Wiring PI Full API

Esta librería contiene funciones para usar el puerto GPIO y otras funciones útiles a la hora de programar la Raspeberri Pi en C. El documento original se encuentra en el sitio web de Gordon Henderson[1].

Before using the ***WiringPi*** GPIO library, you need to include its header file in your programs:

```
#include <wiringPi.h>
```

You may also need to add

```
-I/usr/local/include -L/usr/local/lib -lwiringPi
```

to the compile line of your program depending on the environment you are using. The important one is ***-lwiringPi***

You may also need additional **#include** lines, depending on the modules you are using.

Setup

There are four ways to initialise ***wiringPi***.

- **int wiringPiSetup (void) ;**
- **int wiringPiSetupGpio (void) ;**
- **int wiringPiSetupPhys (void) ;**
- **int wiringPiSetupSys (void) ;**

One of the ***setup*** functions **must** be called at the start of your program or your program will fail to work correctly. You may experience symptoms from it simply not working to segfaults and timing issues.

Note: ***wiringPi*** version 1 returned an error code if these functions failed for whatever reason. Version 2 returns always returns zero. After discussions and inspection of many programs written by users of ***wiringPi*** and observing that many people don't bother checking the return code, I took the stance that should one of the ***wiringPi*** setup functions fail, then it would be considered a fatal program fault and the program execution will be terminated at that point with an error message printed on the terminal.

- If you want to restore the v1 behaviour, then you need to set the environment variable: **WIRINGPI_CODES** (to any value, it just needs to exist)

The differences between the setup functions are as follows:

- **wiringPiSetup (void) ;**

This initialises ***wiringPi*** and assumes that the calling program is going to be using the ***wiringPi*** pin numbering scheme. This is a simplified numbering scheme which provides a mapping from virtual pin numbers 0 through 16 to the real underlying Broadcom GPIO pin numbers. See the pins page for a table which maps the ***wiringPi*** pin number to the Broadcom GPIO pin number to the physical location on the edge connector.

[1] http://wiringpi.com/reference/

This function needs to be called with root privileges.

- **wiringPiSetupGpio (void) ;**

This is identical to above, however it allows the calling programs to use the Broadcom GPIO pin numbers directly with no re-mapping.

As above, this function needs to be called with root privileges, and note that some pins are different from revision 1 to revision 2 boards.

- **wiringPiSetupPhys (void) ;**

Identical to above, however it allows the calling programs to use the physical pin numbers *on the P1 connector only*.

As above, this function needs to be called with root priviliges.

- **wiringPiSetupSys (void) ;**

This initialises ***wiringPi*** but uses the */sys/class/gpio* interface rather than accessing the hardware directly. This can be called as a non-root user provided the GPIO pins have been exported before-hand using the **gpio** program. Pin numbering in this mode is the native Broadcom GPIO numbers – the same as wiringPiSetupGpio() above, so be aware of the differences between Rev 1 and Rev 2 boards.

Note: In this mode you can only use the pins which have been exported via the /sys/class/gpio interface before you run your program. You can do this in a separate shell-script, or by using the system() function from inside your program to call the **gpio** program.

Also note that some functions have no effect when using this mode as they're not currently possible to action unless called with root privileges. (although you can use system() to call **gpio** to set/change modes if needed)

Core Functions

These functions work directly on the Raspberry Pi and also with external GPIO modules such as GPIO expanders and so on, although not all modules support all functions – e.g. the PiFace is pre-configured for its fixed inputs and outputs, and the Raspberry Pi has no on-board analog hardware.

- **void pinMode (int pin, int mode) ;**

This sets the mode of a pin to either **INPUT**, **OUTPUT**, **PWM_OUTPUT** or **GPIO_CLOCK**. Note that only ***wiringPi*** pin 1 (BCM_GPIO 18) supports PWM output and only ***wiringPi*** pin 7 (BCM_GPIO 4) supports CLOCK output modes.

This function has no effect when in *Sys* mode. If you need to change the pin mode, then you can do it with the **gpio** program in a script before you start your program.

- **void pullUpDnControl (int pin, int pud) ;**

This sets the pull-up or pull-down resistor mode on the given pin, which should be set as an input. Unlike the Arduino, the BCM2835 has both pull-up an down internal resistors. The parameter **pud** should be; **PUD_OFF**, (no pull up/down), **PUD_DOWN** (pull to ground) or **PUD_UP** (pull to 3.3v) The internal pull up/down resistors have a value of approximately 50KΩ on the Raspberry Pi.

