C Programming
Using GPIO with `/dev/mem`

Adam Sampson (based on material by Greg Michaelson)
School of Mathematical and Computer Sciences
Heriot-Watt University

Raspberry Pi GPIO

- The Raspberry Pi is built around a Broadcom “system on a chip”, containing the CPU and various peripherals

- Has 54 **General Purpose I/O** lines
  - GPIO 0 to GPIO 53
- Can be used for simple voltage input or output
- Many have specialised uses too, e.g. camera, serial, audio

Function selection

- The GPIO controller has 41 registers, each 32 bits
  - Registers are mapped to **physical** memory addresses
  - Addressed by offset *from the address of the first register*

- The first 6 registers select the function of each pin
  - Input, output...

number	name	pins	offset
0	GPFSEL0	0-9	0x00
1	GPFSEL1	10-19	0x04
2	GPFSEL2	20-29	0x08
3	GPFSEL3	30-39	0x0C
4	GPFSEL4	40-49	0x10
5	GPFSEL5	50-53	0x14

Function selection

- In each register, each pin has 3 bits for its function

GPFSELx	0	1	2	3	4	5	GPFSELx	0	1	2	3	4	5
bits	pin	pin	pin	pin	pin		bits	pin	pin	pin	pin	pin	pin
$b_{29}b_{28}b_{27}$	9	19	29	39	49	-	$b_{14}b_{13}b_{12}$	4	14	24	34	44	-
$b_{26}b_{25}b_{24}$	8	18	28	38	48	-	$b_{11}b_{10}b_{09}$	3	13	23	33	43	53
$b_{23}b_{22}b_{21}$	7	17	27	37	47	-	$b_{08}b_{07}b_{06}$	2	12	22	32	42	52
$b_{20}b_{19}b_{18}$	6	16	26	36	46	-	$b_{05}b_{4}b_{03}$	1	11	21	31	41	51
$b_{17}b_{16}b_{15}$	5	15	25	35	45	-	$b_{02}b_{01}b_{00}$	0	10	20	30	40	50

- e.g. pin 13 = GPFSEL1 bits 9-11, pin 27 = GPFSEL2 bits 21-23

Function selection

- To make a pin act as a simple input or output, set its bits in the corresponding GPFSELx register
  - Input: 000
  - Output: 001
- e.g. to set pin 25 to output:
  - Set GPFSEL2 to 001 << 15 = 001 000 000 000 000 000
  - (The shift moves bits 0-2 in 001 to bits 15-17)

Setting and clearing pins

- To set or clear pins (turn outputs on/off), set the corresponding bit in the `GPSETx` or `GPCLRx` registers

number	name	pins	offset
7	GPSET0	0-31	0x1C
8	GPSET1	32-53	0x20
10	GPCLR0	0-31	0x28
11	GPCLR1	32-53	0x2C

- `GPSET0`/`GPCLR0` affect pins 0-31
- `GPSET1`/`GPCLR1` affect pins 32-53

Constants for register offsets

```
#define GPIO_GPFSEL0    0
#define GPIO_GPFSEL1    1
#define GPIO_GPFSEL2    2
#define GPIO_GPFSEL3    3
#define GPIO_GPFSEL4    4
#define GPIO_GPFSEL5    5

#define GPIO_GPSET0     7
#define GPIO_GPSET1     8

#define GPIO_GPCLR0     10
#define GPIO_GPCLR1     11
```

Accessing registers

- We're going to get a pointer to the first GPIO register – then we can access them like an array...

- Start of GPIO registers in memory will be:
  `volatile uint32_t *gpio;`
  - i.e pointer to 32-bit integer = groups of 4 bytes
- Then GPIO register `i` is `gpio[i]`

The volatile qualifier

- **`volatile`** `uint32_t *gpio;`
- `volatile` means “do exactly as I say with operations on this pointer”
- Normally the compiler is allowed to reorder code in order to make it run faster – so it might swap writes to memory around, for example, provided it gets the same result…
- We don't want it to do that with hardware accesses!
  - (e.g. it might set level before setting mode)

Flashing the LED

- The green LED on the Raspberry Pi board is connected to GPIO pin 47
  - It's “active low” – low output = LED on
- Function is controlled by `GPFSEL4` bits 21-23
- First we need to make it an output:

```
gpio[GPIO_GPFSEL4] = 1 << 21;
```

Flashing the LED

- Pin 47's value is controlled by bit 15 in `GPSET1`/`GPCLR1`
- To flash the LED:

```
while (1) {
    gpio[GPIO_GPCLR1] = 1 << 15; // on
    sleep(1);
    gpio[GPIO_GPSET1] = 1 << 15; // off
    sleep(1);
}
```

Flashing the LED

- Pin 47's value is controlled by bit 15 in `GPSET1`/`GPCLR1`
- To flash the LED:

```
while (1) {
    gpio[GPIO_GPCLR                        n
    sleep(1);
    gpio[GPIO_GPS                          ff
    sleep(1);
}
```

A standard Unix function that sleeps for N seconds

(There's also `usleep`, which sleeps for N microseconds)

Mapping the registers

- But we need to set `gpio` to point at the first GPIO register in memory…

- The Raspberry Pi runs Linux
- Each program runs in own virtual address space –
  it can’t directly access physical memory
- We must ask Linux to map it for us

Mapping the registers

- In Unix, devices usually look like files
- Linux exposes physical I/O memory as a **character device** file: `/dev/mem`
- Open the file using the `open` system call:

```
int fd = open("/dev/mem", O_RDWR | O_SYNC);
if (fd < 0) {
    printf("can't open /dev/mem\n");
    exit(1);
}
```

Using | to combine flags for read/write and synchronise

Mapping the registers

- In Unix, devices usually look
- Linux exposes physical I/O m
  file: `/dev/mem`
- Open the file using th` open`

Unix's open is a lower-level facility than the C library's `fopen`

Unix represents open files as integer "file handles" (stdin is 0, stdout is 1...); if it returns -1, opening failed

```
int fd = open("/dev/mem", O_RDWR | O_SYNC);
if (fd < 0) {
    printf("can't open /dev/mem\n");
    exit(1);
}
```

Mapping the registers

- The Unix mmap function maps part of an open file into virtual memory, returning the address

```
#define GPIO_BASE 0x3f200000    // start of regs
#define GPIO_LENGTH (4*1024)    // length in bytes
...
gpio = mmap(NULL, GPIO_LENGTH,
            PROT_READ | PROT_WRITE,
            MAP_SHARED, fd, GPIO_BASE);
if (gpio == (void *) -1) {
    printf("can't mmap\n");
    exit(1);
}
```

You can do this with regular files too – convenient if you want to work with a file's contents as if it were an array

See the full code in this file.

Don't forget the `#includes`!

`LEDblink.c`

Blinking the LED