F28HS2 Hardware-Software Interface

Lecture 3 - Programming in C 3

Low level programming in C

- want to control hardware devices
- devices mapped into memory
- accessed at specific memory addresses
- set memory locations to bit sequences to:
  - configure devices
  - send/receive data

Declarations & space

- amount of space for type depends on
  - type
  - platform
  - compiler
- measure type sizes in 8 bit bytes

`int sizeof(`*`type`*`)`

- returns number of bytes for *type*

Sizes of types

e.g. typesize.c

```
#include <stdio.h>
main(int argc,char ** argv)
{  printf("char: %d\n",sizeof(char));
   printf("int: %d\n",sizeof(int));
   printf("short: %d\n",sizeof(short));
   printf("long: %d\n",sizeof(long));
   printf("float: %d\n",sizeof(float));
   printf("double: %d\n", sizeof(double));
}
```

Sizes of types

- on 64 bit PC Linux

```
$ typesize
char: 1
int: 4
short: 2
long: 4
float: 4
double: 8
```

Sizes of types

- on Raspbian

```
$ typesize
char: 1
int: 4
short: 1
long: 4
float: 4
double: 8
```

Bits, bytes and hexadecimal

binary	hex	decimal	binary	hex	decimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	a	10
0011	3	3	1011	b	11
0100	4	4	1100	c	12
0101	5	5	1101	d	13
0110	6	6	1110	e	14
0111	7	7	1111	f	15

Bits, bytes and hexadecimal

- hexadecimal: `0X`$h_1h_2...h_N$
  - where h_i == `0..9 A..F`
- can think about hex as bit settings not numbers
- byte is 2 * 4 bits
- `0x`h_1h_0
  - h_1 is bits 7-4
  - h_0 is bits 3-0
- e.g. `0xff` == 1111 1111 == all bits 1

  `0x00` == 0000 0000 == all bits 0

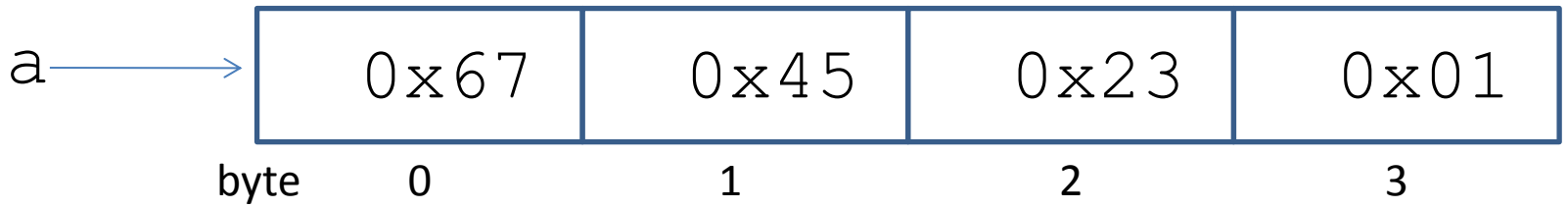
  `0x65` == 0100 0101 == bits 6, 2 and 0 set == ‘e’

Memory allocation to values

- 2 digits == 8 bits == 1 byte
- NB values stored from most significant to least significant byte

