channels 0 to 6 for the secondary channel (a full channel).

A full channel only requires one DMA control block regardless of the length of a pulse delay. Channels 7 to 14 (lite channels) require one DMA control block for each 16383 microseconds of delay. I.e. a 10 second pulse delay requires one control block on a full channel and 611 control blocks on a lite channel.

int gpioCfgPermissions(uint64_t updateMask)

Configures pigpio to restrict GPIO updates via the socket or pipe interfaces to the GPIO specified by the mask. Programs directly calling the pigpio library (i.e. linked with -lpigpio are not affected). A GPIO update is a write to a GPIO or a GPIO mode change or any function which would force such an action.

This function is only effective if called before gpioInitialise.

```
updateMask: bit (1<<n) is set for each GPIO n which may be updated
```

The default setting depends upon the Pi model. The user GPIO are added to the mask.

If the board revision is not recognised then GPIO 2-27 are allowed.

Unknown board	PI_DEFAULT_UPDATE_MASK_UNKNOWN	0x0FFFFFFC
Type 1 board	PI_DEFAULT_UPDATE_MASK_B1	0x03E6CF93
Type 2 board	PI_DEFAULT_UPDATE_MASK_A_B2	0xFBC6CF9C
Type 3 board	PI_DEFAULT_UPDATE_MASK_R3	0x0FFFFFFC

int gpioCfgSocketPort(unsigned port)

Configures pigpio to use the specified socket port.

This function is only effective if called before gpioInitialise.

```
port: 1024-32000
```

The default setting is to use port 8888.

int gpioCfgInterfaces(unsigned ifFlags)

Configures pigpio support of the fifo and socket interfaces.

This function is only effective if called before gpioInitialise.

```
ifFlags: 0-7
```

The default setting (0) is that both interfaces are enabled.

Or in PI_DISABLE_FIFO_IF to disable the pipe interface.

Or in PI_DISABLE_SOCK_IF to disable the socket interface.

Or in PI_LOCALHOST_SOCK_IF to disable remote socket access (this means that the socket interface is only usable from the local Pi).

int gpioCfgMemAlloc(unsigned memAllocMode)

Selects the method of DMA memory allocation.

This function is only effective if called before gpioInitialise.

```
memAllocMode: 0-2
```

There are two methods of DMA memory allocation. The original method uses the /proc/self/pagemap file to allocate bus memory. The new method uses the mailbox property interface to allocate bus memory.

Auto will use the mailbox method unless a larger than default buffer size is requested with gpioCfgBufferSize.

int gpioCfgNetAddr(int numSockAddr, uint32_t *sockAddr)

Sets the network addresses which are allowed to talk over the socket interface.

This function is only effective if called before gpioInitialise.

```
numSockAddr: 0-256 (0 means all addresses allowed)
   sockAddr: an array of permitted network addresses.
```

uint32_t gpioCfgGetInternals(void)

This function returns the current library internal configuration settings.

int gpioCfgSetInternals(uint32_t cfgVal)

This function sets the current library internal configuration settings.

```
cfgVal: see source code
```

int gpioCustom1(unsigned arg1, unsigned arg2, char *argx, unsigned argc)

This function is available for user customisation.

It returns a single integer value.

```
arg1: >=0
arg2: >=0
argx: extra (byte) arguments
argc: number of extra arguments
```

Returns >= 0 if OK, less than 0 indicates a user defined error.

int gpioCustom2(unsigned arg1, char *argx, unsigned argc, char *retBuf, unsigned retMax)

This function is available for user customisation.

It differs from gpioCustom1 in that it returns an array of bytes rather than just an integer.

The returned value is an integer indicating the number of returned bytes.
```
  arg1: >=0
  argx: extra (byte) arguments
  argc: number of extra arguments
retBuf: buffer for returned bytes
retMax: maximum number of bytes to return
```

Returns >= 0 if OK, less than 0 indicates a user defined error.

The number of returned bytes must be retMax or less.

int rawWaveAddSPI(rawSPI_t *spi, unsigned offset, unsigned spiSS, char *buf, unsigned spiTxBits, unsigned spiBitFirst, unsigned spiBitLast, unsigned spiBits)

This function adds a waveform representing SPI data to the existing waveform (if any).

```
        spi: a pointer to a spi object
     offset: microseconds from the start of the waveform
      spiSS: the slave select GPIO
        buf: the bits to transmit, most significant bit first
  spiTxBits: the number of bits to write
spiBitFirst: the first bit to read
 spiBitLast: the last bit to read
    spiBits: the number of bits to transfer
```

Returns the new total number of pulses in the current waveform if OK, otherwise PI_BAD_USER_GPIO, PI_BAD_SER_OFFSET, or PI_TOO_MANY_PULSES.

Not intended for general use.

int rawWaveAddGeneric(unsigned numPulses, rawWave_t *pulses)

This function adds a number of pulses to the current waveform.

```
numPulses: the number of pulses
   pulses: the array containing the pulses
```

Returns the new total number of pulses in the current waveform if OK, otherwise PI_TOO_MANY_PULSES.

The advantage of this function over gpioWaveAddGeneric is that it allows the setting of the flags field.

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 of a delay.

Not intended for general use.

unsigned rawWaveCB(void)

Returns the number of the cb being currently output.