This function has no effect on the Raspberry Pi's GPIO pins when in *Sys* mode. If you need to activate a pull-up/pull-down, then you can do it with the **gpio** program in a script before you start your program.

- **void digitalWrite (int pin, int value) ;**

Writes the value **HIGH** or **LOW** (1 or 0) to the given pin which must have been previously set as an output.

WiringPi treats any non-zero number as **HIGH**, however 0 is the only representation of **LOW**.

- **void pwmWrite (int pin, int value) ;**

Writes the value to the PWM register for the given pin. The Raspberry Pi has one on-board PWM pin, pin 1 (BMC_GPIO 18, Phys 12) and the range is 0-1024. Other PWM devices may have other PWM ranges.

This function is not able to control the Pi's on-board PWM when in *Sys* mode.

- **int digitalRead (int pin) ;**

This function returns the value read at the given pin. It will be **HIGH** or **LOW** (1 or 0) depending on the logic level at the pin.

- **analogRead (int pin) ;**

This returns the value read on the supplied analog input pin. You will need to register additional analog modules to enable this function for devices such as the Gertboard, quick2Wire analog board, etc.

- **analogWrite (int pin, int value) ;**

This writes the given value to the supplied analog pin. You will need to register additional analog modules to enable this function for devices such as the Gertboard.

Raspberry Pi Specifics

These functions are not part of the core ***wiringPi*** set, but act specifically on the Raspberry Pi hardware itself. Some external hardware driver modules may provide some of this functionality though.

- **void digitalWriteByte (int value) ;**

This writes the 8-bit byte supplied to the first 8 GPIO pins. It's the fastest way to set all 8 bits at once to a particular value, although it still takes two write operations to the Pi's GPIO hardware.

- **pwmSetMode (int mode) ;**

The PWM generator can run in 2 modes – "balanced" and "mark:space". The mark:space mode is traditional, however the default mode in the Pi is "balanced". You can switch modes by supplying the parameter: **PWM_MODE_BAL** or **PWM_MODE_MS**.

- **pwmSetRange (unsigned int range) ;**

This sets the range register in the PWM generator. The default is 1024.

- **pwmSetClock (int divisor) ;**

This sets the divisor for the PWM clock.

Note: The PWM control functions can not be used when in Sys mode. To understand more about the PWM system, you'll need to read the Broadcom ARM peripherals manual.

- **piBoardRev (void) ;**

This returns the board revision of the Raspberry Pi. It will be either 1 or 2. Some of the BCM_GPIO pins changed number and function when moving from board revision 1 to 2, so if you are using BCM_GPIO pin numbers, then you need to be aware of the differences.

- **wpiPinToGpio (int wPiPin) ;**

This returns the BCM_GPIO pin number of the supplied ***wiringPi*** pin. It takes the board revision into account.

- **physPinToGpio (int physPin) ;**

This returns the BCM_GPIO pin number of the supplied physical pin on the **P1** connector.

- **setPadDrive (int group, int value) ;**

This sets the "strength" of the pad drivers for a particular group of pins. There are 3 groups of pins and the drive strength is from 0 to 7. Do not use this unless you know what you are doing.

Timing

While Linux provides a multitude of system calls and functions to providing various timing and sleeping functions, sometimes it can be quite confusing, especially if you are new to Linux, so the ones presented here mimic those available on the Arduino platform, making porting code and writing new code somewhat easier.

Note: Even if you are not using any of the input/output functions you still need to call one of the ***wiringPi*** setup functions – just use **wiringPiSetupSys()** if you don't need root access in your program and remember to #include <wiringPi.h>

- **unsigned int millis (void)**

This returns a number representing the number of milliseconds since your program called one of the ***wiringPiSetup*** functions. It returns an unsigned 32-bit number which wraps after 49 days.

- **unsigned int micros (void)**

This returns a number representing the number of microseconds since your program called one of the ***wiringPiSetup*** functions. It returns an unsigned 32-bit number which wraps after approximately 71 minutes.

