


F28HS Hardware-Software Interface: Systems Programming

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh



Semester 2 2017/18

⁰No proprietary software has been used in producing these slides 



Outline

- 1 Tutorial 1: Using Python and the Linux FS for GPIO Control
- 2 Tutorial 2: Programming an LED
- 3 Tutorial 3: Programming a Button input device
- 4 Tutorial 4: Inline Assembler with gcc
- 5 Tutorial 5: Programming an LCD Display
- 6 Tutorial 6: Performance Counters on the RPi 2

Tutorial 1: Using Python for GPIO Control

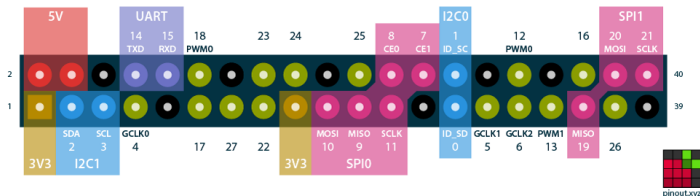
- In this first tutorial we will get started with programming the RPi2 to control **output** devices.
- We will use **Python** as programming language and existing libraries for controlling the GPIO pins on the RPi2, which simplifies the programming considerably.
- The main learning objective for this course, however, is to achieve such control by using C and Assembler, and we will focus on these languages for the remaining tutorials.

GPIO pins of the RPi2

- The Raspberry Pi 2 has 40 **General Purpose Input/Output (GPIO)** pins.
- These can be used to control a range of devices, or to receive data from such devices.
- You need to use the jumper cables in the Raspberry Pi2 starter kit to connect devices.
- In this first tutorial we will attach an LED and make it blink

Map of the GPIO pins of a RPi2

Raspberry Pi GPIO BCM numbering



⁰Available from <http://pinout.xyz/>

Electronics basics and wiring diagrams

For a good, introductory discussion on how to wire-up external devices to the Raspberry Pi 2 [follow this link](#).

You can get a small [CamJam EduKit](#), including LEDs, button, resistors and jumper cables, from ThePiHut. These are all include in your RPi2 starter kit, so you don't need these, but they may be useful for experimentation.

The following slides summarise the main steps from this web page.

Connecting an LED to the RPi2

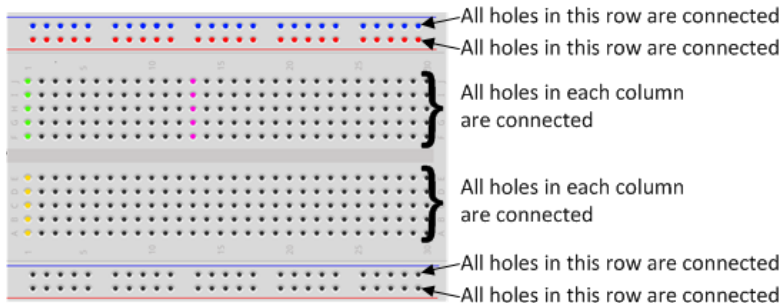
As the first exercise in controlling an external LED we need:

- a Breadboard
- an LED
- a 330 ohm resistor
- two Male-Female jumper wires

⁰For details see [this page](#)

How to use a Breadboard

The breadboard is a way of connecting electronic components to each other without having to solder them together.



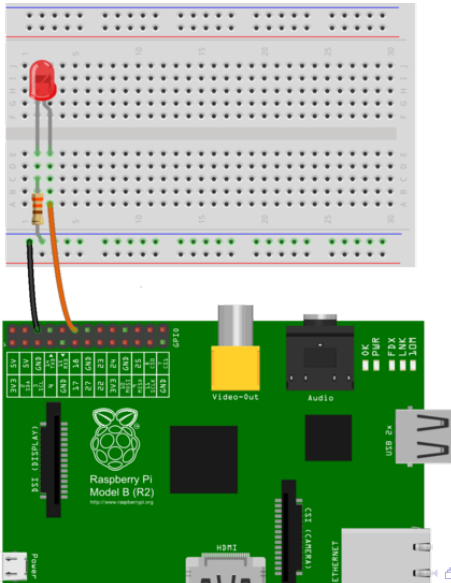
Using a breadboard, like the one above, simplifies the wiring, especially for larger projects (as in CW2).

External devices: LED and Resistor

- In this tutorial, we only want to connect an LED to the RPi2, using a breadboard.
- Note that the LED has two legs of different lengths.
- The **longer leg**, is always connected to the **positive** supply of the circuit.
- The **shorter leg** is connected to the **negative** side of the power supply, known as 'ground'.
- You **must** protect the LED with a resistor, otherwise the LED will try to draw more power than needed and **might burn out the RPi2**.
- Putting the resistors in the circuit will ensure that only this small current will flow and the Pi will not be damaged.



Fritzing Diagrams



The Fritzing Diagram Explained

The Fritzing diagram on the previous slide shows how to wire-up external devices, i.e. which pins to connect to which rows/columns on the breadboard to complete a circuit.

- Use one of the jumper wires to connect a **ground pin** to the rail, marked with **blue**, on the breadboard. The female end goes on the Pi's pin, and the male end goes into a hole on the breadboard.
- Then connect the **resistor** from the same row on the breadboard to a column on the breadboard, as shown in the diagram.
- Next, push the **LEDs** legs into the breadboard, with the long leg (with the kink) on the right.
- Lastly, complete the circuit by connecting **pin 18** to the **right hand leg of the LED**. This is shown here with the orange wire.

Using Python to control a GPIO pin

Details of the software setup can be found in Chapter 8 of “Adventures in Raspberry Pi”.

- First install the Python library for GPIO support:

```
> sudo apt-get install python-RPi.GPIO
```

- To test the version of the RPi you have do the following:

```
>>> import RPi.GPIO as GPIO
>>> GPIO.RPI_REVISION
2
```

Python code to control a GPIO pin

First we import all required libraries and set some constants, in particular the pin number that we use for the LED. We need to specify which numbering of the pins to use, and then setup the connection.

```
#!/usr/bin/python

# External module imports
import RPi.GPIO as GPIO
import time

# Pin Definitons:
ledPin = 23 # Broadcom pin 23 (P1 pin 16)

# Pin Setup:
GPIO.setmode(GPIO.BCM) # Broadcom pin-numbering scheme
GPIO.setup(ledPin, GPIO.OUT) # LED pin set as output

# Initial state for LEDs:
GPIO.output(ledPin, GPIO.LOW)
```

Python code to control a GPIO pin (cont'd)

The main part of the program is the loop below, which continuously turns the LED on and off, using a delay of 75ms:

```
while True:
    try:
        GPIO.output(ledPin, GPIO.HIGH)
        time.sleep(0.075)
        GPIO.output(ledPin, GPIO.LOW)
        time.sleep(0.075)
    except KeyboardInterrupt: # If CTRL+C is pressed, exit
        cleanly:
            GPIO.cleanup() # cleanup all GPIO
```

Tutorial 1: Using the Linux FS for GPIO Control

- One design principle in Linux is to control and view system information through files.
- We have seen this in class by, e.g. looking up details about the CPU by `cat /proc/cpuinfo`
- This tutorial will demonstrate **how filesystem operations can be used to easily control GPIO pins on the RPi 2**

NB: You need a Linux kernel with support for SysFS. Raspbian 7, as we use it in the kit handed out for this course, provides this.

To check whether SysFS is supported do:

```
> sudo sh -c "cat /lib/modules/`uname -r`/build/.config |  
fgrep SYSFS"
```

and look for a line like this

```
> CONFIG_SYSFS=y
```

Basics of SysFS

- The Linux kernel provides several RAM based file systems.
- These file systems provide low-level hardware information and in some cases a way to control these.
- The basic programmer API is to use the system-level `read(2)` and `write(2)` commands on the files in these file systems.
- Each file has a special meaning to enable hardware interaction.
- The `read` and `write` function calls, result in callbacks in the Linux kernel which has access to the corresponding value.
- The benefit of using the `read` and `write` functions is that the user space has a lot of tools available to send data to the kernel space (e.g. `cat(1)`, `echo(1)`).

⁰From: [Kernel Space - User Space Interfaces](#)

SysFS filesystem

- `SysFS` was designed to represent the whole device model as seen from the Linux kernel
- It contains information about devices, drivers and buses and their interconnections.
- `SysFS` is heavily structured and contains a lot of links.

The main subdirectories of interest for us are:

- `sys/block/` all known block devices such as `hda/` `ram/` `sda/`
- **`sys/class/`** for each device type there is a subdirectory: for example `/printer` or `/sound`
- `sys/device/` all devices known by the kernel, organised by the bus they are connected to

Controlling GPIO pins using SysFS

- We want to control the GPIO pins on the RPi2 using the SysFS interface.
- To this end we need to:
 - ▶ tell the system that we want to access a specific pin (“export” this device to the SysFS)
 - ▶ configure the mode of the pin, as either `in` or `out`
 - ▶ read/write from/to the device using standard tools such as `echo` and `cat`
 - ▶ finally, remove the device from the filesystem (“unexport” it from SysFS)
- All of these steps can be done as one-liners from the command-line
- No additional libraries need to be installed

NB: This interface is useful for **testing** a wiring or debugging the hardware. The main learning objective of the course is to learn how to do the above operations directly on the device (in C or Assembler), without involving the operating system or an external library at all.

The Shell Code for Controlling a GPIO

```
# the pin to control
PIN=23

# make this pin available through the SysFS
echo $PIN > /sys/class/gpio/export

# now, set this pin to output
echo out > /sys/class/gpio/gpio${PIN}/direction

# write a value to this pin
echo 1 > /sys/class/gpio/gpio${PIN}/value

# wait for some seconds
sleep 3s

# write a value to this pin
echo 0 > /sys/class/gpio/gpio${PIN}/value

# make this pin unavailable through the SysFS
echo $PIN > /sys/class/gpio/unexport
```

NB: You need to run this as root, i.e. type `sudo sh sysfs_23.sh`

NB: Version with a pin as param: `sudo sh sysfs.sh -p 23`



Other useful information in the SysFS

You can get the information about the model like this:

```
> cat /sys/firmware/devicetree/base/model
```

You can get the information about the cache line size like this (not enabled under Raspbian by default):

```
> cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
```

On Debian-based systems, such as Ubuntu, you can also get this info by typing:

```
> getconf -a
```

Useful information in the ProcFS filesystem

The `/proc` filesystem provides information about the **processor**:

```
> cat /proc/cpuinfo
```

gives detailed information about the processor, split by core, eg. each core is an `ARMv7 Processor` and the `neon` instruction set is enabled. Detailed information about the memory is available via:

```
> cat /proc/meminfo
```

shows that the total memory is `949408 kB`, i. e. ca. 1GB.

```
> cat /proc/iomem
```

shows the structure of the memory, including the location of the GPIO memory.