e.g. `int a;`

```
a = 0x01234567;
```



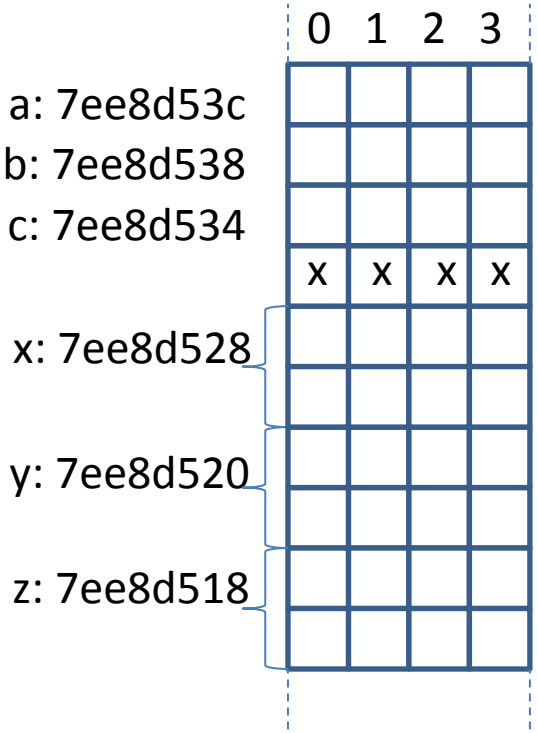
Memory allocation to declarations

- e.g. addrs.c

```
#include <stdio.h>
main(int argc,char ** argv)
{ int a,b,c;
  double x,y,z;
  printf("&a: %x, &b: %x, &c: %x\n",&a,&b,&c);
  printf("&x: %x, &y: %x, &z: %x\n",&x,&y,&z);
}
$ addrs
&a: 7ee8d53c, &b: 7ee8d538, &c: 7ee8d534
&x: 7ee8d528, &y: 7ee8d520, &z: 7ee8d518
```

Memory allocation to declarations

- 3c-38 == 4
- 38-34 == 4
- 34-28 == 12 == 8+4?
- 64 bit machine
- padding to 8 byte boundary
- 28-20 == 8
- 20-18 == 8

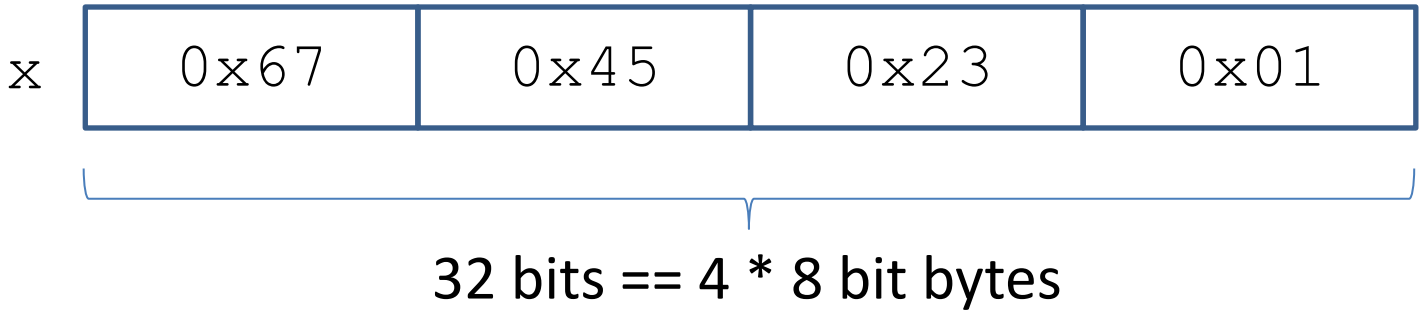


Accessing bytes in C

- can access individual bytes of a 32 bit integer by:
  - casting to:
    - array of 4 chars
    - struct of 4 chars - later
  - addressing each char individually

Casting ints

```
int x;
x = 0x01234567;
```



Casting int to char

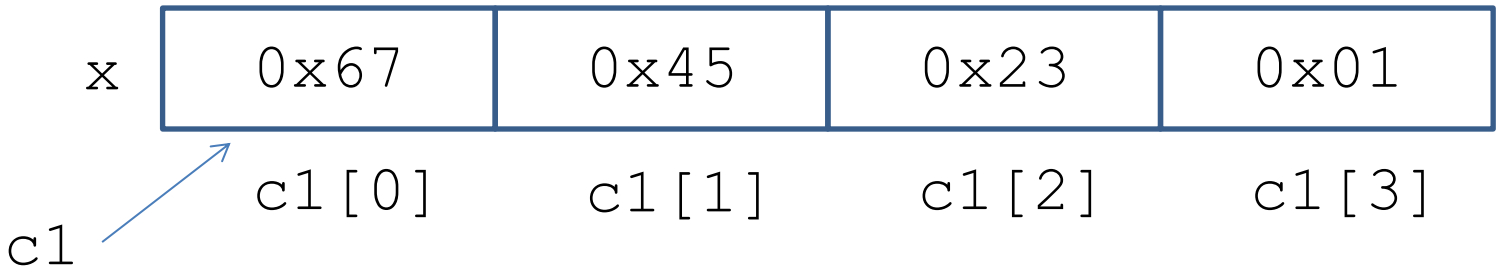
```
char * c1;
c1 = (char *)&x;
printf("%x %x %x %x\n",
       c1[0],c1[1],c1[2],c1[3]);
```