- **void delay (unsigned int howLong)**

This causes program execution to pause for at least ***howLong*** milliseconds**.** Due to the multi-tasking nature of Linux it could be longer. Note that the maximum delay is an unsigned 32-bit integer or approximately 49 days.

- **void delayMicroseconds (unsigned int howLong)**

This causes program execution to pause for at least ***howLong*** microseconds**.** Due to the multi-tasking nature of Linux it could be longer. Note that the maximum delay is an unsigned 32-bit integer microseconds or approximately 71 minutes.

Delays under 100 microseconds are timed using a hard-coded loop continually polling the system time, Delays over 100 microseconds are done using the system nanosleep() function – You may need to consider the implications of very short delays on the overall performance of the system, especially if using threads.

Priority, Interrupts and Threads

WiringPi provides some helper functions to allow you to manage your program (or thread) priority and to help launch a new thread from inside your program. Threads run concurrently with your main program and can be used for a variety of purposes. To learn more about threads, search for "Posix Threads"

Program or Thread Priority

- **int piHiPri (int priority) ;**

This attempts to shift your program (or thread in a multi-threaded program) to a higher priority and enables a real-time scheduling. The **priority** parameter should be from 0 (the default) to 99 (the maximum). This won't make your program go any faster, but it will give it a bigger slice of time when other programs are running. The priority parameter works relative to others – so you can make one program priority 1 and another priority 2 and it will have the same effect as setting one to 10 and the other to 90 (as long as no other programs are running with elevated priorities)

The return value is 0 for success and -1 for error. If an error is returned, the program should then consult the *errno* global variable, as per the usual conventions.

Note: Only programs running as root can change their priority. If called from a non-root program then nothing happens.

Interrupts

With a newer kernel patched with the GPIO interrupt handling code, you can now wait for an interrupt in your program. This frees up the processor to do other tasks while you're waiting for that interrupt. The GPIO can be set to interrupt on a rising, falling or both edges of the incoming signal.

Note: Jan 2013: The waitForInterrupt() function is deprecated – you should use the newer and easier to use wiringPiISR() function below.

- **int waitForInterrupt (int pin, int timeOut) ;**

When called, it will wait for an interrupt event to happen on that pin and your program will be stalled. The **timeOut** parameter is given in milliseconds, or can be -1 which means to wait forever.

The return value is -1 if an error occurred (and *errno* will be set appropriately), 0 if it timed out, or 1 on a successful interrupt event.

Before you call waitForInterrupt, you must first initialise the GPIO pin and at present the only way to do this is to use the **gpio** program, either in a script, or using the system() call from inside your program.

e.g. We want to wait for a falling-edge interrupt on GPIO pin 0, so to setup the hardware, we need to run:

```
gpio edge 0 falling
```

before running the program.

- **int wiringPiISR (int pin, int edgeType, void (*function)(void)) ;**

This function registers a function to received interrupts on the specified pin. The edgeType parameter is either **INT_EDGE_FALLING**, **INT_EDGE_RISING**, **INT_EDGE_BOTH** or **INT_EDGE_SETUP**. If it is **INT_EDGE_SETUP** then no initialisation of the pin will happen – it's assumed that you have already setup the pin elsewhere (e.g. with the ***gpio*** program), but if you specify one of the other types, then the pin will be exported and initialised as specified. This is accomplished via a suitable call to the ***gpio*** utility program, so it need to be available.

The pin number is supplied in the current mode – native wiringPi, BCM_GPIO, physical or Sys modes.

This function will work in any mode, and does not need root privileges to work.

The function will be called when the interrupt triggers. When it is triggered, it's cleared in the dispatcher before calling your function, so if a subsequent interrupt fires before you finish your handler, then it won't be missed. (However it can only track one more interrupt, if more than one interrupt fires while one is being handled then they will be ignored)

This function is run at a high priority (if the program is run using sudo, or as root) and executes concurrently with the main program. It has full access to all the global variables, open file handles and so on.

See the *isr.c* example program for more details on how to use this feature.

Concurrent Processing (multi-threading)

wiringPi has a simplified interface to the Linux implementation of Posix threads, as well as a (simplified) mechanisms to access mutex's (Mutual exclusions)

Using these functions you can create a new process (a function inside your main program) which runs concurrently with your main program and using the mutex mechanisms, safely pass variables between them.