Not intended for general use.

rawCbs_t *rawWaveCBAdr(int cbNum)

Return the (Linux) address of contol block cbNum.

```
cbNum: the cb of interest
```

Not intended for general use.

uint32_t rawWaveGetOOL(int pos)

Gets the OOL parameter stored at pos.

```
pos: the position of interest.
```

Not intended for general use.

void rawWaveSetOOL(int pos, uint32_t lVal)

Sets the OOL parameter stored at pos to value.

```
 pos: the position of interest
lVal: the value to write
```

Not intended for general use.

uint32_t rawWaveGetOut(int pos)

Gets the wave output parameter stored at pos.

DEPRECATED: use rawWaveGetOOL instead.

```
pos: the position of interest.
```

Not intended for general use.

void rawWaveSetOut(int pos, uint32_t lVal)

Sets the wave output parameter stored at pos to value.

DEPRECATED: use rawWaveSetOOL instead.

```
 pos: the position of interest
lVal: the value to write
```

Not intended for general use.

uint32_t rawWaveGetIn(int pos)

Gets the wave input value parameter stored at pos.

DEPRECATED: use rawWaveGetOOL instead.

```
pos: the position of interest
```

Not intended for general use.

void rawWaveSetIn(int pos, uint32_t lVal)

Sets the wave input value stored at pos to value.

DEPRECATED: use rawWaveSetOOL instead.

```
 pos: the position of interest
lVal: the value to write
```

Not intended for general use.

rawWaveInfo_t rawWaveInfo(int wave_id)

Gets details about the wave with id wave_id.

```
wave_id: the wave of interest
```

Not intended for general use.

int getBitInBytes(int bitPos, char *buf, int numBits)

Returns the value of the bit bitPos bits from the start of buf. Returns 0 if bitPos is greater than or equal to numBits.

```
 bitPos: bit index from the start of buf
    buf: array of bits
numBits: number of valid bits in buf
```

void putBitInBytes(int bitPos, char *buf, int bit)

Sets the bit bitPos bits from the start of buf to bit.

```
bitPos: bit index from the start of buf
   buf: array of bits
   bit: 0-1, value to set
```

double time_time(void)

Return the current time in seconds since the Epoch.

void time_sleep(double seconds)

Delay execution for a given number of seconds

```
seconds: the number of seconds to sleep
```

void rawDumpWave(void)

Used to print a readable version of the current waveform to stderr.

Not intended for general use.

void rawDumpScript(unsigned script_id)

Used to print a readable version of a script to stderr.

```
script_id: >=0, a script_id returned by gpioStoreScript
```

Not intended for general use.

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.

argc

The count of bytes passed to a user customised function.

*argx

A pointer to 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.

bit

A value of 0 or 1.

bitPos

A bit position within a byte or word. The least significant bit is position 0.

bits

A value used to select GPIO. If bit n of bits is set then GPIO n is selected.

A convenient way to set bit n is to or in (1<<n).

e.g. to select bits 5, 9, 23 you could use (1<<5) | (1<<9) | (1<<23).

*bsc_xfer

A pointer to a bsc_xfer_t object used to control a BSC transfer.

bsc_xfer_t

```
typedef struct
{
   uint32_t control;          // Write
   int rxCnt;                 // Read only
   char rxBuf[BSC_FIFO_SIZE]; // Read only
   int txCnt;                 // Write
   char txBuf[BSC_FIFO_SIZE]; // Write
} bsc_xfer_t;
```

*buf

A buffer to hold data being sent or being received.

bufSize

The size in bytes of a buffer.

bVal: 0-255 (Hex 0x0-0xFF, Octal 0-0377)

An 8-bit byte value.

cbNum

A number identifying a DMA contol block.

cfgMicros

The GPIO sample rate in microseconds. The default is 5us, or 200 thousand samples per second.

cfgMillis: 100-10000

The size of the sample buffer in milliseconds. Generally this should be left at the default of 120ms. If you expect intense bursts of signals it might be necessary to increase the buffer size.

cfgPeripheral

One of the PWM or PCM peripherals used to pace DMA transfers for timing purposes.

cfgSource

Deprecated.

cfgVal

A number specifying the value of a configuration item. See cfgWhat.

cfgWhat

A number specifying a configuration item.

562484977: print enhanced statistics at termination.
984762879: set the initial debug level.

char

A single character, an 8 bit quantity able to store 0-255.

clkfreq: 4689-250M (13184-375M for the BCM2711)

The hardware clock frequency.

```
PI_HW_CLK_MIN_FREQ 4689
PI_HW_CLK_MAX_FREQ 250000000
PI_HW_CLK_MAX_FREQ_2711 375000000
```

count

The number of bytes to be transferred in an I2C, SPI, or Serial command.

CS

The GPIO used for the slave select signal when bit banging SPI.

data_bits: 1-32

The number of data bits to be used when adding serial data to a waveform.

```
PI_MIN_WAVE_DATABITS 1
PI_MAX_WAVE_DATABITS 32
```

DMAchannel: 0-15

```
PI_MIN_DMA_CHANNEL 0
PI_MAX_DMA_CHANNEL 15
```

double

A floating point number.

dutycycle: 0-range

A number representing the ratio of on time to off time for PWM.

The number may vary between 0 and range (default 255) where 0 is off and range is fully on.

edge: 0-2

The type of GPIO edge to generate an interrupt. See gpioSetISRFunc and gpioSetISRFuncEx.

```
RISING_EDGE 0
FALLING_EDGE 1
EITHER_EDGE 2
```

event: 0-31

An event is a signal used to inform one or more consumers to start an action.

eventFunc_t

```
typedef void (*eventFunc_t) (int event, uint32_t tick);
```

eventFuncEx_t

```
typedef void (*eventFuncEx_t)
   (int event, uint32_t tick, void *userdata);
```

f

A function.