The ProcFS filesystem

There is a special subdirectory: `/proc/sys`. It allows to configure a lot of parameters of the running system.

```
> cat /proc/sys/kernel/osrelease
```

tells us that the kernel version is `3.18.11-v7+`.

There are a lot of files in this directory, showing the current state of the kernel. For interacting with the kernel, the `sysctl` interface should be used.

The `sysctl` infrastructure is designed to configure kernel parameters at run time. E. g.

```
> sysctl --all
```

lists all kernel parameters.

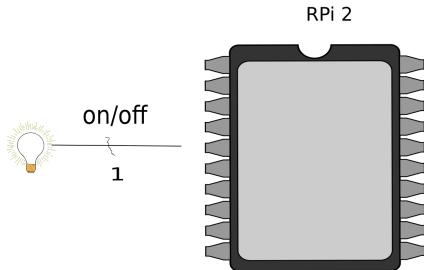
Further Reading & Deeper Hacking

- [Kernel Space - User Space Interfaces](#), Ariane Keller
- [ProcFS Kernel Docu](#)
- [Linux Device Drivers](#), 3rd ed, Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman
- [More detailed documentation on SysFS](#)
- [Shell code samples with SysFS](#)

Tutorial 2: Programming an LED

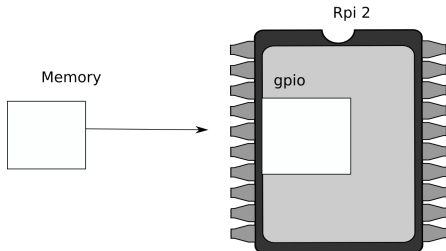
- This tutorial will deal with programming an LED output device.
- This is the “hello world” program for external devices.
- It will deal with programming techniques common to other output devices.
- The **learning objective** of this exercise is to learn how to directly control an external device through C and Assembler programs.
- We will also cover easier ways of external control, however these should only be used to test your hardware/software configuration and don't replace the programming component.

The high-level picture



- From the main chip of the RPi2 we want to control an (external) device, here an LED.
- We use one of the **GPIO** pins to connect the device.
- Logically we want to send **1 bit** to this device to turn it **on/off**.

The low-level picture



Programmatically we achieve that, by

- memory-mapping the address space of the GPIOs into user-space
- now, we can directly access the device via memory read/writes
- we need to pick-up the meaning of the peripheral registers from the BCM2835 peripherals sheet

BCM2835 GPIO Peripherals

Base address: 0x3F000000

0		Pins 0-9	
5	GPFSEL	Pins 50-53	(3-bits per pin)
7		Pins 0-31	
8	GPSET	Pins 32-53	(1-bit per pin)
10		Pins 0-31	
11	GPCLR	Pins 32-53	(1-bit per pin)
13		Pins 0-31	
14	GPLEV	Pins 32-53	(1-bit per pin)
...			

The meaning of the registers is (see p90ff of [BCM2835 ARM peripherals](#)):

- **GPFSEL**: function select registers (3 bits per pin); set it to 0 for input, 1 for output; 6 more alternate functions available
- **GPSET**: **set** the corresponding pin
- **GPCLR**: **clear** the corresponding pin
- **GPLEV**: return the **value** of the corresponding pin

GPIO Register Assignment

Address	Field Name	Description	Size	Read/Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W
0x 7E20 0030	-	Reserved	-	-

The GPIO has 48 32-bit registers (RPi2; 41 for RPi1).

⁰See [BCM Peripherals Manual](#), Chapter 6, Table 6.1

GPIO Register Assignment

GPIO registers (Base address: 0x3F20000)

GPFSEL0	0:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL1	1:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL2	2:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL3	3:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL4	4:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL5	5:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
—	6:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSET0	7:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSET1	8:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
—	9:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFCLR0	10:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFCLR1	11:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
—	12:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13

⁰See BCM Peripherals, Chapter 6, Table 6.1

Locating the GPFSEL register for pin 47 (ACT)

Bit(s)	Field Name	Description	Type	Reset
31-30	---	Reserved	R	0
29-27	FSEL49	<u>FSEL49 - Function Select 49</u> 000 = GPIO Pin 49 is an input 001 = GPIO Pin 49 is an output 100 = GPIO Pin 49 takes alternate function 0 101 = GPIO Pin 49 takes alternate function 1 110 = GPIO Pin 49 takes alternate function 2 111 = GPIO Pin 49 takes alternate function 3 011 = GPIO Pin 49 takes alternate function 4 010 = GPIO Pin 49 takes alternate function 5	R/W	0
26-24	FSEL48	FSEL48 - Function Select 48	R/W	0
23-21	FSEL47	FSEL47 - Function Select 47	R/W	0
20-18	FSEL46	FSEL46 - Function Select 46	R/W	0
17-15	FSEL45	FSEL45 - Function Select 45	R/W	0
14-12	FSEL44	FSEL44 - Function Select 44	R/W	0
11-9	FSEL43	FSEL43 - Function Select 43	R/W	0
8-6	FSEL42	FSEL42 - Function Select 42	R/W	0
5-3	FSEL41	FSEL41 - Function Select 41	R/W	0
2-0	FSEL40	FSEL40 - Function Select 40	R/W	0

Table 6-6 – GPIO Alternate function select register 4

Accessing a GPIO Pin

- Now we want to control the on-chip LED, called ACT, that normally indicates activity.
- The pin number of this device on the RPi2 is: **47**
- We need to calculate registers and bits corresponding to this pin
- The **GPFSEL** register for pin 47 is **4** (per docu, this register covers pins 40-49 (Tab 6-6, p. 94))
- For each register 3 bits are used to select the function of that pin: bits 0–2 for register 40 etc
- Thus, bits 21–23 cover register 47 (7×3)
- The function that we need to select is OUTPUT, which is encoded as the value **1**
- We need to write the value `0x01` into bits 21–23 of register 4

Accessing a GPIO Pin

- Now we want to control the on-chip LED, called ACT, that normally indicates activity.
- The pin number of this device on the RPi2 is: **47**
- We need to calculate registers and bits corresponding to this pin
- The **GPFSEL** register for pin 47 is **4** (per docu, this register covers pins 40-49 (Tab 6-6, p. 94)
- For each register 3 bits are used to select the function of that pin: bits 0–2 for register 40 etc
- Thus, bits 21–23 cover register 47 (7×3)
- The function that we need to select is OUTPUT, which is encoded as the value 1
- We need to write the value $0x01$ into bits 21–23 of register 4

Accessing a GPIO Pin

- Now we want to control the on-chip LED, called ACT, that normally indicates activity.
- The pin number of this device on the RPi2 is: **47**
- We need to calculate registers and bits corresponding to this pin
- The **GPFSEL** register for pin 47 is **4** (per docu, this register covers pins 40-49 (Tab 6-6, p. 94))
- For each register 3 bits are used to select the function of that pin: bits 0–2 for register 40 etc
- Thus, bits 21–23 cover register 47 (7×3)
- The function that we need to select is OUTPUT, which is encoded as the value 1
- We need to write the value $0x01$ into bits 21–23 of register 4

Accessing a GPIO Pin

- Now we want to control the on-chip LED, called ACT, that normally indicates activity.
- The pin number of this device on the RPi2 is: **47**
- We need to calculate registers and bits corresponding to this pin
- The **GPFSEL** register for pin 47 is **4** (per docu, this register covers pins 40-49 (Tab 6-6, p. 94))
- For each register 3 bits are used to select the function of that pin: bits 0–2 for register 40 etc
- Thus, bits 21–23 cover register 47 (7×3)
- The function that we need to select is OUTPUT, which is encoded as the value **1**
- We need to write the value $0x01$ into bits 21–23 of register 4

Accessing GPIO Pin 47

- We want to construct C code to write the value $0x01$ into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value $0x01$ into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value $0x01$ into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value $0x01$ into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value $0x01$ into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`? **Answer:** `gpio+4`
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value $0x01$ into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value $0x01$ into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
 - How do we blank out bits 21–23 from this register?
 - How do we get the value $0x01$ into bits 21–23 of a 32-bit word?
 - How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
Answer: `*(gpio+4)`
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value $0x01$ into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value $0x01$ into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

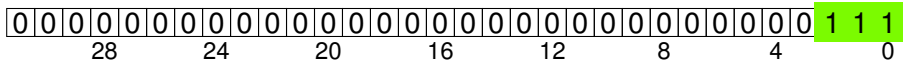
Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
Answer: `*(gpio + 4) & ~(7 << 21)`
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value $0x01$ into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?

C code: 7

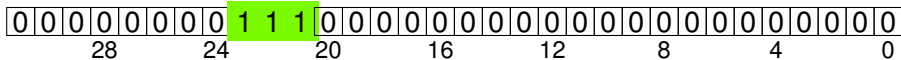


- How do we get the value $0x01$ into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?

C code: `7 << 21`

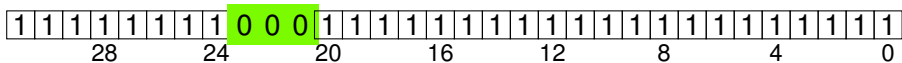


- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value $0x01$ into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?

C code: `~(7 << 21)`

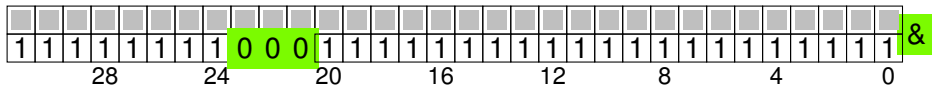


- How do we get the value $0x01$ into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?

C code: `(* (gpio + 4) & ~ (7 << 21))`

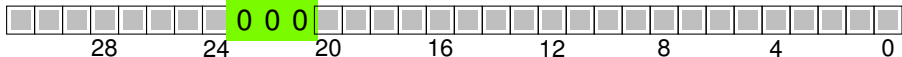


- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value $0x01$ into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?