- **int piThreadCreate (*name*) ;**

This function creates a thread which is another function in your program previously declared using the **PI_THREAD** declaration. This function is then run concurrently with your main program. An example may be to have this function wait for an interrupt while your program carries on doing other tasks. The thread can indicate an event, or action by using global variables to communicate back to the main program, or other threads.

Thread functions are declared as follows:

```
PI_THREAD (myThread) {     .. code here to run concurrently with
the main program, probably in an         infinite loop }
```

and would be started in the main program with:

```
x = piThreadCreate (myThread) ; if (x != 0)   printf ("it didn't
startn")
```

This is really nothing more than a simplified interface to the Posix threads mechanism that Linux supports. See the manual pages on Posix threads (man pthread) if you need more control over them.

- **piLock (int keyNum) ;**
- **piUnlock (int keyNum) ;**

These allow you to synchronise variable updates from your main program to any threads running in your program. keyNum is a number from 0 to 3 and represents a “key”. When another process tries to lock the same key, it will be stalled until the first process has unlocked the same key.

You may need to use these functions to ensure that you get valid data when exchanging data between your main program and a thread – otherwise it’s possible that the thread could wake-up halfway during your data copy and change the data – so the data you end up copying is incomplete, or invalid. See the wfi.c program in the examples directory for an example.

Serial Library

WiringPi includes a simplified serial port handling library. It can use the on-board serial port, or any USB serial device with no special distinctions between them. You just specify the device name in the initial open function.

To use, you need to make sure your program includes the following file:

```
#include <wiringSerial.h>
```

Then the following functions are available:

- **int serialOpen (char *device, int baud) ;**

This opens and initialises the serial device and sets the baud rate. It sets the port into “raw” mode (character at a time and no translations), and sets the read timeout to 10 seconds. The return value is the file descriptor or -1 for any error, in which case *errno* will be set as appropriate.

- **void serialClose (int fd) ;**

Closes the device identified by the file descriptor given.

- **void serialPutchar (int fd, unsigned char c) ;**

Sends the single byte to the serial device identified by the given file descriptor.

- **void serialPuts (int fd, char *s) ;**

Sends the nul-terminated string to the serial device identified by the given file descriptor.

- **void serialPrintf (int fd, char *message, ...) ;**

Emulates the system printf function to the serial device.

- **int serialDataAvail (int fd) ;**

Returns the number of characters available for reading, or -1 for any error condition, in which case *errno* will be set appropriately.

- **int serialGetchar (int fd) ;**

Returns the next character available on the serial device. This call will block for up to 10 seconds if no data is available (when it will return -1)

- **void serialFlush (int fd) ;**

This discards all data received, or waiting to be send down the given device.

Note: The file descriptor (***fd***) returned is a standard Linux file descriptor. You can use the standard read(), write(), etc. system calls on this file descriptor as required. E.g. you may wish to write a larger block of binary data where the serialPutchar() or serialPuts() function may not be the most appropriate function to use, in which case, you can use write() to send the data.

Advanced Serial Port Control

The ***wiringSerial*** library is intended to provide simplified control – suitable for most applications, however if you need advanced control – e.g. parity control, modem control lines (via a USB adapter, there are none on the Pi's on-board UART!) and so on, then you need to do some of this the "old fashioned" way.

For example – To set the serial line into 7 bit mode plus even parity, you need to do this…

In your program:

```
#include <termios.h>
```

and in a function:

```
struct termios options ;    tcgetattr (fd, &options) ;   // Read current
options   options.c_cflag &= ~CSIZE ;  // Mask out size   options.c_cflag |= CS7 ;
// Or in 7-bits   options.c_cflag |= PARENB ;  // Enable Parity - even by default
tcsetattr (fd, &options) ;   // Set new options
```

The 'fd' variable above is the file descriptor that serialOpen() returns.

Please see the man page for tcgetattr for all the options that you can set.

```
man tcgetattr
```

SPI Library

WiringPi includes a library which can make it easier to use the Raspberry Pi's on-board SPI interface.

Before you can use SPI interface, you may need to use the **gpio** utility to load the SPI drivers into the kernel:

```
gpio load spi
```

If you need a buffer size of greater than 4KB, then you can specify the size (in KB) on the command line:

```
gpio load spi 100
```

will allocate a 100KB buffer. (You should rarely need this though, the default is more than enough for most applications).

