**Department of Applied Electronics and Embedded Architectures &**

**Intelligent Systems Operating Systems**

# Laboratory

## Interfacing to the Raspberry Pi Input/Outputs

## 7.1. Introduction

In this laboratory, you will see practical examples that explain how to use general-purpose input/outputs (**GPIOs**) to interface to different types of electronic circuits. The **GPIO** interfacing is performed first using **sysfs**, in order to remember the skills presented in a previous laboratory. Next, memory-mapped approaches are investigated that have impressive performance but are largely specific to the **RPi** platform. Finally, the **wiringPi** library of C functions is discussed in detail.

## 7.2. Equipment Required

To be able to achieve all the laboratory’s goals, it is required to have the following hardware/software components:

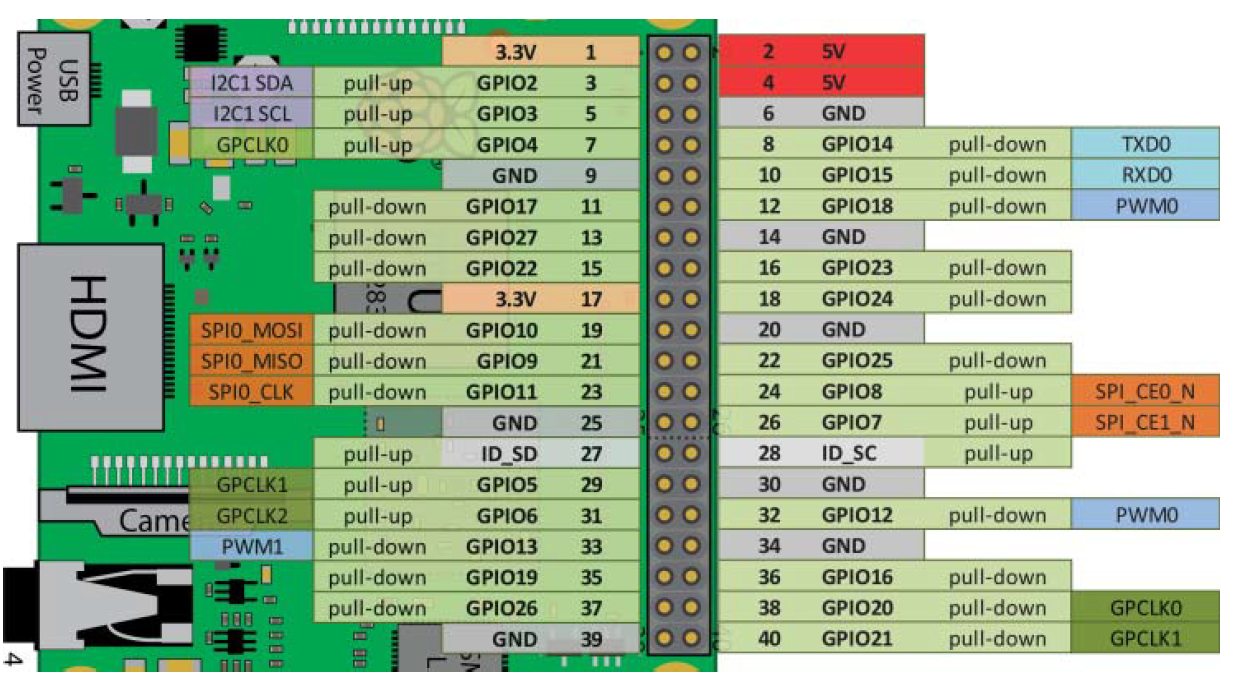
* A **Raspberry Pi** development board with a Raspbian OS;
* Some components: 1 LED, 2 resistors (4K7 & 150 Ω) and a transistor (BC171);
* One push button;
* A breadboard and several hook-up wires.

## 7.3. General-Purpose Input/Outputs

You can interface to the RPi’s GPIO header pins in the following ways:

1. **Digital output:** How you can use a GPIO to turn an electrical circuit on or off. The examples in this laboratory use an LED, but the principles hold for any circuit type; for example, you could even use a relay to turn on/off high powered devices. But, driver circuits are required to ensure that you do not draw too much current from a GPIO.
2. **Digital input:** ensure you can read in a digital output from an electrical circuit into a software application running under Linux.
3. **Analog output:** you can use PWM to output a proportional signal that can be used as an analog voltage level or as a control signal for certain types of devices, such as servo motors.

Although there are 54 GPIO lines in the Raspberry Pi’s processor, only 26 are brought out to the P1 connector on the board (less on the non-plus models); the other pins are used for actually making the processor act like a computer (things like the SD card, USB connector, and LEDs).



**Figure 7.1.** 40 pins **RPi** header

**Figure 7.1** provides you with a first view of the functionality of the inputs and outputs on the GPIO header. Many of these pins are multiplexed, meaning they have more functions (or ALT modes) than what is displayed in the figure. This figure illustrates the most commonly used functionality.