*file

A full file path. To be accessible the path must match an entry in /opt/pigpio/access.

*fpat

A file path which may contain wildcards. To be accessible the path must match an entry in /opt/pigpio/access.

frequency: >=0

The number of times a GPIO is swiched on and off per second. This can be set per GPIO and may be as little as 5Hz or as much as 40KHz. The GPIO will be on for a proportion of the time as defined by its dutycycle.

gpio

A Broadcom numbered GPIO, in the range 0-53.

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 gpioHardwareRevision.

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  -  -  -  -
```

gpioAlertFunc_t

```
typedef void (*gpioAlertFunc_t) (int gpio, int level, uint32_t tick);
```

gpioAlertFuncEx_t

```
typedef void (*eventFuncEx_t)
   (int event, int level, uint32_t tick, void *userdata);
```

gpioCfg*

These functions are only effective if called before gpioInitialise.

gpioCfgBufferSize
gpioCfgClock
gpioCfgDMAchannel
gpioCfgDMAchannels
gpioCfgPermissions
gpioCfgInterfaces
gpioCfgSocketPort
gpioCfgMemAlloc

gpioGetSamplesFunc_t

```
typedef void (*gpioGetSamplesFunc_t)
   (const gpioSample_t *samples, int numSamples);
```

gpioGetSamplesFuncEx_t

```
typedef void (*gpioGetSamplesFuncEx_t)
   (const gpioSample_t *samples, int numSamples, void *userdata);
```

gpioISRFunc_t

```
typedef void (*gpioISRFunc_t)
   (int gpio, int level, uint32_t tick);
```

gpioISRFuncEx_t

```
typedef void (*gpioISRFuncEx_t)
   (int gpio, int level, uint32_t tick, void *userdata);
```

gpioPulse_t

```
typedef struct
{
   uint32_t gpioOn;
   uint32_t gpioOff;
   uint32_t usDelay;
} gpioPulse_t;
```

gpioSample_t

```
typedef struct
{
   uint32_t tick;
   uint32_t level;
} gpioSample_t;
```

gpioSignalFunc_t

```
typedef void (*gpioSignalFunc_t) (int signum);
```

gpioSignalFuncEx_t

```
typedef void (*gpioSignalFuncEx_t) (int signum, void *userdata);
```

gpioThreadFunc_t

```
typedef void *(gpioThreadFunc_t) (void *);
```

gpioTimerFunc_t

```
typedef void (*gpioTimerFunc_t) (void);
```

gpioTimerFuncEx_t

```
typedef void (*gpioTimerFuncEx_t) (void *userdata);
```

gpioWaveAdd*

One of

gpioWaveAddNew
gpioWaveAddGeneric
gpioWaveAddSerial

handle: >=0

A number referencing an object opened by one of

fileOpen
gpioNotifyOpen
i2cOpen
serOpen
spiOpen

i2cAddr: 0-0x7F

The address of a device on the I2C bus.

i2cBus: >=0

An I2C bus number.

i2cFlags: 0

Flags which modify an I2C open command. None are currently defined.

i2cReg: 0-255

A register of an I2C device.

ifFlags: 0-3

```
PI_DISABLE_FIFO_IF 1
PI_DISABLE_SOCK_IF 2
```

*inBuf

A buffer used to pass data to a function.

inLen

The number of bytes of data in a buffer.

int

A whole number, negative or positive.

int32_t

A 32-bit signed value.

invert

A flag used to set normal or inverted bit bang serial data level logic.

level

The level of a GPIO. Low or High.

```
PI_OFF 0
PI_ON 1

PI_CLEAR 0
PI_SET 1

PI_LOW 0
PI_HIGH 1
```

There is one exception. If a watchdog expires on a GPIO the level will be reported as PI_TIMEOUT. See gpioSetWatchdog.

```
PI_TIMEOUT 2
```

lVal: 0-4294967295 (Hex 0x0-0xFFFFFFFF, Octal 0-37777777777)

A 32-bit word value.

memAllocMode: 0-2

The DMA memory allocation mode.

```
PI_MEM_ALLOC_AUTO    0
PI_MEM_ALLOC_PAGEMAP 1
PI_MEM_ALLOC_MAILBOX 2
```

*micros

A value representing microseconds.

micros

A value representing microseconds.

millis

A value representing milliseconds.

MISO

The GPIO used for the MISO signal when bit banging SPI.

mode

1. The operational mode of a GPIO, normally INPUT or OUTPUT.

```
PI_INPUT 0
PI_OUTPUT 1
PI_ALT0 4
PI_ALT1 5
PI_ALT2 6
PI_ALT3 7
PI_ALT4 3
PI_ALT5 2
```

2. A file open mode.

```
PI_FILE_READ  1
PI_FILE_WRITE 2
PI_FILE_RW    3
```

The following values can be or'd into the mode.

