

Functions (API)

Some of the functions in the *WiringPi* library are designed to mimic those in the Arduino Wiring system. There are relatively easy to use and should present no problems for anyone used to the Arduino system, or C programming in-general.

The main difference is that unlike the Arduino system, the main loop of the program is not provided for you – you need to write it yourself. This is often desirable in a Linux system anyway as it can give you access to command-line arguments and so on. See the [examples](#) page for some simple examples and a Makefile to use.

Before using the *WiringPi* library, you need to include its header file:

```
#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*

Setup Functions

There are three ways to initialise *wiringPi*.

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

One of the *setup* functions must be called at the start of your program. If it returns **-1** then the initialisation of the GPIO has failed, and you should consult the global *errno* to see why.

The differences between the setup functions are as follows:

- **wiringPiSetup(void) ;**

This initialises the wiringPi system 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.

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 need to be called with root priveledges

- **wiringPiSetupSys(void)**

This initialises the wiringPi system 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 number in this mode is the native Broadcom GPIO numbers.

Note: In this mode you can only use the pins which have been exported via the `/sys/class/gpio` interface. You must export these pins before you call your program. You can do this in a separate

shell-script, or by using the `system()` function from inside your program.

Also note that some functions (noted below) have no effect when using this mode as they're not currently possible to action unless called with root privileges.

General wiring functions

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

This sets the mode of a pin to either **INPUT**, **OUTPUT**, or **PWM_OUTPUT**. Note that only *wiringPi* pin 1 (BCM_GPIO 18) supports PWM output. The pin number is the number obtained from the [pins table](#).

This function has no effect when in Sys mode.

- **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.

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

This writes the 8-bit byte supplied to the 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 GPIO hardware.

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

Writes the value to the PWM register for the given pin. The value must be between 0 and 1024. (Again, note that only pin 1 (BCM_GPIO 18) supports PWM)

This function has no effect when in Sys mode (see above)

- **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.

- **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 and 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)

This function has no effect 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.

PWM Control

PWM can not be controlled when running in Sys mode.

- **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.

To understand more about the PWM system, you'll need to read the Broadcom ARM peripherals manual.

Timing functions

- **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.

- **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.

Program/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, (ie. any kernel after about June 2012), 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 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 start\n")
```

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.

Misc. Functions

- **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.

- **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.