The 26 GPIO pins can be individually configured as inputs or outputs, and use 3.3V logic levels where a HIGH logic level is represented as nominally 3.3 volts and a LOW by zero volts. **They cannot tolerate higher voltages (like 5 volts) without being damaged.**

## 7.4. Internal Pull-Up and Pull-Down Resistors

Each pin of the **RPi** has pull-up and pull-down resistors. These pull-up and pull-down resistors can be external or internal. Such external resistors are typically “strong” pull-up/down resistors in that they “strongly” tie the input to a high/low value using relatively small resistance values (e.g., 5 kΩ–10 kΩ or maybe less). The **RPi** has “weak” internal pull-up and internal pull-down resistors that can be configured using memory-based GPIO control techniques, see **Figure 7.2**. You need to factor these resistor values into the behavior of your input/output circuits, and you need to be able to alter the internal resistor configuration in certain circumstances. For example, you may even want to turn them off for a particular circuit.

**Practical exercise:** Interestingly, if you connect nothing to GPIO4, which is Pin 7 when you read it, a value of 1 will be returned. That is because this input is connected via an internal pull-up resistor to the 3.3V line. Whereas GPIO27 is low when it is interrogated with nothing connected to it. Please check all of these.

pi@raspberrypi: /sys/class/gpio $ echo 4 > export

pi@raspberrypi: /sys/class/gpio $ cd gpio4

pi@raspberrypi: /sys/class/gpio/gpio4 $ cat direction

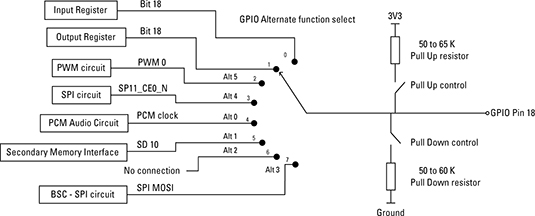
out

pi@raspberrypi: /sys/class/gpio/gpio4 $ echo in > direction

pi@raspberrypi: /sys/class/gpio/gpio4 $ cat value

1

It should be clear at this stage that you need to understand the GPIO configuration, including these internal resistors, to use the GPIO pins properly. ***In the default configuration, e.g. after a reset, pull-up and pull-down resistors are connected accordingly with* Figure 7.1**.



**Figure 7.2.** GPIO18 alternate functions

Also, note that Pin 3 (GPIO2 - SDA) and Pin 5 (GPIO3 - SCL) have two permanent onboard 1.8 kΩ “strong” pull-up resistors attached to the PCB.

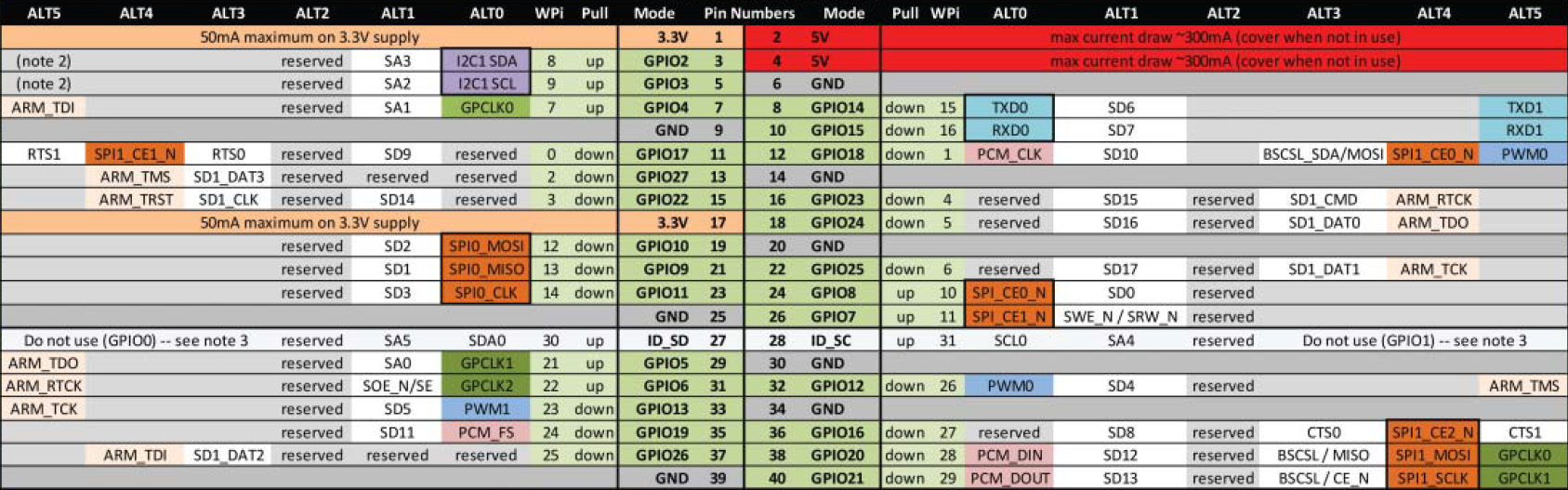
As well as configuring pins to have either a pull-up or a pull-down resistor configuration, there are also different modes for each pin. This is called the ALT mode for the pin. **Figure 7.3** provides a full list of alternative ways for each of the GPIO header pins.