C code: `(* (gpio + 4) & ~ (7 << 21))`



- How do we get the value $0x01$ into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value $0x01$ into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value $0x01$ into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

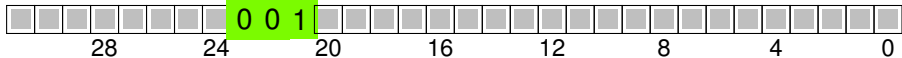
Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
Answer: `(1 << 21)`
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value $0x01$ into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value $0x01$ into bits 21–23 of a 32-bit word?

```
(* (gpio + 4) & ~ (7 << 21)) | (1 << 21)
```



- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value $0x01$ into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value $0x01$ into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

```
*(gpio + 4) = (*(gpio + 4) & ~(7 << 21)) | (1 << 21)
```

C Code: constants and memory mapping

```
// constants for RPi2
gpiobase = 0x3F200000 ;

// memory mapping
// Open the master /dev/memory device, and map it to address
  gpio

if ((fd = open("/dev/mem", O_RDWR | O_SYNC | O_CLOEXEC) ) < 0)
    return failure (FALSE, "Unable_to_open_/dev/mem:_%s\n",
                    strerror(errno)) ;

// gpio is the mmap'ed device memory
gpio = (uint32_t *)mmap(0, BLOCK_SIZE, PROT_READ|PROT_WRITE,
                       MAP_SHARED, fd, gpiobase) ;
if ((int32_t)gpio == -1)
    return failure (FALSE, "_mmap_(GPIO)_failed:_%s\n",
                    strerror(errno)) ;
```

Now, `gpio` is the address of the device memory that we can access directly (if run as root!).

Registers for the GPIO peripherals: GPFSEL

Write into these bits (21–23) to set the **function** for **pin 47**

0:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
1:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
2:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
3:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
4:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
5:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
6:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
7:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
8:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
9:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
10:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
11:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
12:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9



Registers for the GPIO peripherals: GPFSEL

0:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
1:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
2:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
3:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
4:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
5:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
6:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
7:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
8:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
9:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
10:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
11:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
12:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8

C Code: setting the mode of the pin

Essentials

Register no.: 4

Bits: 21–23

Function: 1 (output)

```
// setting the mode for GPIO pin 47
fprintf(stderr, "setting pin %d to %d...\n", pinACT, OUTPUT)
;
fSel = 4; // GPIO 47 lives in register 4 (GPFSEL)
shift = 21; // GPIO 47 sits in slot 7 of register 4, thus
            // shift by 7*3 (3 bits per pin)
*(gpio + fSel) = (*(gpio + fSel) & ~(7 << shift)) | (1 <<
                shift); // Sets bits to one = output
// *(gpio + fSel) = (*(gpio + fSel) & ~(7 << shift));
// Sets bits to zero = input
```

Now, pin 47 (the on-board ACT LED) is set as an output device.

C Code: setting the mode of the pin

Essentials

Register no.: 4

Bits: 21–23

Function: 1 (output)

```
// setting the mode for GPIO pin 47
fprintf(stderr, "setting pin_%d to_%d...\n", pinACT, OUTPUT)
;
fSel = 4; // GPIO 47 lives in register 4 (GPFSEL)
shift = 21; // GPIO 47 sits in slot 7 of register 4, thus
            // shift by 7*3 (3 bits per pin)
*(gpio + fSel) = (*(gpio + fSel) & ~(7 << shift)) | (1 <<
                shift); // Sets bits to one = output
// *(gpio + fSel) = (*(gpio + fSel) & ~(7 << shift));
// Sets bits to zero = input
```

Now, pin 47 (the on-board ACT LED) is set as an output device.

GPIO Registers for Turning the LED on/off

Address	Field Name	Description	Size	Read/Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W
0x 7E20 0030	-	Reserved	-	-

We now need to access the GPSET and GPCLR register for pin 47.

⁰See BCM Peripherals Manual, Chapter 6, Table 6.1

Turning the LED on or off

Write into this bit (15) to **set** pin 47

0:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
1:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
2:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
3:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
4:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
5:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
6:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
7:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
GPSET1	30	29	28	27	26	25	24	23	22	21	20	19	18	17	pin 47	13	12	11	10	9			
9:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
10:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
GPCLR1	30	29	28	27	26	25	24	23	22	21	20	19	18	17	pin 47	13	12	11	10	9			
12:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9

Write into this bit (15) to **clear** pin 47

Turning the LED on or off

Write into this bit (15) to **set** pin 47

0:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
1:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
2:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
3:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
4:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
5:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
6:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
7:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
8:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
9:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
10:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
11:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
12:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8

Write into this bit (15) to **clear** pin 47

Code: blinking LED

```
for (j=0; j<1000; j++) {
    theValue = ((j % 2) == 0) ? HIGH : LOW;
    // write the value into the location corresp. to pin 47
    if ((pinACT & 0xFFFFF0) == 0) // sanity check
    {
        if (theValue == LOW) { // GPCLR
            // GPCLR for GPIOs 32-53 is register 11
            clrOff = 11; // register for clearing a pin value
            *(gpio + clrOff) = 1 << (pinACT & 31) ;
        } else { // GPSET
            // GPSET for GPIOs 32-53 is register 8
            setOff = 8; // register for setting a pin value
            *(gpio + setOff) = 1 << (pinACT & 31) ;
        }
    } else { fprintf(stderr, "only_supporting_on-board_pins\n
        "); exit(1); }

    // delay for howLong ms, using a Linux system function
    ...
}
```

Discussion

- In each iteration of the loop, we toggle `theValue` between the constants `HIGH` and `LOW`
- This is **not** the value written to a register, but a flag for the control flow
- If `theValue` is `LOW`, we write a `1` into the corresponding **GPCLR** register, to turn the LED **off**
- If `theValue` is `HIGH`, we write a `1` into the corresponding **GPSET** register, to turn the LED **off**
- Note, that we determine the bit location in these registers by `pinACT & 31`, which is the same as taking `pinACT` modulo 32
- We then wait for a certain amount of time to control the blinking frequency

See sample source: `tut_led.c`

The main registers that you need to know about

Address	Field Name	Description	Size	Read/Write
FctSelect 0 1 2 3 4 5	GPFSEL0	GPIO Function Select 0	32	R/W
	GPFSEL0	GPIO Function Select 0	32	R/W
	GPFSEL1	GPIO Function Select 1	32	R/W
	GPFSEL2	GPIO Function Select 2	32	R/W
	GPFSEL3	GPIO Function Select 3	32	R/W
	GPFSEL4	GPIO Function Select 4	32	R/W
Set Registers 7 8	GPFSEL5	GPIO Function Select 5	32	R/W
	-	Reserved	-	-
	GPSET0	GPIO Pin Output Set 0	32	W
Clear Registers 10 11	GPSET1	GPIO Pin Output Set 1	32	W
	-	Reserved	-	-
	GPCLR0	GPIO Pin Output Clear 0	32	W
	GPCLR1	GPIO Pin Output Clear 1	32	W
-	Reserved	-	-	

The main registers that you need to know about

Address	Field Name	Description	Size	Read/Write
FctSelect 0 1 2 3 4 5	GPFSEL0	GPIO Function Select 0	32	R/W
	GPFSEL0	GPIO Function Select 0	32	R/W
	GPFSEL1	GPIO Function Select 1	32	R/W
	GPFSEL2	GPIO Function Select 2	32	R/W
	GPFSEL3	GPIO Function Select 3	32	R/W
	GPFSEL4	GPIO Function Select 4	32	R/W
Set Registers 7 8	-	Reserved	-	-
	GPSET0	GPIO Pin Output Set 0	32	W
	GPSET1	GPIO Pin Output Set 1	32	W
Clear Registers 10 11	-	Reserved	-	-
	GPCLR0	GPIO Pin Output Clear 0	32	W
	GPCLR1	GPIO Pin Output Clear 1	32	W
-	Reserved	-	-	

Controlling the LED in Assembler

```
@ ... mmap boilerplate here
ADD    R3, R3, #4           @ add 4 for block 1
LDR    R2, [SP, #16]       @ get virtual mem addr
ADD    R2, R2, #16         @ add 16 for block 4
LDR    R2, [R2, #0]        @ load R2 with value at R2
BIC    R2, R2, #0b111<<21 @ Bitwise clear of three bits
STR    R2, [R3, #0]        @ Store result in Register
LDR    R3, [SP, #16]       @ Get virtual mem address
ADD    R3, R3, #16         @ Add 16 for block 4
LDR    R2, [SP, #16]       @ Get virtual mem addr
ADD    R2, R2, #4          @ add 16 for block 4
LDR    R2, [R2, #0]        @ Load R2 with value at R2
ORR    R2, R2, #1<<21     @ Set bit....
STR    R2, [R3, #0]        @ ...and make output
LDR    R3, [SP, #16]       @ get virt mem addr
ADD    R3, R3, #32         @ add 32 to offset for GPSET1
MOV    R4, #1              @ get 1
MOV    R2, R4, LSL#15      @ Shift by pin number
STR    R2, [R3, #0]        @ write to memory
```

See sample source: [gpio47on.s](#)

⁰From: Bruce Smith "Raspberry Pi Assembly Language: Raspbian", Ch 25

Summary

- Controlling a simple external device means **logically** sending 1 bit of information (on/off)
- Realising this control means **physically** writing into special registers which have special meaning
- The information on the special meaning is usually in bulky hardware-description documentation
- Once uncovered, the code for direct device control is fairly short
- The **sample sources** show a C and an Assembler version of turning pin 47 (ACT) on/off

Thanks to **Gordon Henderson** for his sterling work on the **wiringPi library!**

Tutorial 3: Programming a Button input device

- In this tutorial we want to use a button, connected through a breadboard as an input device.
- This is the simplest input device that we will cover.
- The code needed to control is typical for such devices.
- This tutorial deals with **programming a button as input device.**

Core Techniques

- In the LED tutorial, we have seen that we first need to **identify the registers** that give control to the device.
- For that we will again look into the **BCM Peripherals** documentation.
- We will then go through a simple example of
 - ▶ reading button input data,
 - ▶ blinking an LED on button press.
- We want to connect the button with **pin 24**, using a breadboard.
- These simple activities, will also be at the core of CW2.