To use the SPI library, you need to:

```
#include <wiringPiSPI.h>
```

in your program. Programs need to be linked with **-lwiringPi** as usual.

Functions available:

- **int wiringPiSPISetup (int channel, int speed) ;**

This is the way to initialise a channel (The Pi has 2 channels; 0 and 1). The speed parameter is an integer in the range 500,000 through 32,000,000 and represents the SPI clock speed in Hz.

The returned value is the Linux file-descriptor for the device, or -1 on error. If an error has happened, you may use the standard *errno* global variable to see why.

- **int wiringPiSPIDataRW (int channel, unsigned char *data, int len) ;**

This performs a simultaneous write/read transaction over the selected SPI bus. Data that was in your buffer is overwritten by data returned from the SPI bus.

That's all there is in the helper library. It is possible to do simple read and writes over the SPI bus using the standard read() and write() system calls though – write() may be better to use for sending data to chains of shift registers, or those LED strings where you send RGB triplets of data. Devices such as A/D and D/A converters usually need to perform a concurrent write/read transaction to work.

I2C Library

WiringPi includes a library which can make it easier to use the Raspberry Pi's on-board I2C interface.

Before you can use the I2C interface, you may need to use the **gpio** utility to load the I2C drivers into the kernel:

```
gpio load i2c
```

If you need a baud rate other than the default 100Kbps, then you can supply this on the command-line:

```
gpio load i2c 1000
```

will set the baud rate to 1000Kbps – ie. 1,000,000 bps. (K here is times 1000)

To use the I2C library, you need to:

```
#include <wiringPiI2C.h>
```

in your program. Programs need to be linked with **-lwiringPi** as usual.

You can still use the standard system commands to check the I2C devices, and I recommend you do so – e.g. the **i2cdetect** program. Just remember that on a Rev 1 Raspberry pi it's device 0, and on a Rev. 2 it's device 1. e.g.

```
i2cdetect -y 0 # Rev 1 i2cdetect -y 1 # Rev 2
```

Note that you can use the **gpio** command to run the i2cdetect command for you with the correct parameters for your board revision:

```
gpio i2cdetect
```

is all that's needed.

Functions available

- **int wiringPiI2CSetup (int devId) ;**

This initialises the I2C system with your given device identifier. The ID is the I2C number of the device and you can use the **i2cdetect** program to find this out. wiringPiI2CSetup() will work out which revision Raspberry Pi you have and open the appropriate device in /dev.

The return value is the standard Linux filehandle, or -1 if any error – in which case, you can consult *errno* as usual.

E.g. the popular MCP23017 GPIO expander is usually device Id 0x20, so this is the number you would pass into wiringPiI2CSetup().

For all the following functions, if the return value is negative then an error has happened and you should consult *errno*.

- **int wiringPiI2CRead (int fd) ;**

Simple device read. Some devices present data when you read them without having to do any register transactions.

- **int wiringPiI2CWrite (int fd, int data) ;**

Simple device write. Some devices accept data this way without needing to access any internal registers.

- **int wiringPiI2CWriteReg8 (int fd, int reg, int data) ;**
- **int wiringPiI2CWriteReg16 (int fd, int reg, int data) ;**

These write an 8 or 16-bit data value into the device register indicated.

- **int wiringPiI2CReadReg8 (int fd, int reg) ;**
- **int wiringPiI2CReadReg16 (int fd, int reg) ;**

These read an 8 or 16-bit value from the device register indicated.

Shift Library

WiringPi includes a simple shift library. This allows you to shift 8-bit data values out of the Pi, or into the Pi from devices such as shift-registers (e.g. 74×595) and so-on, although it can also be used in some bit-banging scenarios.

To use, you need to make sure your program includes the following files:

```
#include <wiringPi.h> #include <wiringShift.h>
```

Then the following two functions are avalable:

- **uint8_t shiftIn (uint8_t dPin, uint8_t cPin, uint8_t order) ;**

This shifts an 8-bit data value in with the data appearing on the **dPin** and the clock being sent out on the **cPin**. Order is either **LSBFIRST** or **MSBFIRST**. The data is sampled after the **cPin** goes high. (So **cPin** high, sample data, **cPin** low, repeat for 8 bits) The 8-bit value is returned by the function.