You can see the basic arrangement for one pin, GPIO 18 in **Figure 7.2**. The general-purpose input/output (GPIO) pins can be switched between input or output and have a pull-up, or pull-down resistor enabled, but there are a host of other peripherals in the Raspberry Pi chip that can be switched to these pins. So, all the other pins have a similar arrangement but with different blocks to select from. The numbers on the switch are the three-bit register value that has to be set in the collection of alternate function select registers.

The first thing to spot in **Figure 7.3** is that there are two types of “nothing here”: **(a)** one is blank and **(b)** the other is labeled as “reserved”. It’s likely that these reserved functions are used for factory testing of the chip or for purposes not disclosed in the datasheet. The blank ones are simply not implemented.

For the full story, the BCM2835 ARM Peripherals document (<http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf>) is where you want to look, but here’s a quick look at some of the functions:

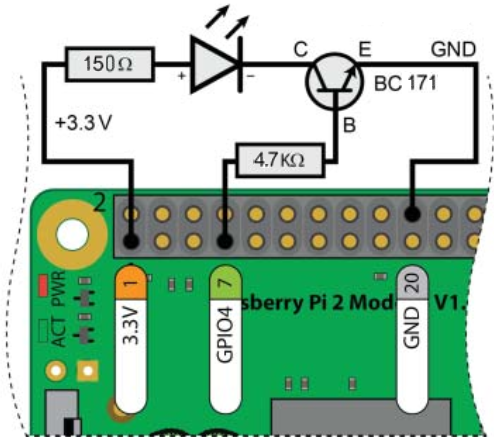
1. **ALT 0:** Where most of the interesting and useful alternate functions are as far as the **Raspberry Pi** is concerned. The **SDA** and **SCL** 0 and 1 are the two I2C buses, and the **TXD0** and **RXD0** are the serial connections. The **GPCLK** lines are a general-purpose clock output that can be set to run at a fixed frequency independent of any software. The **PWM** (pulse-width modulation is supported in hardware on pins GPIO12, GPIO13) pins provide the two pulse-width modulated outputs; the **SPI** 0 is the serial peripheral interface bus lines. Finally, the **PCM** pins provide pulse code modulated audio outputs.
2. **ALT 1:** The pins are used as a secondary memory bus. Due to the design of the Raspberry Pi, this is of no use at all.
3. **ALT 2:** The only ALT 2 pins brought out to the GPIO pin header are reserved.
4. **ALT 3:** The most useful pins here are the **CTS0** and **RTS0** lines; these are handshaking lines for the serial module if you need them. The **BSC** lines are for the Broadcom Serial Controller, which is a fast mode I2C-compliant bus supporting 7-bit and 10-bit addressing and having the timing controlled by internal registers. The SD1 lines are probably for the control of an SD card.
5. **ALT 4:** The **SPI** **1** lines are a second **SPI** bus. And the **ARM** pins are for a **JTAG** interface. **JTAG** is a way of talking to the chip without any software on it. It’s very much used for the initial tests on a system during development, although it can be used for hardware debugging as well.



**Figure 7.3.** The **RPi** alternate functions

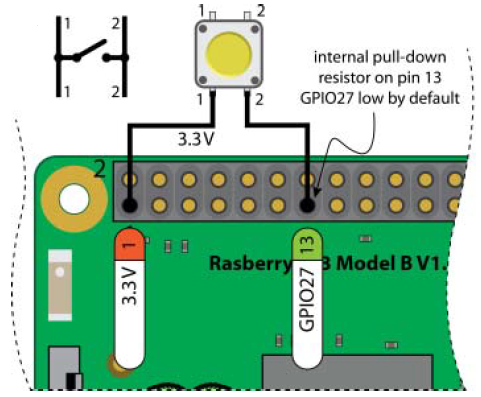
1. **ALT 5:** The useful pins here are the secondary serial port data and handshaking lines. The **PWM** lines are precisely the same **PWM** lines that are switched to **GPIO12** and **13** under **ALT** 0. There are also two of the general-purpose clock lines along with another copy of the ARM JTAG signals.

## 7.5. The hardware part

**Figure 7.4** is presented a circuit based on which you can connect to the **RPi** in a safe way to drive an LED, using a BC 171 component – an NPN transistor. This circuit is all you need to test all the code that is presented in this laboratory. In this example the **GPIO** pin (**GPIO4**), which is connected to Pin 7 on the **GPIO** header, see **Figure 7.4**, provides the low current required to switch the transistor on or off, depending on whether the **GPIO** state is high or low.