– c now points at x's space

➔ 67 45 23 1

Casting int to to char

- `&x` returns address of 1st byte of `int`
- `(char *)` coerces address of 1st byte of `int` to address of 1st byte of array of `char`
  - `c1[0]` is 1st byte of `x` == `0x67`
  - `c1[1]` is 2nd byte of `x` == `0x45` etc



Manipulating bit patterns in C

- in CPU (ARM, Intel) registers are 32 bit
- so addressing individual bytes involves manipulating a 32 bit register
- C exposes details of low level bit manipulation
- use logical & shift operations
- with hexadecimal representations of bit patterns in bytes

Logical operations

X	Y	X & Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X | Y
0	0	0
0	1	1
1	0	1
1	1	1

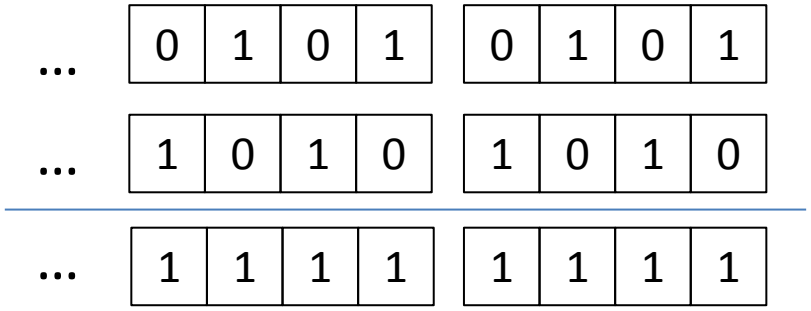
X	Y	X ^ Y
0	0	0
0	1	1
1	0	1
1	1	0

- X & Y == AND == 1 if both 1
  - use 0 in Y to set to 0
- X | Y == OR == 1 if either 1
  - use 1 in Y to set to 1
- X ^ Y == XOR - exclusive OR == 1 if both different
  - use 1 in Y to flip X from 0 to 1 or 1 to 0

Logical operations

- `&`, `|` and `^` operate in parallel across all bits in bytes
- e.g.

```
int x,y;
x = 0x55;
y = 0xaa;
x | y ==> 0xff
```



Masks

- mask
  - bit pattern that affects specific bits
- e.g. to select bytes in 32 bit register with &

```
#define byte0 0x000000ff
```

0000 0000 0000 0000 0000 0000 1111 1111

```
#define byte1 0x0000ff00
```

0000 0000 0000 0000 1111 1111 0000 0000

```
#define byte2 0x00ff0000
```

0000 0000 1111 1111 0000 0000 0000 0000

```
#define byte3 0xff000000
```

1111 1111 0000 0000 0000 0000 0000 0000

Masks

```
int word;
word = 0x61626364;
```

word == 01000001 01000010 01000011 01000100

word & byte0 == 00000000 00000000 00000000 **01000100**

word & byte1 == 00000000 00000000 **01000011** 00000000

word & byte2 == 00000000 **01000010** 00000000 00000000

word & byte3 == **01000001** 00000000 00000000 00000000

Masking bytes - high to low

```
#define BYTE0 0x000000ff
#define BYTE1 0x0000ff00
#define BYTE2 0x00ff0000
#define BYTE3 0xff000000

main()
{   int word;

    word = 0x61626364; /* a b c d */

    printf("%8x\n",word&BYTE0);
    printf("%8x\n",word&BYTE1);
    printf("%8x\n",word&BYTE2);
    printf("%8x\n",word&BYTE3);
}
```

```
==>       64
        6300
      620000
    61000000
```