```
PI_FILE_APPEND 4
PI_FILE_CREATE 8
PI_FILE_TRUNC  16
```

MOSI

The GPIO used for the MOSI signal when bit banging SPI.

numBits

The number of bits stored in a buffer.

numBytes

The number of bytes used to store characters in a string. Depending on the number of bits per character there may be 1, 2, or 4 bytes per character.

numPar: 0-10

The number of parameters passed to a script.

numPulses

The number of pulses to be added to a waveform.

numSegs

The number of segments in a combined I2C transaction.

numSockAddr

The number of network addresses allowed to use the socket interface.

0 means all addresses allowed.

offset

The associated data starts this number of microseconds from the start of the waveform.

*outBuf

A buffer used to return data from a function.

outLen

The size in bytes of an output buffer.

pad: 0-2

A set of GPIO which share common drivers.

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

padStrength: 1-16

The mA which may be drawn from each GPIO whilst still guaranteeing the high and low levels.

*param

An array of script parameters.

pctBOOL: 0-100

percent On-Off-Level (OOL) buffer to consume for wave output.

pctCB: 0-100

the percent of all DMA control blocks to consume.

pctTOOL: 0-100

the percent of OOL buffer to consume for wave input (flags).

pi_i2c_msg_t

```
typedef struct
{
   uint16_t addr;  // slave address
   uint16_t flags;
   uint16_t len;   // msg length
   uint8_t  *buf;  // pointer to msg data
} pi_i2c_msg_t;
```

port: 1024-32000

The port used to bind to the pigpio socket. Defaults to 8888.

pos

The position of an item.

primaryChannel: 0-15

The DMA channel used to time the sampling of GPIO and to time servo and PWM pulses.

*pth

A thread identifier, returned by gpioStartThread.

pthread_t

A thread identifier.

pud: 0-2

The setting of the pull up/down resistor for a GPIO, which may be off, pull-up, or pull-down.

```
PI_PUD_OFF 0
PI_PUD_DOWN 1
PI_PUD_UP 2
```

pulseLen

1-100, the length of a trigger pulse in microseconds.

*pulses

An array of pulses to be added to a waveform.

pulsewidth: 0, 500-2500

```
PI_SERVO_OFF 0
PI_MIN_SERVO_PULSEWIDTH 500
PI_MAX_SERVO_PULSEWIDTH 2500
```

PWMduty: 0-1000000 (1M)

The hardware PWM dutycycle.

```
PI_HW_PWM_RANGE 1000000
```

PWMfreq: 1-125M (1-187.5M for the BCM2711)

The hardware PWM frequency.

```
PI_HW_PWM_MIN_FREQ 1
PI_HW_PWM_MAX_FREQ 125000000
PI_HW_PWM_MAX_FREQ_2711 187500000
```

range: 25-40000

```
PI_MIN_DUTYCYCLE_RANGE 25
PI_MAX_DUTYCYCLE_RANGE 40000
```

rawCbs_t

```
typedef struct // linux/arch/arm/mach-bcm2708/include/mach/dma.h
{
   unsigned long info;
   unsigned long src;
   unsigned long dst;
   unsigned long length;
   unsigned long stride;
   unsigned long next;
   unsigned long pad[2];
} rawCbs_t;
```

rawSPI_t

```
typedef struct
{
   int clk;     // GPIO for clock
   int mosi;    // GPIO for MOSI
   int miso;    // GPIO for MISO
   int ss_pol;  // slave select off state
   int ss_us;   // delay after slave select
   int clk_pol; // clock off state
   int clk_pha; // clock phase
   int clk_us;  // clock micros
} rawSPI_t;
```

rawWave_t

```
typedef struct
{
   uint32_t gpioOn;
   uint32_t gpioOff;
   uint32_t usDelay;
   uint32_t flags;
} rawWave_t;
```

rawWaveInfo_t

```
typedef struct
{
   uint16_t botCB;  // first CB used by wave
   uint16_t topCB;  // last CB used by wave
   uint16_t botOOL; // last OOL used by wave
   uint16_t topOOL; // first OOL used by wave
   uint16_t deleted;
   uint16_t numCB;
   uint16_t numBOOL;
   uint16_t numTOOL;
} rawWaveInfo_t;
```

*retBuf

A buffer to hold a number of bytes returned to a used customised function,

retMax

The maximum number of bytes a user customised function should return.

*rxBuf

A pointer to a buffer to receive data.

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

A pointer to the text of a script.

script_id

An id of a stored script as returned by gpioStoreScript.

*scriptName

The name of a shell script to be executed. The script must be present in /opt/pigpio/cgi and must have execute permission.

*scriptString

The string to be passed to a shell script to be executed.

SDA

The user GPIO to use for data when bit banging I2C.

secondaryChannel: 0-6

The DMA channel used to time output waveforms.

*seconds

A pointer to a uint32_t to store the second component of a returned time.

seconds

The number of seconds.

seekFrom

```
PI_FROM_START   0
PI_FROM_CURRENT 1
PI_FROM_END     2
```

seekOffset

The number of bytes to move forward (positive) or backwards (negative) from the seek position (start, current, or end of file).

*segs

An array of segments which make up a combined I2C transaction.

serFlags

Flags which modify a serial open command. None are currently defined.

*sertty

The name of a serial tty device, e.g. /dev/ttyAMA0, /dev/ttyUSB0, /dev/tty1.

setting

A value used to set a flag, 0 for false, non-zero for true.

signum: 0-63

```
PI_MIN_SIGNUM 0
PI_MAX_SIGNUM 63
```

size_t

A standard type used to indicate the size of an object in bytes.

*sockAddr

An array of network addresses allowed to use the socket interface encoded as 32 bit numbers.

E.g. address 192.168.1.66 would be encoded as 0x4201a8c0.

*spi

A pointer to a rawSPI_t structure.

spiBitFirst

GPIO reads are made from spiBitFirst to spiBitLast.

spiBitLast

GPIO reads are made from spiBitFirst to spiBitLast.

spiBits

The number of bits to transfer in a raw SPI transaction.

spiChan

A SPI channel, 0-2.

spiFlags

See spiOpen and bbSPIOpen.

spiSS

The SPI slave select GPIO in a raw SPI transaction.

spiTxBits

The number of bits to transfer dring a raw SPI transaction

steady: 0-300000

The number of microseconds level changes must be stable for before reporting the level changed (gpioGlitchFilter) or triggering the active part of a noise filter (gpioNoiseFilter).

stop_bits: 2-8

The number of (half) stop bits to be used when adding serial data to a waveform.