- **void shiftOut (uint8_t dPin, uint8_t cPin, uint8_t order, uint8_t val) ;**

The shifts an 8-bit data value **val** out with the data being sent out on **dPin** and the clock being sent out on the **cPin**. order is as above. Data is clocked out on the rising or falling edge – ie. **dPin** is set, then **cPin** is taken high then low – repeated for the 8 bits.

Software PWM Library

WiringPi includes a software-driven PWM handler capable of outputting a PWM signal on any of the Raspberry Pi's GPIO pins.

There are some limitations… To maintain a low CPU usage, the minimum pulse width is 100µS. That combined with the default suggested range of 100 gives a PWM frequency of 100Hz. You can lower the range to get a higher frequency, at the expense of resolution, or increase to get more resolution, but that will lower the frequency. If you change the pulse-width in the driver code, then be aware that at delays of less than 100µS ***wiringPi*** does it in a software loop, which means that CPU usage will rise dramatically, and controlling more than one pin will be almost impossible.

Also note that while the routines run themselves at a higher and real-time priority, Linux can still affect the accuracy of the generated signal.

However, within these limitations, control of a light/LED or a motor is very achievable.

To use:

```
#include <wiringPi.h> #include <softPwm.h>
```

When compiling your program you must include the *pthread* library as well as the *wiringPi* library:

```
cc -o myprog myprog.c -lwiringPi -lpthread
```

You must initialise ***wiringPi*** with one of *wiringPiSetup(), wiringPiSetupGpio()* or *wiringPiSetupPhys()* functions. *wiringPiSetupSys()* is not fast enough, so you must run your programs with sudo.

Some expansion modules may also be fast enough to handle software PWM – it has been tested with the MCP23S17 GPIO expander on the PiFace for example.

The following two functions are available:

- **int softPwmCreate (int pin, int initialValue, int pwmRange) ;**

This creates a software controlled PWM pin. You can use any GPIO pin and the pin numbering will be that of the ***wiringPiSetup***() function you used. Use 100 for the **pwmRange**, then the value can be anything from 0 (off) to 100 (fully on) for the given pin.

The return value is 0 for success. Anything else and you should check the global *errno* variable to see what went wrong.

- **void softPwmWrite (int pin, int value) ;**

This updates the PWM value on the given pin. The value is checked to be in-range and pins that haven't previously been initialised via softPwmCreate will be silently ignored. Notes

- Each "cycle" of PWM output takes 10mS with the default range value of 100, so trying to change the PWM value more than 100 times a second will be futile.
- Each pin activated in softPWM mode uses approximately 0.5% of the CPU.
- There is currently no way to disable softPWM on a pin while the program in running.
- You need to keep your program running to maintain the PWM output!

Software Tone Library

WiringPi includes a software-driven sound handler capable of outputting a simple tone/square wave signal on any of the Raspberry Pi's GPIO pins.

There are some limitations… To maintain a low CPU usage, the minimum pulse width is 100μS. That gives a maximum frequency of 1/0.0002 = 5000Hz.

Also note that while the routines run themselves at a higher and real-time priority, Linux can still affect the accuracy of the generated tone.

However, within these limitations, simple tones on a high impedance speaker or piezo sounder is possible.

To use:

```
#include <wiringPi.h> #include <softTone.h>
```

When compiling your program you must include the *pthread* library as well as the *wiringPi* library:

```
cc -o myprog myprog.c -lwiringPi -lpthread
```

You must initialise ***wiringPi*** with one of *wiringPiSetup(), wiringPiSetupGpio()* or *wiringPiSetupPhys()* functions. *wiringPiSetupSys()* is not fast enough, so you must run your programs with sudo.

Some expansion modules may also be fast enough to handle software PWM – it has been tested with the MCP23S17 GPIO expander on the PiFace for example.

The following two functions are available:

- **int softToneCreate (int pin) ;**

This creates a software controlled tone pin. You can use any GPIO pin and the pin numbering will be that of the ***wiringPiSetup***() function you used.

The return value is 0 for success. Anything else and you should check the global *errno* variable to see what went wrong.

- **void softToneWrite (int pin, int freq) ;**

This updates the tone frequency value on the given pin. The tone will be played until you set the frequency to 0. Notes

- Each pin activated in softTone mode uses approximately 0.5% of the CPU.
- You need to keep your program running to maintain the sound output!