Shifting

- how to isolate individual bytes?
- shifting
  - move bits in sequence a specified number of places left/right
  - pad to right/left with 0s
- X << Y == shift X left Y places and add Y 0s to right
- X >> Y == shift X right Y places and add Y 0s to left

Selecting bytes: low to high

`word` == 01000001 01000010 01000011 01000100 == 0x61626364

`word & byte0` == 00000000 00000000 00000000 **01000100**

== 0x64

`word >> 8` == **00000000** 01000001 01000010 01000011

== 0x00616263

`(word >> 8) & byte0` == 00000000 00000000 00000000 **01000011**

== 0x63

`(word >> 16)` == **00000000 00000000** 00000001 01000010

== 0x00006162

`(word >> 16) & byte0` == 00000000 00000000 00000000 **01000010**

== 0x62

etc

Selecting bytes: low to high

```
#define BYTE0 0x000000ff
main()
{   int word;
    int i;

    word = 0x61626364; /* a b c d */

    for(i=0;i<4;i++)
    {   printf("%8x\n",word&BYTE0);
        word = word >> 8;
    }
}
```

```
==>         64
            63
            62
            61
```

Selecting bytes: high to low

- mask leftmost byte & shift right 3 bytes
- then shift one byte left

```
#define BYTE3 0xff000000
main()
{   ...

    for(i=0;i<4;i++)
    {   printf("%8x\n",(word&BYTE3)>>24);
        word = word << 8;
    }
}
```

```
==>         61
            62
            63
            64
```

Selecting bits: low to high

```
#define BIT0 0x00000001


main()
{   int n,i;
    printf("enter value> ");
    scanf("%d",&n);
    for(i=0;i<32;i++)
     printf("%3d",i);
    putchar('\n');
    for(i=0;i<32;i++)
    {   if(n&BIT1)printf("  1");
        else printf("  0");
        n = n>>1; }
    putchar('\n');
}
```

```
enter value> 247
  0  1  2  3  4  5  6  7...
  1  1  1  0  1  1  1  1...
```

Selecting bits: high to low

```
#define BIT31 0x80000000

main()
{   int n; int i;
    ...
    for(i=31;i>0;i--)
     printf("%3d",i);
    putchar('\n');
    for(i=0;i<31;i++)
    {   if(n&BIT31)printf("  1");
        else printf("  0");
        n = n<<1; }
    putchar('\n');
}
```

```
enter number> 247
 ...7  6  5  4  3  2  1  0
 ...1  1  1  1  0  1  1  1
```

Raspberry Pi GPIO

- General Purpose I/O chip
  - BCM2853
- connects RPi to external devices
- has 54 general purpose I/O lines
  - GPIO 0 to GPIO 53
- NB many I/O lines have dedicated use in RPi

Raspberry Pi GPIO

- controller has 41 * 32-bit registers
  - registers mapped to memory
  - addressed by offset *from address of first register*
- first 6 registers control pin functions

number	name	pins	offset
0	GPSEL0	0-9	0x00
1	GPSEL1	10-19	0x04
2	GPSEL2	20-29	0x08
3	GPSEL3	30-39	0x0C
4	GPSEL4	40-49	0x10
5	GPSEL5	50-53	0x14

Raspberry Pi GPIO

- in each register, each pin has 3 bits for control

GPSEL ->	0	1	2	3	4	5	GPSEL ->	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 ==GPSEL1 bits 9-11
- e.g. pin 27 == GPSEL2 21-23

Raspberry Pi GPIO

- to set pin to input or output
- set bits in corresponding GSEL register
  - input == 000
  - output == 001
- e.g. to set pin 25 to output
- set GPSEL2 to 001 << 15 ==>