**Figure 7.4.** Driving a LED by a GPIO pin using an NPN transistor

**WARNING:** Be very careful when wiring circuits such as those in **Figure 7.4**. Incorrect connections or the use of the wrong header pin can destroy your RPi. **It is mandatory, to wire such circuits with the power to the RPi disconnected**. Only power up the **RPi** once you have carefully checked the circuit configuration.



**Figure 7.5.** Connecting a pushbutton to the **RPi** using the internal pull-down resistor

Please build, also, the circuit presented in **Figure 7.5**. The circuit shown in **Figure 7.5** consists of a normally open push button that is connected to the **RPi** **Pin 13** (**GPIO27**). You can notice that in a normal situation, a pull-up or pull-down resistors on pushbutton switches are required, but none are present in this circuit. This is not accidental, because **Pin 13** on the **GPIO** header is connected by default to GND using an internal pull-down resistor.

The LED circuit, presented in **Figure 7.4**, should be left connected when building this input circuit (shown in **Figure 7.5**) because both circuits are used throughout this laboratory.

## 7.6. Memory-Based GPIO Control

Based on the full datasheet for the Broadcom BCM2835 Peripherals is possible to use such low-level I/O detail to bypass the Linux kernel, using direct memory manipulation to take control of the SoC’s inputs and outputs. While this approach can achieve much better I/O performance, you should avoid using it if possible, because your programs will not be portable to other embedded Linux platforms. Besides, since the Linux kernel is unaware of such direct memory manipulations, you could potentially generate resource conflicts.

Linux uses a virtual memory system, which means that there is a difference between the physical address used by the hardware and the virtual address that is used to access the device. In 32-bit Linux the virtual memory system utilizes the full 32-bit addressing to allocate a virtual space that is much larger than the available physical memory; 32-bit addressing supports 232 addresses (i.e., 4 GB), whereas there is 1 GB of RAM available on the **RPi 3**. The extended address range allows for memory paging and the mapping of physical devices (e.g., peripherals) into a unified address space.

The GPIO peripheral base address on the RPi 2/3 is 0x3f20 0000.

### GPIO Control Using devmem2

You can query the value at a memory address using C code that accesses **/dev/mem** directly. However, to become familiar with the steps, it is best that you build and install Jan-Derk Bakker’s **devmem2** program, which is a handy command-line tool for reading from and writing to memory locations:

pi@raspberrypi:~ $ wget http://www.lartmaker.nl/lartware/port/devmem2.c

pi@raspberrypi:~ $ gcc devmem2.c -o devmem2

pi@raspberrypi:~ $ ./devmem2

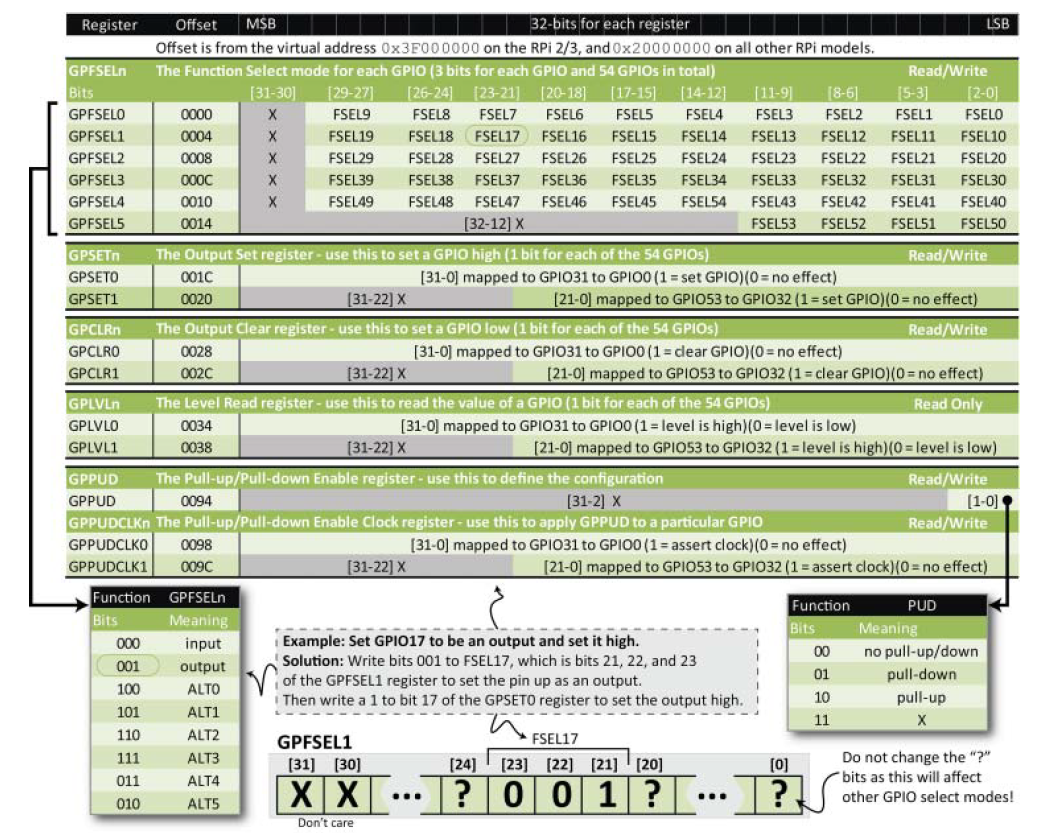
pi@raspberrypi:/sys/class/gpio $ ls

Usage: ./devmem2 { address } [ type [ data ] ]