```
PI_MIN_WAVE_HALFSTOPBITS 2
PI_MAX_WAVE_HALFSTOPBITS 8
```

*str

An array of characters.

timeout

A GPIO level change timeout in milliseconds.

gpioSetWatchdog
```
PI_MIN_WDOG_TIMEOUT 0
PI_MAX_WDOG_TIMEOUT 60000
```

gpioSetISRFunc and gpioSetISRFuncEx
```
<=0 cancel timeout
>0 timeout after specified milliseconds
```

timer

```
PI_MIN_TIMER 0
PI_MAX_TIMER 9
```

timetype

```
PI_TIME_RELATIVE 0
PI_TIME_ABSOLUTE 1
```

*txBuf

An array of bytes to transmit.

uint32_t: 0-0-4,294,967,295 (Hex 0x0-0xFFFFFFFF)

A 32-bit unsigned value.

uint64_t: 0-(2^64)-1

A 64-bit unsigned value.

unsigned

A whole number >= 0.

updateMask

A 64 bit mask indicating which GPIO may be written to by the user.

If GPIO#n may be written then bit (1<<n) is set.

user_gpio

0-31, a Broadcom numbered GPIO.

See gpio.

*userdata

A pointer to arbitrary user data. This may be used to identify the instance.

You must ensure that the pointer is in scope at the time it is processed. If it is a pointer to a global this is automatic. Do not pass the address of a local variable. If you want to pass a transient object then use the following technique.

In the calling function:

```
user_type *userdata;

user_type my_userdata;

userdata = malloc(sizeof(user_type));

*userdata = my_userdata;
```

In the receiving function:

```
user_type my_userdata = *(user_type*)userdata;

free(userdata);
```

void

Denoting no parameter is required

wave_id

A number identifying a waveform created by gpioWaveCreate.

wave_mode

The mode determines if the waveform is sent once or cycles repeatedly. The SYNC variants wait for the current waveform to reach the end of a cycle or finish before starting the new waveform.

```
PI_WAVE_MODE_ONE_SHOT      0
PI_WAVE_MODE_REPEAT        1
PI_WAVE_MODE_ONE_SHOT_SYNC 2
PI_WAVE_MODE_REPEAT_SYNC   3
```

wVal: 0-65535 (Hex 0x0-0xFFFF, Octal 0-0177777)

A 16-bit word value.

Socket Command Codes

```
#define PI_CMD_MODES  0
#define PI_CMD_MODEG  1
#define PI_CMD_PUD    2
#define PI_CMD_READ   3
#define PI_CMD_WRITE  4
#define PI_CMD_PWM    5
#define PI_CMD_PRS    6
#define PI_CMD_PFS    7
#define PI_CMD_SERVO  8
#define PI_CMD_WDOG   9
#define PI_CMD_BR1   10
#define PI_CMD_BR2   11
#define PI_CMD_BC1   12
#define PI_CMD_BC2   13
#define PI_CMD_BS1   14
#define PI_CMD_BS2   15
#define PI_CMD_TICK  16
#define PI_CMD_HWVER 17
#define PI_CMD_NO    18
#define PI_CMD_NB    19
#define PI_CMD_NP    20
#define PI_CMD_NC    21
#define PI_CMD_PRG   22
#define PI_CMD_PFG   23
#define PI_CMD_PRRG  24
#define PI_CMD_HELP  25
#define PI_CMD_PIGPV 26
#define PI_CMD_WVCLR 27
#define PI_CMD_WVAG  28
#define PI_CMD_WVAS  29
#define PI_CMD_WVGO  30
#define PI_CMD_WVGOR 31
#define PI_CMD_WVBSY 32
#define PI_CMD_WVHLT 33
#define PI_CMD_WVSM  34
#define PI_CMD_WVSP  35
#define PI_CMD_WVSC  36
#define PI_CMD_TRIG  37
#define PI_CMD_PROC  38
#define PI_CMD_PROCD 39
#define PI_CMD_PROCR 40
#define PI_CMD_PROCS 41
#define PI_CMD_SLRO  42
#define PI_CMD_SLR   43
#define PI_CMD_SLRC  44
#define PI_CMD_PROCP 45
#define PI_CMD_MICS  46
#define PI_CMD_MILS  47
#define PI_CMD_PARSE 48
#define PI_CMD_WVCRE 49
#define PI_CMD_WVDEL 50
#define PI_CMD_WVTX  51
#define PI_CMD_WVTXR 52
#define PI_CMD_WVNEW 53

#define PI_CMD_I2CO  54
#define PI_CMD_I2CC  55
#define PI_CMD_I2CRD 56
#define PI_CMD_I2CWD 57
#define PI_CMD_I2CWQ 58
#define PI_CMD_I2CRS 59
#define PI_CMD_I2CWS 60
#define PI_CMD_I2CRB 61
#define PI_CMD_I2CWB 62
#define PI_CMD_I2CRW 63
#define PI_CMD_I2CWW 64
#define PI_CMD_I2CRK 65
#define PI_CMD_I2CWK 66
```