GPIO Register Assignment

Registers

13

14

	GPLEV0	GPIO Pin Level 0	32	R
	GPLEV1	GPIO Pin Level 1	32	R
0x 7E20 003C	-	Reserved	-	-
0x 7E20 0040	GPEDS0	GPIO Pin Event Detect Status 0	32	R/W
0x 7E20 0044	GPEDS1	GPIO Pin Event Detect Status 1	32	R/W
0x 7E20 0048	-	Reserved	-	-
0x 7E20 004C	GPREN0	GPIO Pin Rising Edge Detect Enable 0	32	R/W
0x 7E20 0050	GPREN1	GPIO Pin Rising Edge Detect Enable 1	32	R/W
0x 7E20 0054	-	Reserved	-	-
0x 7E20 0058	GPFEN0	GPIO Pin Falling Edge Detect Enable 0	32	R/W
0x 7E20 005C	GPFEN1	GPIO Pin Falling Edge Detect Enable 1	32	R/W

⁰See [BCM Peripherals Manual](#), Chapter 6, Table 6.1

GPIO Register Assignment

Registers

13

14

	GPLEV0	GPIO Pin Level 0	32	R
	GPLEV1	GPIO Pin Level 1	32	R
0x 7E20 003C	-	Reserved	-	-
0x 7E20 0040	GPEDS0	GPIO Pin Event Detect Status 0	32	R/W
0x 7E20 0044	GPEDS1	GPIO Pin Event Detect Status 1	32	R/W
0x 7E20 0048	-	Reserved	-	-
0x 7E20 004C	GPREN0	GPIO Pin Rising Edge Detect Enable 0	32	R/W
0x 7E20 0050	GPREN1	GPIO Pin Rising Edge Detect Enable 1	32	R/W
0x 7E20 0054	-	Reserved	-	-
0x 7E20 0058	GPFEN0	GPIO Pin Falling Edge Detect Enable 0	32	R/W
0x 7E20 005C	GPFEN1	GPIO Pin Falling Edge Detect Enable 1	32	R/W

⁰See [BCM Peripherals Manual](#), Chapter 6, Table 6.1

BCM2835 GPIO Peripherals

Base address: 0x3F000000

0		Pins 0-9	(3-bits per pin)
5	GPFSEL	Pins 50-53	
7		Pins 0-31	(1-bit per pin)
8	GPSET	Pins 32-53	
10		Pins 0-31	(1-bit per pin)
11	GPCLR	Pins 32-53	
13		Pins 0-31	(1-bit per pin)
14	GPLEV	Pins 32-53	
...			

The main registers that we need in this case are (see p90ff of [BCM2835 ARM peripherals](#)):

- **GPFSEL**: function select registers (3 bits per pin); set it to 0 for input, 1 for output; 6 more alternate functions available
- **GPSET**: set the corresponding pin
- **GPCLR**: clear the corresponding pin
- **GPLEV**: return the **value** of the corresponding pin

Define the button pin as an INPUT device

Write into these bits (12–14) to set the **function** for **pin 24**

0:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
1:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
2:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	pin 24	13	12	11	10	9	8
3:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
4:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
5:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
6:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
7:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
8:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
9:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
10:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
11:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
12:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8

Define the button pin as an INPUT device

Write into these bits (12–14) to set the **function** for pin 24

0:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
1:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
2:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
3:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
4:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
5:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
6:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
7:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
8:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
9:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
10:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
11:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
12:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8

NB: Recall, each GPFSEL register controls 10 pins (1–10, 11–20, etc),
for each pin, 3-bits control the behaviour

Reading from the button input

Contents:

Bit positions

GPIO registers (Base address: 0x3F20000)

4:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
5:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
6:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
7:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
8:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
9:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
10:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
11:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
12:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
GPLEV0	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	
14:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9
15:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9

Read this bit (24)

Reading from the button input

Contents:

Bit positions

GPIO registers (Base address: 0x3F200000)

4:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
5:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
6:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
7:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
8:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
9:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
10:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
11:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
12:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
13:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
14:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10
15:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10

Read from this bit (24)

Reading from the button input

Contents:

Bit values

GPIO registers (Base address: 0x3F200000)

4																				
5																				
6																				
7																				
8																				
9																				
10																				
11																				
12																				
13							1													
14																				
15																				

Input: HIGH

Reading from the button input

Contents:
Bit values

GPIO registers (Base address: 0x3F200000)

4																																				
5																																				
6																																				
7																																				
8																																				
9																																				
10																																				
11																																				
12																																				
13								0																												
14																																				
15																																				

Input: LOW

Sample C code: Button input

First we define some constants that we will need.

```
// Tunables:
// PINs (based on BCM numbering)
#define LED 23
#define BUTTON 24
// delay for loop iterations (mainly), in ms
#define DELAY 200

#define INPUT 0
#define OUTPUT 1

#define LOW 0
#define HIGH 1
```

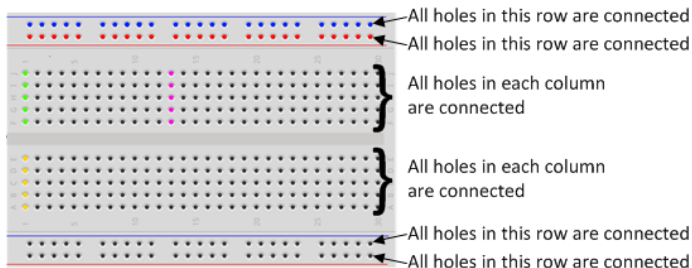
This assumes that we have wired-up the button with GPIO pin 24 and the LED with GPIO pin 23.

Using a breadboard

To control an external LED, you could directly connect GPIO pins with the LED and a resistor using jumper cables.

However, a breadboard is a more flexible way of wiring peripherals, such as LEDs or buttons, to the RPi.

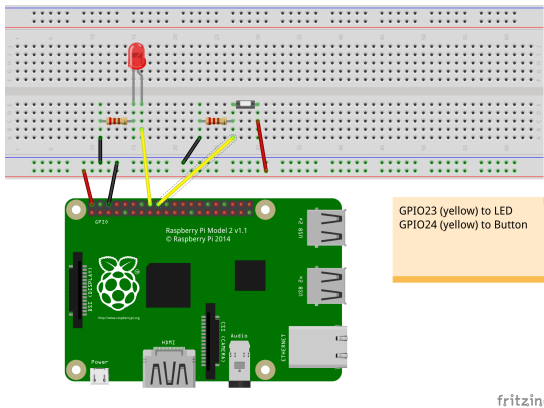
You need to understand how the columns and the rows on a breadboard are connected, though.



For a good basic intro on how to use a breadboard [follow this link](#)

The wiring as a Fritzing diagram

To describe a specific wiring, we use **Fritzing diagrams** like this:



An **LED**, as output device, is connected to the RPi2 using **GPIO pin 23**.

A **button**, as input device, is connected to **GPIO pin 24**.

Sample C code: Button input

We memory-map the addresses for the GPIO registers (as before).

```
gpiobase = 0x3F200000;
// memory mapping
if ((fd = open ("/dev/mem", O_RDWR | O_SYNC |
    O_CLOEXEC) ) < 0)
    return failure (FALSE, "setup:_Unable_to_open_/_
        dev/mem:_%s\n", strerror (errno)) ;
// GPIO:
gpio = (uint32_t *)mmap(0, BLOCK_SIZE, PROT_READ |
    PROT_WRITE, MAP_SHARED, fd, gpiobase) ;
if ((int32_t)gpio == -1)
    return failure (FALSE, "setup:_mmap_(GPIO)_
        failed:_%s\n", strerror (errno)) ;
```

Sample C code: Button input

We set the modes for the LED pin (OUTPUT) and the button pin (INPUT).

```
// setting the mode
fSel  = 2; // register 2 (GPFSEL2)
shift = 9; // slot 3 (shift 3*3)
// set the above pin to output mode
*(gpio + fSel) = (*(gpio + fSel) & ~(7 << shift))
    | (1 << shift) ; // Sets bits to one = output

fSel  = 2; // register 2 (GPFSEL2)
shift = 12; // slot 4 (shift 4*3)
// set the above pin to input mode
*(gpio + fSel) = (*(gpio + fSel) & ~(7 << shift))
    ; // Sets bits to zero = input
```

Sample C code: Button input

Inside the main loop, we first read from the bit associated with the button input in the `GPLEV0` register.

```
for (j=0; j<1000; j++) {
    if ((* (gpio + 13 /* GPLEV0 */) & (1 << (BUTTON &
        31))) != 0)
        theValue = HIGH ;
    else
        theValue = LOW ;
}
```

Sample C code: Button input

Further down the loop, we write to the bit associated with the LED output in the `GPCLR0` or `GPSET0` register.

```
if (theValue == LOW) {
    clrOff = 10; // GPCLR0 for pin 23
    *(gpio + clrOff) = 1 << (LED & 31); // 23-rd bit
        in the register
} else {
    setOff = 7; // GPSET0 for pin 23
    *(gpio + setOff) = 1 << (LED & 31); // 23-rd bit
        in the register
}
// delay ...
```

Sample C code: Button input

Finally, we want to clean-up by setting the LED to LOW. Which kind of code do we need here?

```
// clean-up by setting the LED pin to LOW
```

Summary

- Reading input from a button works in the same way as writing to the LED:
 - ▶ We need to identify the relevant registers and bits for our pin
 - ▶ We declare the pin an INPUT device in the `GPFSEL` register
 - ▶ We read from the associated bit in the `GPLEV` register to get the input
- With the button you have a basic input device to communicate with the system
- In the CW we will combine a button (for input), an LED (for output) and an LCD display (for nicer output) and implement a small app for this configuration.

See sample source: `tut_button.c`

Tutorial 4: Inline Assembler with gcc

- So far we have developed either C or Assembler programs separately.
- Linking the compiled code of both C and Assembler sources together we can call one from the other.
- This is ok, but sometimes inconvenient because
 - ▶ errors occur only at link time, and carry little information
 - ▶ we can't easily parameterise the Assembler code (e.g. with the `gpio` base address)
- In this tutorial we will cover **how to embed assembler code into a C program, using the `gcc` and the GNU toolchain**

A Simple Example

Essentials

val provides the input
asm code returns its value
val3 receives the output

Look-up the value in `val` and copy it to `val3`:

```
static volatile int val = 1024, val3,  
asm( /* multi-line example of value look-up and return  
*/  
    "\tMOV_R0,_%[value]\n"          /* load the address  
        into R0 */  
    "\tLDR_%[result],_%[R0,_%#0]\n" /* get and return  
        the value at that address */  
    : [result] "=r" (val3)          /* output parameter */  
    : [value] "r" (&val)           /* input parameter */  
    : "r0", "cc" );                /* registers used */  
  
fprintf(stderr, "Value_lookup_at_address_%x_(expect_%d  
:_%d\n", &val, val, val3);
```

⁰Sample source in [sample0.c](#); see also [ARM inline assembly blog](#)

Example explained

- The `asm` command defines a block of assembler code that is put at that location into the C code (embedded).
- The assembler code itself is written as a sequence of strings, each starting with a TAB (`\t`) and ending with a newline (`\n`) to match usual assembler code formatting.
- Inside the strings, the code can refer to arguments provided in the “output parameter” and “input parameter” sections.
- These sections define a **name** (e.g. `result`) that can be used in the assembler code (e.g. `%[result]`), and which is bound to a concrete variable or value (e.g. `val3`).
- Think of these in the same way as formatting strings in `printf` statements.