address : memory address to act upon

type : access operation type: [b]yte, [h]alfword, [w]ord

data : data to be written

The registers that are important for GPIO control are described in **Figure 7.6**.

**Figure 7.6.** Examples of the registers available for memory-mapped GPIO manipulation

If the circuit is connected as in **Figure 7.4**, it is possible to use the devmem2 program to control the **LED** circuit. Assuming that the **devmem2** program is currently present in the pi user home directory, you can use it to read the value of the **GPLVL0** register on the **RPi**:

pi@raspberrypi: /sys/class/gpio $ echo 4 > export

pi@raspberrypi: /sys/class/gpio $ cd gpio4

pi@raspberrypi: /sys/class/gpio/gpio4 $ echo out > direction

pi@raspberrypi: /sys/class/gpio/gpio4 $ cat value

0

pi@raspberrypi: /sys/class/gpio/gpio4 $ sudo ~/devmem2 0x3F200034

/dev/mem opened.

Memory mapped at address 0x76f0e000.

Value at address 0x3F200034 (0x76f0e034): 0xC1EF

pi@raspberrypi: /sys/class/gpio/gpio4 $ echo 1 > value

pi@raspberrypi: /sys/class/gpio/gpio4 $ sudo ~/devmem2 0x3F200034

/dev/mem opened.

Memory mapped at address 0x76f0e000.

Value at address 0x3F200034 (0x76f0e034): 0xC1FF

Notice that the difference is 0x10, which is 10000 in binary (i.e., 1 followed by 4 zeros, or 1<<4). **GPIO4** is in the first bank of addresses. For **GPIO32** to **GPIO53**, you have to read the **GPLVL1** register. The output above indicates that the output is low the first time that the **GPLVL0** register is displayed, and high the second time.

You can use the same **devmem2** program to set the LED to be off by setting bit 4 on the **GPCLR0** (0028) register, and the **LED** to be on by setting bit 4 on the **GPSET0** (001C) register:

pi@raspberrypi: /sys/class/gpio/gpio4 $ cat value

1

pi@raspberrypi: /sys/class/gpio/gpio4 $ sudo ~/devmem2 0x3F200028 w 0x10

/dev/mem opened.

Memory mapped at address 0x76fd6000.

Value at address 0x3F200028 (0x76fd6028): 0x6770696F

Written 0x10; readback 0x6770696F

pi@raspberrypi: /sys/class/gpio/gpio4 $ cat value

0

pi@raspberrypi: /sys/class/gpio/gpio4 $ sudo ~/devmem2 0x3F20001C w 0x10

/dev/mem opened.

Memory mapped at address 0x76f10000.

Value at address 0x3F20001C (0x76f1001c): 0x0x6770696F

Written 0x10; readback 0x6770696F

pi@raspberrypi: /sys/class/gpio/gpio4 $ cat value

1

Here you can see that setting these bits has a direct impact on the **sysfs** value entry for the GPIO4 entry.

**Homework:** Move the B (base) connection of the BC171 transistor through 4.7K resistor at the **GPIO10** - pin 19 of the **RPi** header, see **Figure 7.1**. Now turn on and off the LED.

### GPIO Control Using C and /dev/mem

**Figure 7.6** also provides details and an example of how to configure the mode of a pin. All of the GPIOs can be set to read or write mode, or they can be set to an ALT mode, which is listed in **Figure 7.3**. The mode is set using a 3-bit value as listed in the table on the bottom left of **Figure 7.6**. For example, by setting the 3-bit value to be 000 then the pin will act as an input.

The location to which you should write the 3-bit mode is described in **Figure 7.6**. For example, to set **GPIO17** to be an output, you can write 001 to the FSEL17 value, which is bits 21, 22, and 23 in the **GPFSEL1** register. Importantly, you need to ensure that you only manipulate those specific 3 bits when you change **FSEL17**, because to change any other bits will impact on **GPIO10**–**GPIO19**, likely changing their pin modes. Below is a listing that provides a C code example that sets up GPIO4 as output and flashes an LED. It also sets up GPIO27 as an input, so that the LED will continue to flash until a pushbutton is pressed. The comments in the code listing describe the bit manipulations that are used.