  001 000 000 000 000 000
- i.e. bits 0-2 in 001 now bits 15-17

Raspberry Pi GPIO

- to set or clear pins
- set corresponding bit in set and clear 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

Raspberry Pi GPIO

- definitions

```
#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
```

Raspberry Pi GPIO

- suppose start of GPIO registers in memory is:

```
volatile unsigned int * gpio;
```

- i.e pointer to int == groups of 4 bytes
- so GPIO register `i` is `gpio[i]`

Raspberry Pi GPIO

- e.g. green light on the RPi board is connected to GPIO pin 47
- == GPSEL4 bits 21-23
- to set as output:

```
gpio[GPIO_GPSEL4] = 1 << 21;
```

Raspberry Pi GPIO

- pin 47 controlled by bit 15 in GPSET1 & GPCLR1
- to turn on & off:

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

Raspberry Pi GPIO

- need to set `gpio` to address of first GPIO register in memory
- RPi runs Linux
- each program runs in own virtual address space
- can’t directly access physical memory

Raspberry Pi GPIO

- Linux sees physical memory as a character device file `/dev/mem`
- can open this for access:

```
#include <fcntl.h>
int fd;
...
   if ((fd = open ("/dev/mem",
                   O_RDWR | O_SYNC | O_CLOEXEC) ) < 0)
   {  printf("can't open /dev/mem\n"); exit(0); }
```

- NB use of | to combine flags for read/write, synchronise & close on exit

Raspberry Pi GPIO

- RPi 2 registers start at `0x3f200000`
- map opened `/dev/mem` onto this physical location:

```
#include <sys/mman.h>
#define BLOCK_SIZE (4*1024)
#define GPIO_BASE 0x3f200000
...
   gpio = (uint32_t *)mmap(0, BLOCK_SIZE,
                           PROT_READ|PROT_WRITE,
                           MAP_SHARED, fd, GPIO_BASE) ;
   if((int)gpio==-1)
   {  printf("can't mmap\n"); exit(0); }
```

Raspberry Pi GPIO

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <fcntl.h>
#include <sys/mman.h>

#define BLOCK_SIZE      (4*1024)
#define GPIO_BASE       0x3F200000UL

#define LED_GPFSEL      GPIO_GPFSEL4
#define LED_GPFBIT      21
#define LED_GPSET       GPIO_GPSET1
#define LED_GPCLR       GPIO_GPCLR1
```

Raspberry Pi GPIO

```
#define LED_GPIO_BIT    15
#define GPIO_GPFSEL4    4
#define GPIO_GPSET1     8
#define GPIO_GPCLR1     11