Example explained (cont'd)

- For example the line
`: [result] "=r" (val3)`
says “the name `result`, which is referred to in the assembler code as `%[result]`, is bound to the C variable `val3`; moreover, it should be represented as a register (`"r"`)”
- So, what this example code does is to load the address of the C variable `val` into the register `R0`, and then to load the value at this address, i.e. the contents of the C variable `val`, into the C variable `val3`, which should be kept in a register (`"r"`)
- The last section of the `asm` block defines which registers are modified by this assembler block. This information is needed by the compiler when doing register allocation.

GCC Extended Assembler Commands

Using gcc you can embed assembler code into your C programs, i.e. write “inline assembler” code in C.

The format for the inline assembler code is

```
asm [volatile] ( AssemblerTemplate
                  : OutputOperands
                  [ : InputOperands
                  [ : Clobbers ] ] )
```

⁰See [GCC Manual, Section “Extended Asm”](#)

GCC Extended Assembler Commands

Using gcc you can embed assembler code into your C programs, i.e. write “inline assembler” code in C.

The format for the inline assembler code is

```
asm [volatile] ( AssemblerTemplate
                  : OutputOperands
                  [ : InputOperands
                  [ : Clobbers ] ] )
```

`AssemblerTemplate`: This is a literal string that is the template for the assembler code. It is a combination of fixed text and tokens that refer to the input, output, and goto parameters.

⁰See [GCC Manual, Section “Extended Asm”](#)

GCC Extended Assembler Commands

Using gcc you can embed assembler code into your C programs, i.e. write “inline assembler” code in C.

The format for the inline assembler code is

```
asm [volatile] ( AssemblerTemplate
                 : OutputOperands
                 [ : InputOperands
                 [ : Clobbers ] ])
```

`OutputOperands`: A comma-separated list of the C variables modified by the instructions in the `AssemblerTemplate`. An empty list is permitted.

⁰See [GCC Manual, Section “Extended Asm”](#)

GCC Extended Assembler Commands

Using gcc you can embed assembler code into your C programs, i.e. write “inline assembler” code in C.

The format for the inline assembler code is

```
asm [volatile] ( AssemblerTemplate
                  : OutputOperands
                  [ : InputOperands
                  [ : Clobbers ] ] )
```

InputOperands: A comma-separated list of C expressions read by the instructions in the `AssemblerTemplate`. An empty list is permitted.

⁰See [GCC Manual, Section “Extended Asm”](#)

GCC Extended Assembler Commands

Using gcc you can embed assembler code into your C programs, i.e. write “inline assembler” code in C.

The format for the inline assembler code is

```
asm [volatile] ( AssemblerTemplate
                  : OutputOperands
                  [ : InputOperands
                  [ : Clobbers ] ])
```

Clobbers: A comma-separated list of registers or other values changed by the AssemblerTemplate, beyond those listed as outputs. An empty list is permitted.

⁰See [GCC Manual, Section “Extended Asm”](#)

Another Example

Using a pair data structure, the function below computes the sum of both fields.

```
typedef struct {
    ulong min;    ulong max;
} pair_t;

ulong sumpair_asm(pair_t *pair) {
    ulong res;
    asm volatile( /* sum over int values */
        "\tLDR_R0,_[%[inp],_#0]\n"
        "\tLDR_R1,_[%[inp],_#4]\n"
        "\tADD_R0,_R0,_R1\n"
        "\tMOV_%[result],_R0\n"
        : [result] "=r" (res)
        : [inp] "r" (pair)
        : "r0", "r1", "cc" );

    return res;
}
```

Essentials

C variable `pair` is passed as `inp`
"r": keep in register
"=r": the register is written to

Modifiers and constraints to the input/output operands

When mapping **names** to C **variables** or **expressions**, the following constraints and modifiers can be specified:

Constraint	Specification
<code>f</code>	Floating point registers <code>f0 ... f7</code>
<code>r</code>	General register <code>r0 ... r15</code>
<code>m</code>	Memory address
<code>I</code>	Immediate value

Modifier	Specification
<code>=</code>	Write-only operand, usually used for all output operands
<code>+</code>	Read-write operand, must be listed as an output operand
<code>&</code>	A register that should be used for output only

E.g. : `[result] "=r" (res)`

means that the name `result` should be a register in the assembler code, and that it will be written to, by the assembler code.

Modifiers and constraints to the input/output operands

When mapping **names** to C **variables** or **expressions**, the following constraints and modifiers can be specified:

Constraint	Specification
<code>f</code>	Floating point registers <code>f0 ... f7</code>
<code>r</code>	General register <code>r0 ... r15</code>
<code>m</code>	Memory address
<code>I</code>	Immediate value

Modifier	Specification
<code>=</code>	Write-only operand, usually used for all output operands
<code>+</code>	Read-write operand, must be listed as an output operand
<code>&</code>	A register that should be used for output only

E.g. : `[result] "=r" (res)`

means that the name `result` should be a register in the assembler code, and that it will be written to, by the assembler code.

Extended inline assembler: Example

Using a pair data structure, the function below puts the smaller value into the `min` and the larger value into the `max` field:

```
typedef struct {
    ulong min;  ulong max;
} pair_t;

void minmax_c(pair_t *pair) {
    ulong t;
    if (pair->min > pair->max) {
        t = pair->min;
        pair->min = pair->max;
        pair->max = t;
    }
}
```

⁰Sample source: [sumav1_asm.c](#)

Extended inline assembler: Example

```
void minmax_asm(pair_t *pair) {
    pair_t *res;
    asm volatile("\tLDR_R0,_[%[inp],_#0]\n"
                "\tLDR_R1,_[%[inp],_#4]\n"
                "\tCMP_R0,_R1\n"
                "\tBLE_done\n"
                "\tMOV_R3,_R0\n"
                "\tMOV_R0,_R1\n"
                "\tMOV_R1,_R3\n"
                "done:_STR_R0,_[%[inp],_#0]\n"
                "\tSTR_R1,_[%[inp],_#4]\n"
                : [result] "=r" (res)
                : [inp] "r" (pair)
                : "r0", "r1", "r3", "cc" );
}
```

Discussion

- `inp` needs to be in a register, because it contains the base address in a load operation (`LDR`)
- we don't use `res` in this case, but it usually needs the `"=r"` modifier and constraint
- the clobber list must name **all registers that are modified** in the code: `r0, r1, r3`
- we could pass in an immediate value `sizeof(ulong)` and use it instead of the literal `#4` to make the code less hardware-dependent

Summary

- With gcc's in-line assembler commands (`asm`) you can embed assembler code into C code.
- This avoids having to write code in separate files and then link them together.
- The assembler code can be parameterised over C variables and expressions, to simplify passing arguments.
- Care needs to be taken to define **constraints** and **modifiers** (keep data in registers or memory)
- Registers that are modified need to be explicitly identified in the “clobber list”.
- It is recommended to use such in-line assembler code for CW2, where you need to develop an application in C and assembler.

Sample sources: [sample0.c](#), and [sumav1_asm.c](#)

Tutorial 5: Programming an LCD Display

This tutorial will focus on programming a simple output device:
an 16x2 LCD display using an Hitachi HD44780U controller

This will be an exercise of controlling a device slightly more complicated than the LED and button devices so far.
The principles of programming are the same as before.

Overview

We will cover:

- 1 Connecting an LCD display to the RPi2
- 2 Low-level interface in assembler (`digitalWrite`)
- 3 Medium-level interface in C (`lcdClear` etc)
- 4 Sending characters and strings (`lcdPutChar`, `lcdPuts`)
- 5 Character data (defining own characters)

Acknowledgements

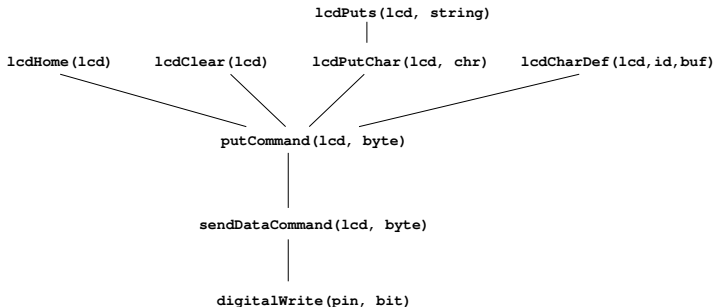
The code in this tutorial is mostly taken directly from the **wiringPi** library for the Raspberry Pi, by **Gordon Henderson**.

If you have downloaded the sources, you can look-up examples in the directory `wiringPi/examples` (e.g. `lcd.c`) and the code for the LCD functions in `wiringPi/devLib` (also `lcd.c`)

```
* wiringPi:
* Arduino look-a-like Wiring library for the Raspberry Pi
* Copyright (c) 2012-2015 Gordon Henderson
* Additional code for pwmSetClock by Chris Hall <chris@kchall.plus.com>
*
* Thanks to code samples from Gert Jan van Loo and the
* BCM2835 ARM Peripherals manual, however it's missing
* the clock section /grr/mutter/
*****
* This file is part of wiringPi:
* https://projects.drogon.net/raspberry-pi/wiringpi/
```

Function dependencies

Here is a simple picture of the dependencies of the API functions:



NB: only the lowest level, `digitalWrite` is in assembler, the rest is in C

LCD commands

We need some constant definitions and boilerplate code:
Here is a list of instructions for the Hitachi HD44780U controller:

```
#define LCD_CLEAR      0x01
#define LCD_HOME      0x02
#define LCD_ENTRY     0x04
#define LCD_CTRL      0x08
#define LCD_CDSHIFT   0x10
#define LCD_FUNC      0x20
#define LCD_CGRAM     0x40
#define LCD_DGRAM     0x80
```

⁰See Table 6 and Figure 11 in the [HD4478 Technical Reference](#) 