In a previous laboratory I told you that the **NetBeans IDE** will be the default development tools for this discipline. But, the following program requires to be run as a root in order to work properly – this is a little bit difficult from **NetBeans IDE**. So, as result please develop the program directly on the **Rasberry Pi** platform. After the built of the program, executed it using the **sudo** tool.

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <errno.h>

#include <sys/mman.h>

#include <stdint.h> // for uint32\_t - 32-bit unsigned integer

// GPIO\_BASE is 0x20000000 on RPi models other than the RPi 2/3

#define GPIO\_BASE 0x3F200000 // on the RPi 2/3

#define GPSET0 0x1c // output set register ----------- from **Figure 7.6**

#define GPCLR0 0x28 // output clear register

#define GPLVL0 0x34 // read register

static volatile uint32\_t \*gpio; // pointer to the gpio (\*int)

int main ( int argc, char\*\* argv )

{

int fd, x;

printf ("Start of GPIO memory-manipulation test program.\n");

if ( getuid() != 0 )

{

printf("You must run this program as root. Exiting.\n");

return -1;

}

if ( (fd = open("/dev/mem", O\_RDWR | O\_SYNC)) < 0 )

{

printf("Unable to open /dev/mem: %s\n", strerror(errno));

return -2; //busy

}

// get a pointer that points to the peripheral base for the GPIOs

gpio = (uint32\_t \*) mmap (0, getpagesize(), PROT\_READ|PROT\_WRITE,

MAP\_SHARED, fd, GPIO\_BASE);

if ( (int32\_t) gpio < 0 )

{

Printf ("Memory mapping failed: %s\n", strerror(errno));

return -2; //busy

}

// Here the gpio pointer points to the GPIO peripheral base address.

// Set up the LED GPIO FSEL4 mode = 001 at addr GPFSEL0 (0000).

// Writing NOT 7 (i.e., ~111) clears bits 12, 13 and 14.

\*gpio = (\*gpio & ~(7 << 12) | (1 << 12));

// Set up the button GPIO FSEL27 mode = 000 at addr GPFSEL2 (0008).

// Remember that adding two 32-bit value moves the addr by 8 bytes.

\*(gpio + 2) = (\*(gpio + 2) & ~(7 << 21));

// Writing the 000 is not necessary but is there for clarity.

do {

// Turn the LED on using bit 4 on the GPSET0 register

\*(gpio + (GPSET0/4)) = 1 << 4;

usleep(50000);

\*(gpio + (GPCLR0/4)) = 1 << 4; // turn the LED off

usleep(50000);

} while((\*(gpio+(GPLVL0/4))&(1<<27))==0); // only true if bit 27 high

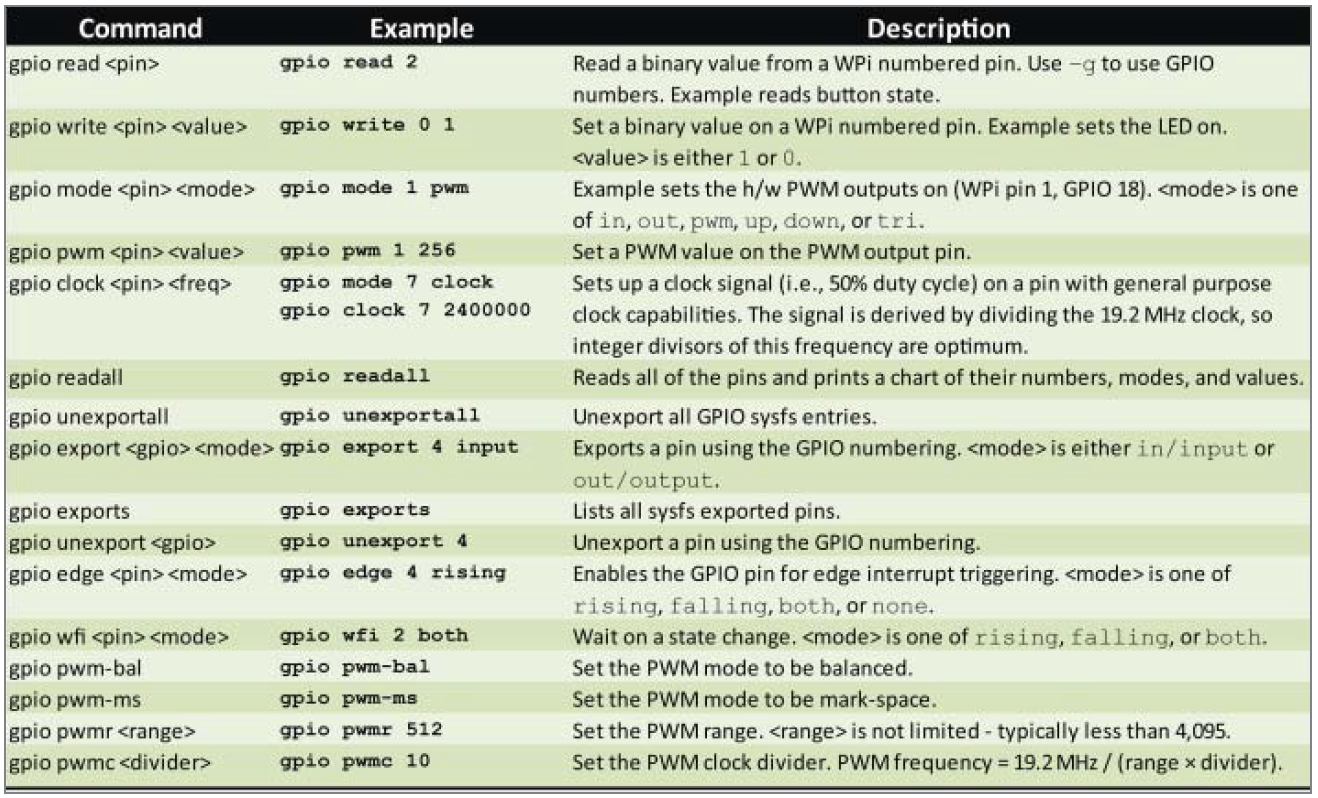
printf("Button was pressed - end of example program.\n");