```
#define PI_CMD_I2CRI 67
#define PI_CMD_I2CWI 68
#define PI_CMD_I2CPC 69
#define PI_CMD_I2CPK 70

#define PI_CMD_SPIO  71
#define PI_CMD_SPIC  72
#define PI_CMD_SPIR  73
#define PI_CMD_SPIW  74
#define PI_CMD_SPIX  75

#define PI_CMD_SERO  76
#define PI_CMD_SERC  77
#define PI_CMD_SERRB 78
#define PI_CMD_SERWB 79
#define PI_CMD_SERR  80
#define PI_CMD_SERW  81
#define PI_CMD_SERDA 82

#define PI_CMD_GDC   83
#define PI_CMD_GPW   84

#define PI_CMD_HC    85
#define PI_CMD_HP    86

#define PI_CMD_CF1   87
#define PI_CMD_CF2   88

#define PI_CMD_BI2CC 89
#define PI_CMD_BI2CO 90
#define PI_CMD_BI2CZ 91

#define PI_CMD_I2CZ  92

#define PI_CMD_WVCHA 93

#define PI_CMD_SLRI  94

#define PI_CMD_CGI   95
#define PI_CMD_CSI   96

#define PI_CMD_FG    97
#define PI_CMD_FN    98

#define PI_CMD_NOIB  99

#define PI_CMD_WVTXM 100
#define PI_CMD_WVTAT 101

#define PI_CMD_PADS  102
#define PI_CMD_PADG  103

#define PI_CMD_FO    104
#define PI_CMD_FC    105
#define PI_CMD_FR    106
#define PI_CMD_FW    107
#define PI_CMD_FS    108
#define PI_CMD_FL    109

#define PI_CMD_SHELL 110

#define PI_CMD_BSPIC 111
#define PI_CMD_BSPIO 112
#define PI_CMD_BSPIX 113

#define PI_CMD_BSCX  114

#define PI_CMD_EVM   115
#define PI_CMD_EVT   116
```

```
#define PI_CMD_PROCU 117
#define PI_CMD_WVCAP 118
```

Error Codes

```
#define PI_INIT_FAILED       -1 // gpioInitialise failed
#define PI_BAD_USER_GPIO     -2 // GPIO not 0-31
#define PI_BAD_GPIO          -3 // GPIO not 0-53
#define PI_BAD_MODE          -4 // mode not 0-7
#define PI_BAD_LEVEL         -5 // level not 0-1
#define PI_BAD_PUD           -6 // pud not 0-2
#define PI_BAD_PULSEWIDTH    -7 // pulsewidth not 0 or 500-2500
#define PI_BAD_DUTYCYCLE     -8 // dutycycle outside set range
#define PI_BAD_TIMER         -9 // timer not 0-9
#define PI_BAD_MS           -10 // ms not 10-60000
#define PI_BAD_TIMETYPE     -11 // timetype not 0-1
#define PI_BAD_SECONDS      -12 // seconds < 0
#define PI_BAD_MICROS       -13 // micros not 0-999999
#define PI_TIMER_FAILED     -14 // gpioSetTimerFunc failed
#define PI_BAD_WDOG_TIMEOUT -15 // timeout not 0-60000
#define PI_NO_ALERT_FUNC    -16 // DEPRECATED
#define PI_BAD_CLK_PERIPH   -17 // clock peripheral not 0-1
#define PI_BAD_CLK_SOURCE   -18 // DEPRECATED
#define PI_BAD_CLK_MICROS   -19 // clock micros not 1, 2, 4, 5, 8, or 10
#define PI_BAD_BUF_MILLIS   -20 // buf millis not 100-10000
#define PI_BAD_DUTYRANGE    -21 // dutycycle range not 25-40000
#define PI_BAD_DUTY_RANGE   -21 // DEPRECATED (use PI_BAD_DUTYRANGE)
#define PI_BAD_SIGNUM       -22 // signum not 0-63
#define PI_BAD_PATHNAME     -23 // can't open pathname
#define PI_NO_HANDLE        -24 // no handle available
#define PI_BAD_HANDLE       -25 // unknown handle
#define PI_BAD_IF_FLAGS     -26 // ifFlags > 4
#define PI_BAD_CHANNEL      -27 // DMA channel not 0-15
#define PI_BAD_PRIM_CHANNEL -27 // DMA primary channel not 0-15
#define PI_BAD_SOCKET_PORT  -28 // socket port not 1024-32000
#define PI_BAD_FIFO_COMMAND -29 // unrecognized fifo command
#define PI_BAD_SECO_CHANNEL -30 // DMA secondary channel not 0-15
#define PI_NOT_INITIALISED  -31 // function called before gpioInitialise
#define PI_INITIALISED      -32 // function called after gpioInitialise
#define PI_BAD_WAVE_MODE    -33 // waveform mode not 0-3
#define PI_BAD_CFG_INTERNAL -34 // bad parameter in gpioCfgInternals call
#define PI_BAD_WAVE_BAUD    -35 // baud rate not 50-250K(RX)/50-1M(TX)
#define PI_TOO_MANY_PULSES  -36 // waveform has too many pulses
#define PI_TOO_MANY_CHARS   -37 // waveform has too many chars
#define PI_NOT_SERIAL_GPIO  -38 // no bit bang serial read on GPIO
#define PI_BAD_SERIAL_STRUC -39 // bad (null) serial structure parameter
#define PI_BAD_SERIAL_BUF   -40 // bad (null) serial buf parameter
#define PI_NOT_PERMITTED    -41 // GPIO operation not permitted
#define PI_SOME_PERMITTED   -42 // one or more GPIO not permitted
#define PI_BAD_WVSC_COMMND  -43 // bad WVSC subcommand
#define PI_BAD_WVSM_COMMND  -44 // bad WVSM subcommand
#define PI_BAD_WVSP_COMMND  -45 // bad WVSP subcommand
#define PI_BAD_PULSELEN     -46 // trigger pulse length not 1-100
#define PI_BAD_SCRIPT       -47 // invalid script
#define PI_BAD_SCRIPT_ID    -48 // unknown script id
#define PI_BAD_SER_OFFSET   -49 // add serial data offset > 30 minutes
#define PI_GPIO_IN_USE      -50 // GPIO already in use
#define PI_BAD_SERIAL_COUNT -51 // must read at least a byte at a time
#define PI_BAD_PARAM_NUM    -52 // script parameter id not 0-9
#define PI_DUP_TAG          -53 // script has duplicate tag
#define PI_TOO_MANY_TAGS    -54 // script has too many tags
#define PI_BAD_SCRIPT_CMD   -55 // illegal script command
#define PI_BAD_VAR_NUM      -56 // script variable id not 0-149
#define PI_NO_SCRIPT_ROOM   -57 // no more room for scripts
```