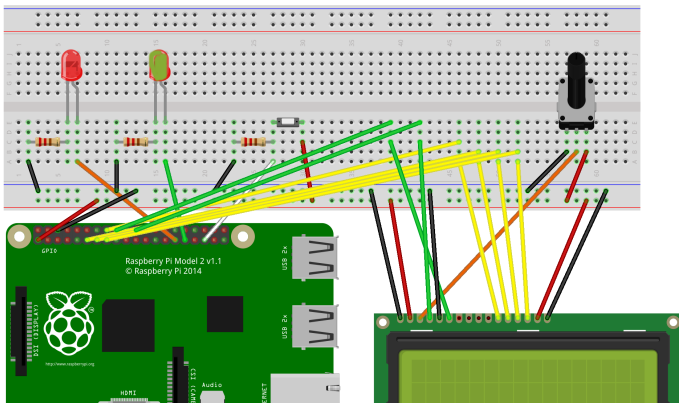
1. The wiring

Pin 19 (White) Button
Pin 6 (Green) green LED
Pin 5 (Orange) red LED

All power wires are red
All ground wires are black

LCD Data (Yellow)
GPIO 23 <-> LCD 11
GPIO 17 <-> LCD 12
GPIO 27 <-> LCD 13
GPIO 22 <-> LCD 14

LCD Control (Green)
GPIO 25 <-> LCD 4
GPIO 24 <-> LCD 6



The wiring: encoded

To encode this wiring in the program we define:

```
#define STRB_PIN 24
#define RS_PIN 25
#define DATA0_PIN 23
#define DATA1_PIN 17
#define DATA2_PIN 27
#define DATA3_PIN 22
```

Data structure for the LCD-connection

The following data structure stores the pin numbers and cursor position:

```
struct lcdDataStruct
{
    int bits, rows, cols ;
    int rsPin, strbPin ;
    int dataPins [8] ;
    int cx, cy ;
} ;
```

2. Low-level Assembler interface

This code is essentially the same as in the blinking LED example, i.e. we want to “send” one bit to a pin that’s an argument to the interface:

- Set the mode of the pin to output (before calling the function)
- Identify the register and bit to write to
- Write one bit (**1**) into this location
- It is recommended that you use inline assembler to implement this function

2. Low-level Assembler interface

```
asm volatile( /* inline assembler version of setting/clearing LED to
              output */
    "\tB___bonzo0\n"
    "_bonzo0:_NOP\n"
    "\tLDR_R1,_%[gpio]\n"
    "\tADD_R0,_R1,_%[off]\n" /* R0 = GPSET/GPCLR register */
    "\tMOV_R2,_#1\n"
    "\tMOV_R1,_%[pin]\n" /* NB: this works only for pin 0-31 */
    "\tAND_R1,_#31\n"
    "\tLSL_R2,_R1\n" /* R2 = bitmask set/clear reg %[act] */
    "\tSTR_R2,_[R0,_#0]\n" /* write bitmask */
    "\tMOV_%[result],_R2\n"
    : [result] "=r" (res)
    : [pin] "r" (pin)
    , [gpio] "m" (gpio)
    , [off] "r" (off*4)
    : "r0", "r1", "r2", "cc");
```


3. Medium-level interface

Sending data uses `digitalWrite` to send bits over the 4 pins:

```
void sendDataCmd (const struct lcdDataStruct *lcd, unsigned char data)
{
    unsigned char          i, d4 ;

    d4 = (myData >> 4) & 0x0F;
    for (i = 0 ; i < 4 ; ++i)
    {
        digitalWrite (lcd->dataPins [i], (d4 & 1)) ;
        d4 >>= 1 ;
    }
    strobe (lcd) ;

    d4 = myData & 0x0F ;
    for (i = 0 ; i < 4 ; ++i)
    {
        digitalWrite (lcd->dataPins [i], (d4 & 1)) ;
        d4 >>= 1 ;
    }
    strobe (lcd) ;
}
```

Sending a command

Sending a command works like sending a byte, except that we only need 4 bits to encode the command, and therefore only one loop in the body:

```
void lcdPut4Command (const struct lcdDataStruct *lcd, unsigned char
    command) {
    register unsigned char myCommand = command ;
    register unsigned char i ;

    digitalWrite (lcd->rsPin, 0) ;

    for (i = 0 ; i < 4 ; ++i) {
        digitalWrite (lcd->dataPins [i], (myCommand & 1)) ;
        myCommand >>= 1 ;
    }
    strobe (lcd) ;
}
```

Move cursor home

Now that we can send a command, we can create instances for each of the commands that are specified for the HD44780U controller:

```
void lcdHome (struct lcdDataStruct *lcd) {
    lcdPutCommand (lcd, LCD_HOME) ;
    lcd->cx = lcd->cy = 0 ;
    delay (5) ;
}
```

4. Sending characters and strings

Sending a character involves, sending the char as a byte, moving to the next position, and updating the position on the LCD display:

```
void lcdPuchar (struct lcdDataStruct *lcd, unsigned
    char data) {
    digitalWrite (lcd->rsPin, 1) ;
    sendDataCmd  (lcd, data) ;

    if (++lcd->cx == lcd->cols) {
        lcd->cx = 0 ;
        if (++lcd->cy == lcd->rows)
            lcd->cy = 0 ;

        lcdPutCommand (lcd, lcd->cx + (LCD_DGRAM | (lcd
            ->cy>0 ? 0x40 : 0x00) /* rowOff [lcd->cy]
            */ )) ;
    }
```

Writing strings

Once we can send characters, we only need a loop on top of it to send entire strings:

```
void lcdPuts (struct lcdDataStruct *lcd, const char
             *string) {
    while (*string)
        lcdPuchar (lcd, *string++) ;
}
```

5. Putting things together

In the main function we:

- Memory-map the GPIO address into user space (`gpio`)
- Configure an `lcd` data structure with the pin numbers for our wiring
- Initialise the connection to this `lcd`
- Initialise the display using `lcdClear()` and `lcdHome()`
- Write “Hello World” using `lcdPuts`

See the `lcd-hello.c` sample program.

5. Putting things together

In the main function we:

- Memory-map the GPIO address into user space (`gpio`)
- Configure an `lcd` data structure with the pin numbers for our wiring
- Initialise the connection to this `lcd`
- Initialise the display using `lcdClear()` and `lcdHome()`
- Write “Hello World” using `lcdPuts`

See the `lcd-hello.c` sample program.

Tutorial 6: Performance Counters on the RPi 2

- Performance counters are hardware support for monitoring basic operations on the CPU
- They are very accurate and useful for monitoring resource consumption
- It is possible to count cycles, but also cache misses, (mispredicted) branches etc
- In this tutorial we will cover **how to use performance counters to get a precise measure of the runtime of a program**

Architecture Support

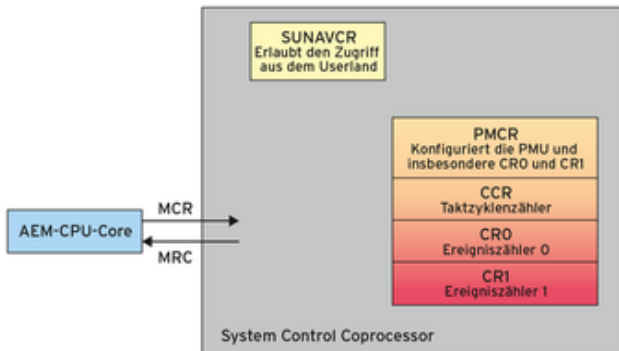
- Both the BCM2835 (of the RPi 1) and BCM2836 (of the RPi 2) provide a **Performance Monitoring Unit (PMU)** as a co-processor on the chip
- The unit supports in total 4 counter registers and a separate cycle counter register.
- These 4 registers can be configured to count a range of low-level events.
- There are 2 different interfaces for accessing this information.
 - ▶ the APB interface, which uses memory mapping and access registers on the PMU directly
 - ▶ the **CP15 interface**, which uses special assembler instructions for communicating between processor and PMU
- The PMU operations are usually not available for user programs (trying to run them directly will trigger a SIGILL exception)
- However, we can write a simple Linux kernel module to enable this functionality, and then use it through assembler instructions in our user code.

Overview: How to use the PMU

We need to go through the following steps:

- 1 Find out how to interact with the PMU
- 2 Enable access to the PMU from “user space”
- 3 Define what we want to monitor
- 4 Use access to the PMU to measure programs

Step 1: Find out how to interact with the PMU



The PMU is a **co-processor**, called **CP15**, separate from the main processor, but on the same chip.

The special assembler instructions **MRC** and **MCR** transfer data between processor register (R) and co-processor (C).

⁰From [Linux Magazin 05/2015: Kerntechnik](#)

Instructions for data transfer between processor and co-processor

- The ARM instruction set provides 2 instructions for the
 - ▶ **MCR**: Move to Coproc from ARM Reg
 - ▶ **MRC**: Move to ARM Reg from Coproc

The technical reference manual describes the instructions like this:

To access the PMCR, read or write the CP15 registers with:

MRC p15, 0, <Rt>, c9, c12, 0; Read Performance Monitor Control Register

MCR p15, 0, <Rt>, c9, c12, 0; Write Performance Monitor Control Register

⁰See [Cortex A7 MPcore Technical Reference Manual, Table 11-1 PMU register summary, p 241](#)

Step 2: Enabling PMU access through a kernel module

- By default, the PMU can only be accessed in “privileged mode”, but this can be changed
- We need to construct a small Linux kernel module that enables the access to the PMU
- In essence, we need to embed some assembler instructions into an API pre-scribed by the Linux kernel
- For details on how to build a Linux kernel module see
 - ▶ [The Linux Kernel Module Programming Guide](#), Peter Jay Salzman
 - ▶ [Building instructions from a course on “Introduction to Embedded Computing”](#) at Univ of California, San Diego, by Tajana Simunic Rosing
- Here, I’ll just shortly summarise the steps needed, and how to use performance monitoring in a simple example program

Table 11-1: PMU registers

Table 11-1 PMU register summary

Register number	Offset	CRn	Op1	CRm	Op2	Name	Type	Description
0	0x000	c9	0	c13	2	PMXEVCNTR0	RW	Event Count Register, see the <i>ARM Architecture Reference Manual</i>
1	0x004	c9	0	c13	2	PMXEVCNTR1	RW	
2	0x008	c9	0	c13	2	PMXEVCNTR2	RW	
3	0x00C	c9	0	c13	2	PMXEVCNTR3	RW	
4-30	0x010-0x78	-	-	-	-	-	-	Reserved
31	0x07C	c9	0	c13	0	PMCCNTR	RW	Cycle Count Register, see the <i>ARM Architecture Reference Manual</i>
32-255	0x080-0x3FC	-	-	-	-	-	-	Reserved
256	0x400	c9	0	c13	1	PMXEVTYPER0	RW	Event Type Selection Register, see the <i>ARM Architecture Reference Manual</i>
257	0x404	c9	0	c13	1	PMXEVTYPER1	RW	
258	0x408	c9	0	c13	1	PMXEVTYPER2	RW	
259	0x40C	c9	0	c13	1	PMXEVTYPER3	RW	