close(fd);

return (EXIT\_SUCCESS);

}

**Homework:** Re-design the program, previously presented, so that the blinking of the LED will start after the button is pressed.



**Figure 7.7.** Some of the **gpio** command options

## 7.7. WiringPi

**The WiringPi** is a pin-based **GPIO** access library written in C for the BCM2835, BCM2836 and BCM2837 SoC devices used in all **Raspberry Pi** versions. So, **WiringPi** ([www.wiringpi.com](http://www.wiringpi.com)) is an extensive **GPIO** control library for the **RPi** platform. The library function syntax is similar to that in the Arduino Wiring library, and it is a popular choice among **RPi** users. The **wiringPi** library also has third-party bindings for **Python**, **Ruby**, and **Perl**.

**WiringPi** utilizes the **sysfs** and memory-mapped techniques described in this laboratory and in a previous one, to create a highly efficient library and command set that has been custom developed for the **RPi** platform. It is recommended that you use this library for controlling the **GPIOs** on the **RPi** when fast **GPIO** switching is required; however, be aware that this approach is mostly specific to the **RPi** platform and not to embedded Linux devices in general.

### Installing wiringPi

To install the **wiringPi** do as follows:

pi@raspberrypi:~ $ sudo apt-get install wiringpi

In most cases, the **wiringPi is** pre-installedwith standard **Raspbian** systems, so it is not required to install once again.

### The gpio Command

**Installed as part of the wiringPi build, the gpio program is a handy command-line tool for accessing and controlling the GPIOs on the RPi. Figure 7.7 provides a summary of some of the available commands, along with some example usage.**

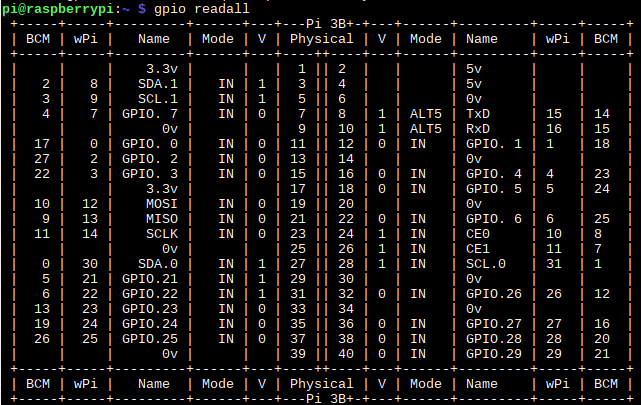
**For historical reasons, wiringPi tends use a different numbering scheme than the physical pin number or GPIO number. These numbers are displayed in the WPi column in Figure 7.3. However, many gpio commands can also accept regular GPIO numbering by using a -g option. So, you can use the gpio command to write Linux scripts to control the GPIOs.**

**To check the version of the wiringPi use:**

pi@raspberrypi:~ $ gpio -v

**To reads all the normally accessible pins, prints a table of their numbers along with their modes and current values you can use:**

pi@raspberrypi:~ $ gpio readall

**This command will detect the version/model of your RPi and printout the pin diagram appropriate to your Pi, as it is presented in Figure 7.8.**

**Figure 7.8.** The results of the **gpio readall**

**This command will detect the version/model of your RPi and will printout the pin diagram appropriate to your RPi system.**

**In Figure 7.8, the BCM is the pin number when using GPIO (aka BCM) numbering scheme, and wPi is the pin number when using the wiringPi's own pin numbering scheme. So when using wiringPi, if you choose wiringPiSetup(), it will default to using the wPi pin numbers, and wiringPiSetupGpio() will result in using the GPIO/BCM numbers. In the “Physical” column are** the pin numbers on the connector. The “V” column indicates the voltage level of the pins. As you may notice, the “0/1” value is applicable for GPIO pins only, and leave blank for the rest. V = 0 indicates the voltage level on the pin is LOW; the exact voltage value is 0V. In contrast, V = 1 indicates the voltage level on the pin is HIGH, the voltage value is around 3.3V.

