**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](https://projects.drogon.net/raspberry-pi/wiringpi/pins/) 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 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.

***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.