⁰See [Cortex A7 MPcore Technical Reference Manual, Table 11-1 PMU register summary, p 237](#)

Table 11-1: PMU registers

897	0xE04	c9	0	c12	0	PMCR	RW	<i>Performance Monitor Control Register on page 11-7</i>
898	0xE08	c9	0	c14	0	PMUSERENR	RW	User Enable Register, see the <i>ARM Architecture Reference Manual</i>
899-903	0xE0C-0xE1C	-	-	-	-	-	-	Reserved

- The two main registers that we need to access are `PMCR` and `PMUSERENR`
 - ▶ `PMCR`: controls access to the PMU in general
 - ▶ `PMUSERENR`: is the **User Enable Register** that needs to be configured to allow user code to access the PMU

⁰See [Cortex A7 MPcore Technical Reference Manual, Table 11-1 PMU register summary, p 237](#)

Table 11-1: PMU registers

897	0xE04	c9	0	c12	0	PMCR	RW	<i>Performance Monitor Control Register on page 11-7</i>
898	0xE08	c9	0	c14	0	PMUSERENR	RW	User Enable Register, see the <i>ARM Architecture Reference Manual</i>
899-903	0xE0C-0xE1C	-	-	-	-	-	-	Reserved

- The two main registers that we need to access are **PMCR** and **PMUSERENR**
 - ▶ **PMCR**: controls access to the PMU in general
 - ▶ **PMUSERENR**: is the **User Enable Register** that needs to be configured to allow user code to access the PMU

⁰See [Cortex A7 MPcore Technical Reference Manual, Table 11-1 PMU register summary, p 237](#)

Structure of the PMCR register

To enable access to the PMU, we need to access the PMCR register. The **Performance Monitor Control Register (PMCR)** defines the core behaviour of the PMU:



Figure 11-2 Performance Monitor Control Register bit assignments

⁰See [Cortex A7 MPCore Technical Reference Manual, Figure 11-2 Performance Monitor Control Register bit assignments, p 240](#)

The bits in the PMCR

Table 11-2 PMCR bit assignments (continued)

Bits	Name	Function
[4]	X	Export enable. This bit permits events to be exported to another debug device, such as a trace macrocell, over an event bus: 0 Export of events is disabled. This is the reset value. 1 Export of events is enabled. This bit is read/write.
[3]	D	Clock divider: 0 When enabled, PMCCNTR counts every clock cycle. This is the reset value. 1 When enabled, PMCCNTR counts once every 64 clock cycles. This bit is read/write.
[2]	C	Clock counter reset: 0 No action. This is the reset value. 1 Reset PMCCNTR to 0. This bit is write-only, and always RAZ.
[1]	P	Event counter reset: 0 No action. This is the reset value. 1 Reset all event counters, not including PMCCNTR, to 0. In Non-secure modes other than Hyp mode, writing a 1 to this bit does not reset event counters that the HDCR.HPMN field reserves for Hyp mode use. See <i>Hyp Debug Control Register</i> on page 4-68. In Secure state and Hyp mode, writing a 1 to this bit resets all event counters. This bit is write-only, and always RAZ.
[0]	E	Enable bit. Performance monitor overflow IRQs are only signaled when the enable bit is set to 1. 0 All counters, including PMCCNTR, are disabled. This is the reset value. 1 All counters are enabled. This bit is read/write.

Configuring the PMCR register

We are almost there!

The **encoding** for the PMCR register is (see Table 11-1): `c9, c12, 0`

We now configure the PMCR by setting the **E**, **P**, **C**, and **X** bits.

These are bits 0, 1, 2, and 4 in the PMCR register.

This means we need a bitmask of `0b00010111` or `0x17`.

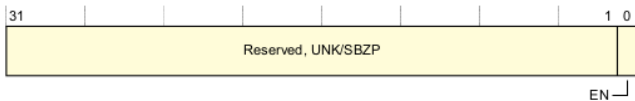
Here is the code:

```
mov r2, #0x17          @ store bitmask 0x17 in reg r2
mcr p15, 0, r2, c9, c12, 0 @ transfer to PMCR
```

NB: For longer running programs you probably also want to enable the **D** bit, which divides the cycle counter by 64!

The PMUSERENR register

The PMUSERENR bit assignments are:



Bits[31:1] Reserved, UNK/SBZP.

EN, bit[0] User mode access enable bit. The possible values of this bit are:

- 0** User mode access to the Performance Monitors disabled.
- 1** User mode access to the Performance Monitors enabled.

Some MCR and MRC instruction accesses to the Performance Monitors are UNDEFINED in User mode when the EN bit is set to 0. For more information, see [Access permissions on page C12-2330](#).

Accessing the PMUSERENR

To access the PMUSERENR, read or write the CP15 registers with <opc1> set to 0, <CRn> set to c9, <CRm> set to c14, and <opc2> set to 0. For example:

```
MRC p15, 0, <Rt>, c9, c14, 0 : Read PMUSERENR into Rt
MCR p15, 0, <Rt>, c9, c14, 0 : Write Rt to PMUSERENR
```

⁰See [ARM Architecture Reference Manual Cortex-A7, Sec B6.1.81, PMUSERENR Performance Monitors User Enable Register, p 1924](#)

Enabling access to the PMU

We can enable access to the PMU from “user space”, from normal applications that are running outside the Linux “kernel space”, by setting the lowest bit in the `PMUSERENR`:

```
mov r2, #0x01 @ store bitmask 0x01 in reg r2
mcr p15, 0, r2, c9, c14, 0 @ transfer r2 to PMUSERENR
```

The `MCR` instruction transfers a value in a register to the co-processor. To find the **encoding** of the `PMUSERENR` we look up Table 11-1:
`c9, c14, 0`

⁰See also [ARM Architecture Reference Manual Cortex-A7, Sec B5.8.2, Table B5-11: Summary of PMSA CP15 register descriptions, p 1796](#)

Enabling access to the PMU

We also need to configure the following registers

- **PMCNTENSET**: Count Enable Set Register¹:

Purpose: The PMCNTENSET register enables the Cycle Count Register, PMCCNTR, and any implemented event counters, PMNx. Reading this register shows which counters are enabled. This register is a Performance Monitors register.

- **PMOVSr**: Overflow Status Register **PMCNTENSET**: Count Enable Set Register²:

Purpose: The PMOVSr holds the state of the overflow bits for:

- ▶ *the Cycle Count Register, PMCCNTR*
- ▶ *each of the implemented event counters, PMNx.*

Software must write to this register to clear these bits.

This register is a Performance Monitors register.

¹See [ARM Architecture Reference Manual Cortex-A7, Sec B6.1.74, p 1910](#)

²See [ARM Architecture Reference Manual Cortex-A7, Sec B6.1.78, p 1908](#)

Table 11-1: PMCNTENSET and PMOVSR registers

We now have to find the register encodings for PMCNTENSET and PMOVSR.

Table 11-1 PMU register summary (continued)

Register number	Offset	CRn	Op1	CRm	Op2	Name	Type	Description
800	0xC80	c9	0	c12	3	PMOVSR	RW	Overflow Flag Status Register, see the <i>ARM Architecture Reference Manual</i>
801-807	0xC84-0xC9C	-	-	-	-	-	-	Reserved
768	0xC00	c9	0	c12	1	PMCNTENSET	RW	Count Enable Set Register, see the <i>ARM Architecture Reference Manual</i>
769-775	0xC04-0xC1C	-	-	-	-	-	-	Reserved
776	0xC20	c9	0	c12	2	PMCNTENCLR	RW	Count Enable Clear Register, see the <i>ARM Architecture Reference Manual</i>
777-783	0xC24-0xC3C	-	-	-	-	-	-	Reserved

²See either [Cortex A7 MPcore Technical Reference Manual, Figure 11-2 Performance Monitor Control Register bit assignments, p 240](#) or [ARM Architecture Reference Manual Cortex-A7, Sec B5.8.2, Table B5-11: Summary of PMSA CP15 register descriptions, p 1796](#)

Bits in `PMCNTENSET` and `PMOVSr` registers

The `PMCNTENSET` register enables the Cycle Count Register, `PMCCNTR`, and any implemented event counters, `PMNx`³

The `PMCNTENSET` register bit assignments are:

31	30	N	N-1	0
C	Reserved, RAZ/WI		Event counter enable bits, P_x , for $x = 0$ to $(N-1)$	

The `PMOVSr` holds the state of the overflow bit for: (i) the Cycle Count Register, `PMCCNTR`; (ii) each of the implemented event counters, `PMNx`.⁴

The `PMOVSr` bit assignments are:

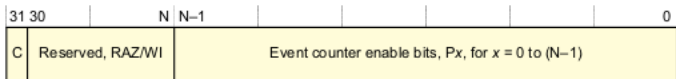
31	30	N	N-1	0
C	Reserved, RAZ/WI		Event counter overflow bits, P_x , for $x = 0$ to $(N-1)$	

³See [ARM Architecture Reference Manual Cortex-A7, Sec B4.1.116, p 1676](#)

Bits in PMCNTENSET and PMOVSR registers

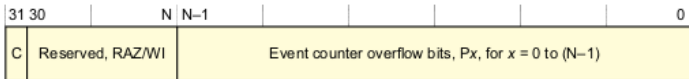
The PMCNTENSET register enables the Cycle Count Register, PMCCNTR, and any implemented event counters, PMNx³

The PMCNTENSET register bit assignments are:



The PMOVSR holds the state of the overflow bit for: (i) the Cycle Count Register, PMCCNTR; (ii) each of the implemented event counters, PMNx.⁴

The PMOVSR bit assignments are:



³See [ARM Architecture Reference Manual Cortex-A7, Sec B4.1.116, p 1676](#)

⁴See [ARM Architecture Reference Manual Cortex-A7, Sec B4.1.116, p 1685](#)

Enabling access to the PMU

Almost there!

Both registers hold bitmasks over the event counters, to enable them and to control overflow.

We want to turn on the bit for every counter.