**The “Mode” shows how the pin is being used. Here is the list of valid modes able to set for GPIO: in/out/pwm/clock/up/down/tri. However, you also see the “ALT[#]” listed in the Mode column. This ALT[#] indicates the GPIO pin is being used as alternative function – a specific function, not as general purpose function.**

* **in**: the GPIO pin is used as input without software-register-pull-up or software-register-pull-down. In this mode, if the pin is not connected to any reference point, the input state will be unknown (it is floated).
* **out**: the GPIO pin is used as an output, and without software-register-pull-up or software-register-pull-down. In this mode, if the pin is not connected to any reference point, the input state will be unknown (it is floated).
* **pwm**: the GPIO pin is used as hardware Pulse-Width-Modulation mode which is square waveform generation. This mode is usually used to control DC motor speed or LED dimming. Keep in mind that only BCM 18 (or GPIO.1) supports PWM output mode.
* **clock**: the GPIO pin is used as a clock generator. The frequency of clock is driven from main crystal frequency of Raspberry Pi. Note that only BCM 4 (or GPIO.7) supports CLOCK output mode.
* **up**: The GPIO pin is used as an input with software-register-pull-up. In this mode, if the pin is not connected to any reference point, the input state will be HIGH (or 1).
* **down**: The GPIO pin is used as an input with software-register-pull-down. In this mode, if the pin is not connected to any reference point, the input state will be LOW (or 0).
* **tri**: The GPIO pin is used as input without software-register-pull-down or up. In this mode, if the pin is not connected to any reference point, the input state will be LOW or HIGH.
* **ALT[#]**: The GPIO pin is used as a special function. You can check the list of those special functions at this link.

### Programming with wiringPi

**WiringPi contains a comprehensive library of C functions for controlling RPi GPIOs, regardless of the board model. The following listing provides a first wiringPi program that displays information about the board that you are using. Again, it is assumed for these examples that the board is connected to the LED and Button circuits, as illustrated in Figure 7.4 and Figure 7.5. Mainly because the wiringPi library is used, the linker requires –lwiringPi option.**

**Use NetBeans to implement and run the following program:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <wiringPi.h>**

**#define LED\_GPIO 4 // this is GPIO4, Pin 7**

**#define BUTTON\_GPIO 27 // this is GPIO27, Pin 13**

**int main(int argc, char\*\* argv)**

**{**

**wiringPiSetupGpio(); // use the GPIO numbering form**

**pinMode(LED\_GPIO, OUTPUT); // the LED set up as an output**

**pinMode(BUTTON\_GPIO, INPUT); // the Button set up as an input**

**int model, rev, mem, maker, overVolted;**

**piBoardId(&model, &rev, &mem, &maker, &overVolted);**

**printf ("\n This is an RPi: %s", piModelNames[model]);**

**printf ("\n with revision number: %s", piRevisionNames[rev]);**

**printf ("\n manufactured by: %s", piMakerNames[maker]);**

**printf ("\n it has: %d RAM and o/v: %d", mem, overVolted);**

**printf ("\n\n Button GPIO has ALT mode: %d", getAlt(BUTTON\_GPIO));**

**printf ("\n and value: %d", digitalRead(BUTTON\_GPIO));**

**printf ("\n\n LED GPIO has ALT mode: %d", getAlt(LED\_GPIO));**

**printf ("\n and value: %d", digitalRead(LED\_GPIO));**

**return (EXIT\_SUCCESS);**

**}**

**This code can be built using gcc by linking to the wiringPi library, based on -lwiringPi that explicitly links to libwiringPi.so.**

### Toggling an LED Using wiringPi

**The following listing provides a code example for toggling a GPIO.**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <wiringPi.h>**

**#define LED\_GPIO 4 // this is GPIO4, Pin 7**

**int main(int argc, char\*\* argv)**

**{**

**wiringPiSetupGpio(); // use GPIO, not WPi, labels**

**printf ( "Starting fast GPIO toggle on GPIO%d", LED\_GPIO );**

**printf ( "Press CTRL+C to quit..." );**

**pinMode(LED\_GPIO, OUTPUT); // GPIO4 is an output pin**

**while(1)**

**{ // loop forever - await ^C press**

**digitalWrite(LED\_GPIO, HIGH); // LED on**

**usleep(500000);**

**digitalWrite(LED\_GPIO, LOW); // LED off**

**usleep(500000);**

**} // the duty cycle somewhat**

**return 0; // program will never reach here!**

**}**

Develop a new program so that the LED goes on and on only when the user presses the button.

**Homework:** Develop a new program so that the LED will be on all the time and will go off only when the user presses the button. Use for this **digitalRead(BUTTON\_GPIO**) function.