```
#define PI_NO_MEMORY        -58 // can't allocate temporary memory
#define PI_SOCK_READ_FAILED -59 // socket read failed
#define PI_SOCK_WRIT_FAILED -60 // socket write failed
#define PI_TOO_MANY_PARAM   -61 // too many script parameters (> 10)
#define PI_NOT_HALTED       -62 // DEPRECATED
#define PI_SCRIPT_NOT_READY -62 // script initialising
#define PI_BAD_TAG          -63 // script has unresolved tag
#define PI_BAD_MICS_DELAY   -64 // bad MICS delay (too large)
#define PI_BAD_MILS_DELAY   -65 // bad MILS delay (too large)
#define PI_BAD_WAVE_ID      -66 // non existent wave id
#define PI_TOO_MANY_CBS     -67 // No more CBs for waveform
#define PI_TOO_MANY_OOL     -68 // No more OOL for waveform
#define PI_EMPTY_WAVEFORM   -69 // attempt to create an empty waveform
#define PI_NO_WAVEFORM_ID   -70 // no more waveforms
#define PI_I2C_OPEN_FAILED  -71 // can't open I2C device
#define PI_SER_OPEN_FAILED  -72 // can't open serial device
#define PI_SPI_OPEN_FAILED  -73 // can't open SPI device
#define PI_BAD_I2C_BUS      -74 // bad I2C bus
#define PI_BAD_I2C_ADDR     -75 // bad I2C address
#define PI_BAD_SPI_CHANNEL  -76 // bad SPI channel
#define PI_BAD_FLAGS        -77 // bad i2c/spi/ser open flags
#define PI_BAD_SPI_SPEED    -78 // bad SPI speed
#define PI_BAD_SER_DEVICE   -79 // bad serial device name
#define PI_BAD_SER_SPEED    -80 // bad serial baud rate
#define PI_BAD_PARAM        -81 // bad i2c/spi/ser parameter
#define PI_I2C_WRITE_FAILED -82 // i2c write failed
#define PI_I2C_READ_FAILED  -83 // i2c read failed
#define PI_BAD_SPI_COUNT    -84 // bad SPI count
#define PI_SER_WRITE_FAILED -85 // ser write failed
#define PI_SER_READ_FAILED  -86 // ser read failed
#define PI_SER_READ_NO_DATA -87 // ser read no data available
#define PI_UNKNOWN_COMMAND  -88 // unknown command
#define PI_SPI_XFER_FAILED  -89 // spi xfer/read/write failed
#define PI_BAD_POINTER      -90 // bad (NULL) pointer
#define PI_NO_AUX_SPI       -91 // no auxiliary SPI on Pi A or B
#define PI_NOT_PWM_GPIO     -92 // GPIO is not in use for PWM
#define PI_NOT_SERVO_GPIO   -93 // GPIO is not in use for servo pulses
#define PI_NOT_HCLK_GPIO    -94 // GPIO has no hardware clock
#define PI_NOT_HPWM_GPIO    -95 // GPIO has no hardware PWM
#define PI_BAD_HPWM_FREQ    -96 // invalid hardware PWM frequency
#define PI_BAD_HPWM_DUTY    -97 // hardware PWM dutycycle not 0-1M
#define PI_BAD_HCLK_FREQ    -98 // invalid hardware clock frequency
#define PI_BAD_HCLK_PASS    -99 // need password to use hardware clock 1
#define PI_HPWM_ILLEGAL    -100 // illegal, PWM in use for main clock
#define PI_BAD_DATABITS    -101 // serial data bits not 1-32
#define PI_BAD_STOPBITS    -102 // serial (half) stop bits not 2-8
#define PI_MSG_TOOBIG      -103 // socket/pipe message too big
#define PI_BAD_MALLOC_MODE -104 // bad memory allocation mode
#define PI_TOO_MANY_SEGS   -105 // too many I2C transaction segments
#define PI_BAD_I2C_SEG     -106 // an I2C transaction segment failed
#define PI_BAD_SMBUS_CMD   -107 // SMBus command not supported by driver
#define PI_NOT_I2C_GPIO    -108 // no bit bang I2C in progress on GPIO
#define PI_BAD_I2C_WLEN    -109 // bad I2C write length
#define PI_BAD_I2C_RLEN    -110 // bad I2C read length
#define PI_BAD_I2C_CMD     -111 // bad I2C command
#define PI_BAD_I2C_BAUD    -112 // bad I2C baud rate, not 50-500k
#define PI_CHAIN_LOOP_CNT  -113 // bad chain loop count
#define PI_BAD_CHAIN_LOOP  -114 // empty chain loop
#define PI_CHAIN_COUNTER   -115 // too many chain counters
#define PI_BAD_CHAIN_CMD   -116 // bad chain command
#define PI_BAD_CHAIN_DELAY -117 // bad chain delay micros
#define PI_CHAIN_NESTING   -118 // chain counters nested too deeply
#define PI_CHAIN_TOO_BIG   -119 // chain is too long
#define PI_DEPRECATED      -120 // deprecated function removed
#define PI_BAD_SER_INVERT  -121 // bit bang serial invert not 0 or 1
#define PI_BAD_EDGE        -122 // bad ISR edge value, not 0-2
#define PI_BAD_ISR_INIT    -123 // bad ISR initialisation
#define PI_BAD_FOREVER     -124 // loop forever must be last command
#define PI_BAD_FILTER      -125 // bad filter parameter
```