volatile unsigned int* gpio;
volatile unsigned int tim;
```

Raspberry Pi GPIO

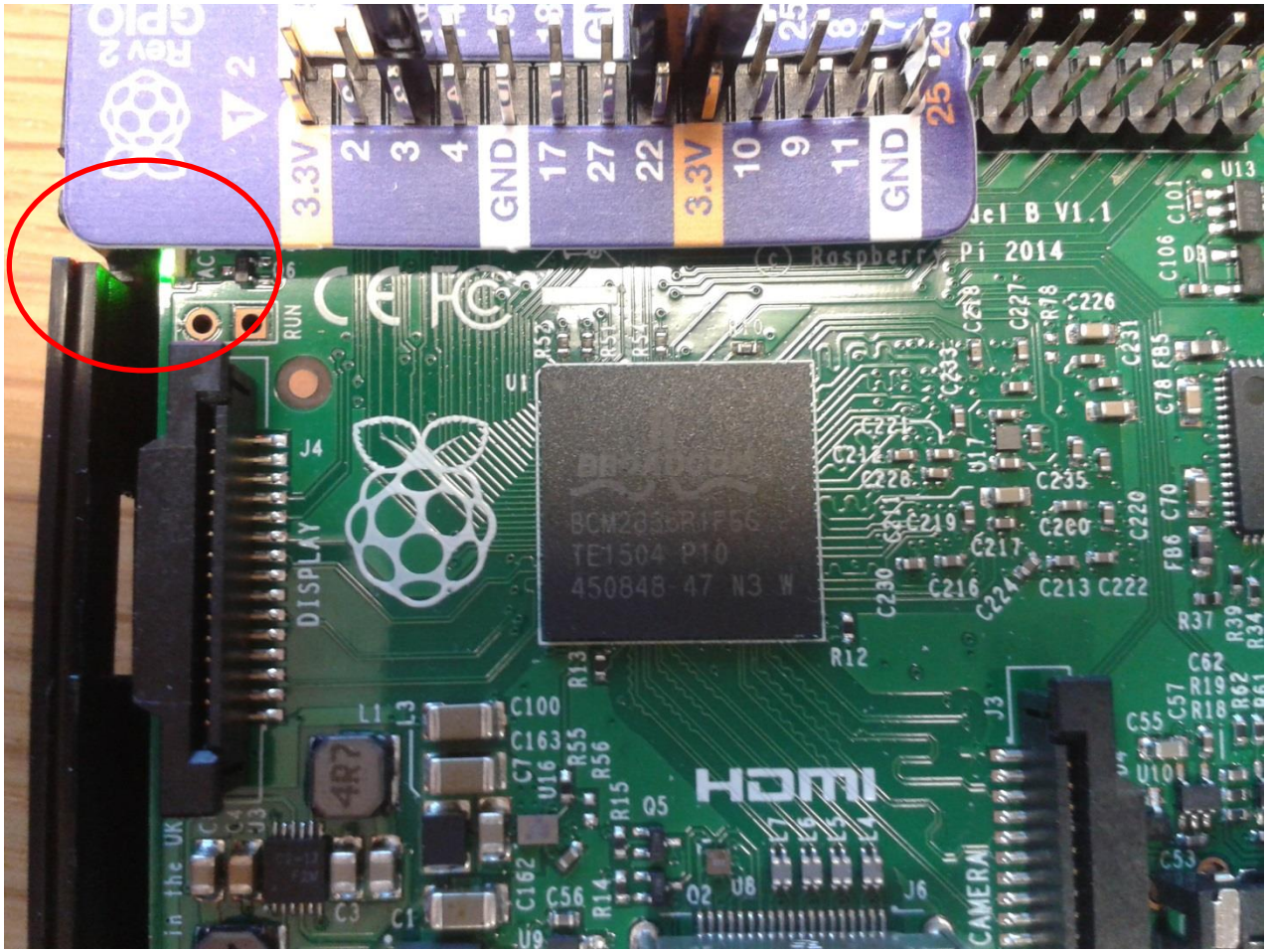
```
int main(void)
{  int fd;

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

   gpio = (uint32_t *)mmap(0, BLOCK_SIZE,
                           PROT_READ|PROT_WRITE,
                           MAP_SHARED, fd, GPIO_BASE) ;
   if((int)gpio==-1)
   {  printf("can't mmap\n"); exit(0); }
```

Raspberry Pi GPIO

```
    gpio[LED_GPFSEL] |= (1 << LED_GPFBIT);
    while(1)
    {   for(tim = 0; tim < 10000000; tim++);
        gpio[LED_GPCLR] = (1 << LED_GPIO_BIT);
        for(tim = 0; tim < 10000000; tim++);
        gpio[LED_GPSET] = (1 << LED_GPIO_BIT);
    }
}
```

- with thanks to:
- for basic program: http://www.valvers.com/open-software/raspberry-pi/step01-bare-metal-programming-in-cpt1/
- for mmap advice, Gordon Henderson: https://projects.drogon.net/raspberry-pi/gpio-examples/tux-crossing/gpio-examples-1-a-single-led/

Raspberry Pi GPIO



Viewing bytes

- `od` *file* - octal dump
- each line
  - byte number + byte representations
- show as hex
  - `od -x` *file*
- show as ASCII
  - `od -c` *file*
- show as hex & ASCII
  - `od -x -c` *file*