We have 4 counters in total, so we need to set the 4 least significant bits: we need a bitmask of `0b1111` or `0x0f`

Finally, here is the code to set the `PMCNTENSET` and `PMOVSER` registers:

```
mov r2, #0x0f          @ store bitmask 0x0f in reg r2
mcr p15, 0, r2, c9, c12, 1 @ transfer to PMCNTENSET
mov r2, #0x0f          @ store bitmask 0x0f in reg r2
mcr p15, 0, r2, c9, c12, 3 @ transfer to PMOVSER
```

Step 3: Defining what to monitor

- Now that the PMU is enabled we need to decide what we want to monitor
- The PMU contains one cycle counter register, which we can use without special configuration: `PMCCNTR`
- The PMU contains 4 configurable counter registers
- For each of these registers we need to specify an **event type** to monitor

Table 16-1: PMU monitor events

Table 16-1 Performance monitor events

Number	Event counted
0x00	Software increment of the Software Increment Register
0x01	Instruction fetch that causes a Level 1 instruction cache refill
0x02	Instruction fetch that causes a Level 1 instruction TLB refill
0x03	Data Read or Write operation that causes a Level 1 instruction TLB refill
0x04	Data Read or Write operation that causes a Level 1 data cache access
0x05	Data Read or Write operation that causes a Level 1 data TLB refill
0x06	Memory-reading instruction executed
0x07	Memory-writing instruction executed
0x09	Exception taken
0x0A	Exception return executed
0x0B	Instruction that writes to the Context ID register
0x0C	Software change of program counter
0x0D	Immediate branch instruction executed
0x0F	Unaligned load or store
0x10	Branch mispredicted or not predicted
0x11	Cycle count; the register is incremented on every cycle

⁴From [ARM Cortex-A Programmer's Guide, Table 16-1, p222](#)

Table 16-1: PMU monitor events

0x11	Cycle count; the register is incremented on every cycle
0x12	Predictable branch speculatively executed
0x13	Data memory access
0x14	Level 1 instruction cache access
0x15	Level 1 data cache write-back
0x16	Level 1 data cache write-back
0x17	Level 2 data cache refill
0x18	Level 2 data cache write-back
0x19	Bus access
0x1A	Local memory error
0x1B	Instruction speculatively executed
0x1C	Instruction write to TTBR
0x1D	Bus cycle
0x1E-0x3F	Reserved

⁴From [ARM Cortex-A Programmer's Guide, Table 16-1, p222](#)

Defining what to monitor

We can define the events we want to monitor like this:

```
mov r2, #0x00          @ counter #0
mcr p15, 0, r2, c9, c12, 5 @ transfer to PMSELR
mov r2, #0x11          @ event type #11: cycle count
mcr p15, 0, r2, c9, c13, 1 @ transfer to PMXEVTYPER
```

The first 2 lines identify counter no. 0 (0x00) as the counter we are configuring.

The next 2 lines specify that this counter should monitor event no. 0x11: instruction cycles.

Defining what to monitor

We can define the events we want to monitor like this:

```
mov r2, #0x00          @ counter #0
mcr p15, 0, r2, c9, c12, 5 @ transfer to PMSELR
mov r2, #0x11          @ event type #11: cycle count
mcr p15, 0, r2, c9, c13, 1 @ transfer to PMXEVTYPER
```

The first 2 lines identify counter no. 0 (0x00) as the counter we are configuring.

The next 2 lines specify that this counter should monitor event no. 0x11: instruction cycles.

The complete kernel module

```
// 1. Enable "User Enable Register"
asm volatile("mcr_p15,_0,_%0,_c9,_c14,_0\n\t" :: "r" (0
    x00000001));

// 2. Reset Performance Monitor Control Register (PMCR), Count
    Enable Set Register, and Overflow Flag Status Register
asm volatile ("mcr_p15,_0,_%0,_c9,_c12,_0\n\t" :: "r" (0
    x00000017));
asm volatile ("mcr_p15,_0,_%0,_c9,_c12,_1\n\t" :: "r" (0
    x8000000f));
asm volatile ("mcr_p15,_0,_%0,_c9,_c12,_3\n\t" :: "r" (0
    x8000000f));

// 3. Disable Interrupt Enable Clear Register
asm volatile("mcr_p15,_0,_%0,_c9,_c14,_2\n\t" :: "r" (~0));

// 4. Read how many event counters exist
asm volatile("mrc_p15,_0,_%0,_c9,_c12,_0\n\t" : "=r" (v)); //
    Read PMCR
printk("pmon_init():_have_%d_configurable_event_counters.\n", (
    v >> 11) & 0x1f);
```



Build the module

You first need to download the kernel sources.

To build the module, get the sample sources from [PMU_pmuon](#) and do this:

```
sudo make clean
sudo make
sudo insmod ./pmuon.ko
dmesg | tail
sudo rmmod pmuon
```

Step 4: Use the PMU in a user program

First we define macros for assembler 1-liners, which reset all counters (by writing to `PMCR`) and read the counters from the PMU:

```
#define armv7_reset_counters \
asm volatile ("mcr_p15,_0,_%0,_c9,_c12,_0\n\t" :: "r"(0
             x00000017)) /* write to PMCR */

#define armv7_read_ccr( val ) \
asm volatile("mrc_{}_p15,_0,_%0,_c9,_c13,_0" : "=r"(val)
            )

#define armv7_read_cr0( val ) \
asm volatile("mrc_{}_p15,_0,_%0,_c9,_c12,_5" :: "r"(0x00
            )); /* select counter #0 */ \
asm volatile("mrc_{}_p15,_0,_%0,_c9,_c13,_2" : "=r"(val)
            ) /* read its value */
```

Measuring a simple C loop

The core of our user program is a counting loop:

```
armv7_reset_counters;  
armv7_read_ccr( before_ccr );  
armv7_read_cr0( before_cr0 );  
  
for (i=0; i<n; i++ ) /* nothing */ ; // code to measure  
  
armv7_read_ccr( after_ccr );  
armv7_read_cr0( after_cr0 );
```

Example: running the measurement

```
> gcc -DCP15 -o rpi2-pmu01 rpi2-pmu01.c
> sudo ./rpi2-pmu01 10
Raspberry Pi 2 performance monitoring, using CP15 interface
The result is: 10
ccr: 338 (before: 0 after: 338) CYCLES
cr0: 338 (before: 6 after: 344) CYCLES
cr1: 12 (before: 0 after: 12) BRANCHES
cr2: 48 (before: 3 after: 51) CACHE HITS (Data read or write
      operation that causes a cache access at (at least) the
      lowest level of data or unified cache)
cr3: 32 (before: 0 after: 32) CACHE MISSES (Data read
      architecturally executed)
PMCR=41072011
Done.
```

Measuring assembler code

This is an assembler version of the counting loop:

```
armv7_reset_counters;
armv7_read_ccr( before_ccr );
armv7_read_cr0( before_cr0 );

asm volatile( /* inline assembler version of a counting loop */
  "_measure_me_asm_%=: \n"
  "\t_____MOVSW_____R3, _#0x00_____@_initialise_counter_
  register\n"
  "\t_____B_____TEST%= _@_uncond._jump\n"
  "LOOP%=: _____@_loop_over_counter_R3\
  n"
  "\t_____ADD_____R3, _R3, _#1_____@_increment_counter
  _____\n"
  "TEST%=: _____CMP_____R3, _%[n]_____@_test_end_value\n"
  "\t_____BLT_____LOOP%= \n"
  "\t_____MOV_____ %[res], _R3_____@_done_\n"
  : [res] "=r" (i) : [n] "r" (n) : "r3", "cc");

armv7_read_ccr( after_ccr );
armv7_read_cr0( after_cr0 );
```

Output

```
> gcc -DCP15 -o rpi2-pmu01 rpi2-pmu01.c
> sudo ./rpi2-pmu01 10
Raspberry Pi 2 performance monitoring, using CP15 interface
The result is: 10
ccr: 249 (before: 0 after: 249) CYCLES
cr0: 249 (before: 6 after: 255) CYCLES
cr1: 12 (before: 0 after: 12) BRANCHES
cr2: 7 (before: 3 after: 10) CACHE HITS (Data read or write
      operation that causes a cache access at (at least) the
      lowest level of data or unified cache)
cr3: 1 (before: 0 after: 1) CACHE MISSES (Data read
      architecturally executed)
PMCR=41072011
Done.
```

NB: we get precise runtime in machine-cycles; because we execute the loop 10 times (plus entry and exit), the branch counter shows 12; most operations work in registers, only a few memory access are needed and most of them can use the cache

```

armv7_reset_counters;
armv7_read_ccr( before_ccr );
armv7_read_cr0( before_cr0 );

asm volatile( /* inline assembler version of a counting loop
              with bad branch prediction */
    "_measure_me_asm_%=: \n"
    "\t_____MOVSW_____R3, _#0x00_____@_initialise_counter_
      register\n"
    "TEST%=: _____CMP_____R3, _%[n]_____@_test_end_value\n"
    "\t_____BGE_____LEAVE%=_____@_leave_loop_(BAD_
      BRANCH_PRED!) _\n"
    "\t_____ADD_____R3, _R3, _#1_____@_increment_counter
      _____\n"
    "\t_____B_____TEST%=_____@_unconditional_jump_\n"
    "LEAVE%=: _____MOV_____ %[res], _R3_____@_done_\n"
    : [res] "=r" (i) : [n] "r" (n) : "r3", "cc");

armv7_read_ccr( after_ccr );
armv7_read_cr0( after_cr0 );

```

Output

```
> gcc -DCP15 -o rpi2-pmu01 rpi2-pmu01.c
> sudo ./rpi2-pmu01 10
Raspberry Pi 2 performance monitoring, using CP15 interface
The result is: 10
ccr: 116 (before: 0 after: 116) CYCLES
cr0: 116 (before: 6 after: 122) CYCLES
cr1: 21 (before: 0 after: 21) BRANCHES
cr2: 7 (before: 3 after: 10) CACHE HITS (Data read or write
operation that causes a cache access at (at least) the
lowest level of data or unified cache)
cr3: 1 (before: 0 after: 1) CACHE MISSES (Data read
architecturally executed)
PMCR=41072011
Done.
```

NB: In this case we have 21 rather than 12 branches, for the same kind of counting loop; this is because each iteration resulted in a mis-predicted branch, which was partially executed by the processor-pipeline, but then had to be aborted.

A larger user program: sum-and-average

Code example: `sumav3_asm_pmu.c`

Summary

- The ARM Cortex-A7 has an on-chip co-processor for hardware performance monitoring (PMU)
- The PMU can be configured to count a range of low-level events, e.g. cycles, branches, cache hits
- The PMU needs to be enabled from within a kernel module, so that user space programs can access it
- Once configured, inline assembler instructions can be used to start/stop counting and read values
- The relevant assembler instructions are `MCR` and `MRC`, with a bespoke formatting of specifying registers on the CP15 co-processor (and on other on-chip co-processor)