```
#define PI_BAD_PAD          -126 // bad pad number
#define PI_BAD_STRENGTH     -127 // bad pad drive strength
#define PI_FIL_OPEN_FAILED -128 // file open failed
#define PI_BAD_FILE_MODE    -129 // bad file mode
#define PI_BAD_FILE_FLAG    -130 // bad file flag
#define PI_BAD_FILE_READ    -131 // bad file read
#define PI_BAD_FILE_WRITE   -132 // bad file write
#define PI_FILE_NOT_ROPEN   -133 // file not open for read
#define PI_FILE_NOT_WOPEN   -134 // file not open for write
#define PI_BAD_FILE_SEEK    -135 // bad file seek
#define PI_NO_FILE_MATCH    -136 // no files match pattern
#define PI_NO_FILE_ACCESS   -137 // no permission to access file
#define PI_FILE_IS_A_DIR    -138 // file is a directory
#define PI_BAD_SHELL_STATUS -139 // bad shell return status
#define PI_BAD_SCRIPT_NAME -140 // bad script name
#define PI_BAD_SPI_BAUD     -141 // bad SPI baud rate, not 50-500k
#define PI_NOT_SPI_GPIO     -142 // no bit bang SPI in progress on GPIO
#define PI_BAD_EVENT_ID     -143 // bad event id
#define PI_CMD_INTERRUPTED -144 // Used by Python
#define PI_NOT_ON_BCM2711   -145 // not available on BCM2711
#define PI_ONLY_ON_BCM2711 -146 // only available on BCM2711

#define PI_PIGIF_ERR_0    -2000
#define PI_PIGIF_ERR_99   -2099

#define PI_CUSTOM_ERR_0   -3000
#define PI_CUSTOM_ERR_999 -3999
```

Defaults

```
#define PI_DEFAULT_BUFFER_MILLIS           120
#define PI_DEFAULT_CLK_MICROS              5
#define PI_DEFAULT_CLK_PERIPHERAL          PI_CLOCK_PCM
#define PI_DEFAULT_IF_FLAGS                0
#define PI_DEFAULT_FOREGROUND              0
#define PI_DEFAULT_DMA_CHANNEL             14
#define PI_DEFAULT_DMA_PRIMARY_CHANNEL     14
#define PI_DEFAULT_DMA_SECONDARY_CHANNEL   6
#define PI_DEFAULT_DMA_PRIMARY_CH_2711     7
#define PI_DEFAULT_DMA_SECONDARY_CH_2711   6
#define PI_DEFAULT_DMA_NOT_SET             15
#define PI_DEFAULT_SOCKET_PORT             8888
#define PI_DEFAULT_SOCKET_PORT_STR         "8888"
#define PI_DEFAULT_SOCKET_ADDR_STR         "localhost"
#define PI_DEFAULT_UPDATE_MASK_UNKNOWN     0x0000000FFFFFFCLL
#define PI_DEFAULT_UPDATE_MASK_B1          0x03E7CF93
#define PI_DEFAULT_UPDATE_MASK_A_B2        0xFBC7CF9C
#define PI_DEFAULT_UPDATE_MASK_APLUS_BPLUS 0x0080480FFFFFFCLL
#define PI_DEFAULT_UPDATE_MASK_ZERO        0x0080000FFFFFFCLL
#define PI_DEFAULT_UPDATE_MASK_PI2B        0x0080480FFFFFFCLL
#define PI_DEFAULT_UPDATE_MASK_PI3B        0x0000000FFFFFFCLL
#define PI_DEFAULT_UPDATE_MASK_PI4B        0x0000000FFFFFFCLL
#define PI_DEFAULT_UPDATE_MASK_COMPUTE     0x00FFFFFFFFFFFFLL
#define PI_DEFAULT_MEM_ALLOC_MODE          PI_MEM_ALLOC_AUTO

#define PI_DEFAULT_CFG_INTERNALS           0
```