

# Real-time Systems D0003E

Lecture 2:  
Bit manipulation and hardware interfacing  
Burn/Wellings ch. 15

## Overview

- bit-manipulation in C
- memory mapped I/O
- port-mapped I/O
- some examples

## Accessing hardware in C

Whenever the CPU finds a read instruction

Whenever the CPU finds a write instruction

Whenever a user types

70 times a second

We'll return to the question of I/O synchronization in a while...

## The naked computer

- Two methods for accessing hardware
- memory mapped I/O
  - hardware looks like memory
- separate I/O-bus
- Approaches look different from software point of view

For now, let's concentrate on how to read and write to I/O ports

## Separate bus architecture

- I/O ports accessed by special I/O instructions
- I/O-port maps to actual hardware registers
  - when writing a port, we actually write to the hardware
- Example: Intel x86 family

## Memory-mapped architecture

- I/O devices appear as plain memory cells
- We write to memory
  - on special addresses

Example: the AVR architecture

## Memory-mapped I/O in C

- If `ptr` contains the address of the port:
  - Reading: `value = *ptr;`
  - Writing: `*ptr = new_val;`
  - Just as reading a regular memory cell
- The **type** of `ptr` must match the **width** of the port:
  - 8 bits: `char *ptr;`
  - 16 bits (on the AVR): `int *ptr;`
- Setting the address (assign value to pointer):  
`char *ptr = (char *)0x1ff;`
- To avoid confusion caused by signed values:  
`unsigned char *ptr = (unsigned char *)0x1ff;`
- To stop the compiler from removing `*ptr = x; x = *ptr; :`  
`volatile char *ptr = (volatile char *)0x1ff;`
- `volatile` tells compiler not to optimize that code

## Memory-mapped I/O

- It is very common to use the preprocessor to simplify port accesses:
 

```
#define SOME_REG *((char *)0x1ff)
...
SOME_REG = 0;
x = SOME_REG & 0x01; /* mask out lsb */
```
- Explanation:
  - `0x1ff` is memory address of hardware
  - cast to char pointer: `(char*) 0x1ff`
  - dereference pointer (we get what it is pointing to):  
`*((char *)0x1ff)`
- Such definitions are usually found in platform-specific `.h` files (see `avr/iom169.h` for example)
- On the AVR - it's even simpler than this!

## Separate bus I/O

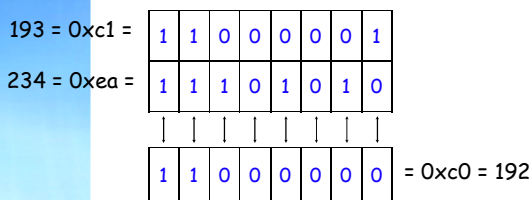
- Requires **inline assembler** instructions (goes beyond C)
  - special instruction to write to I/O-port
- Usually presented as preprocessor macros
- For example, the QNX real-time OS provides a set of macros called `in8`, `out8`, `in16`, `out16`, etc
- Reading a byte from port `0x30d`
  - `unsigned char val = in8(0x30d);`
- Writing a 32-bit word to port `0xf4`
  - `out32(0xf4, expr);`

## Bit operations

- Bitwise AND `(a & b)`
- Bitwise OR `(a | b)`
- Bitwise XOR `(a ^ b)`
- Bitwise NOT `(~ a)`
- Shift bits left `(a << b)`
  - shift a, b bits right
- Shift bits right `(a >> b)`
  - arithmetic shift if type is signed!
    - on some machines... :-)
    - otherwise normal shift
  - arithmetic shift
    - shift in 1's instead of 0's if value negative
    - this will keep the value negative
      - shift divides by two
- use unsigned types to avoid errors

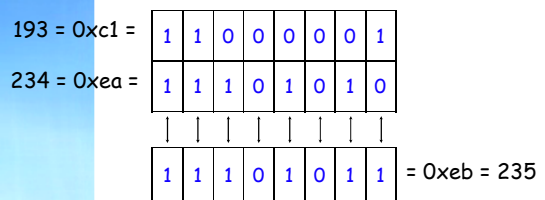
## Bitwise AND

$$193 \& 234 = 192$$



## Bitwise OR

$$193 | 234 = 235$$



## C.f. logical AND, OR

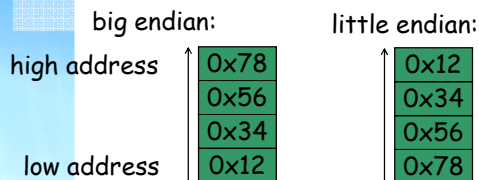
- Don't confuse the bitwise operators `&` and `|` with the logical operators `&&` and `||` that work on integer expressions:
  - `0 && 17 = 0`
  - `31 || -28 = 1`
  - `0x40 && 0x02 = 1`
- These operators only distinguish between the values zero and non-zero
  - zero means false
  - non-zero means true

## Big vs. little endian

- "first" byte of a word?
  - to the right or to the left?
  - low memory address or high memory address?
- Example:
  - `long a = 0x12345678; //assign value`
  - `char *p = (char *)&a; //pointer to "first" byte`
  - `printf("%02X\n", *p); //print "first" byte`
- printout depends on architecture:
  - *little-endian* machine (Intel): `0x78`
  - *big-endian* machine (most others): `0x12`
- treating words as a sequence of bytes (and vice versa)!
  - Use caution!

## Big vs. little endian

- Example:
  - `long a = 0x12345678; //assign value`



## Examples

- Testing a certain bit (memory-mapped I/O)
 

```

#define OK_BIT 5
#define OK_MASK (1 << OK_BIT) //create mask with 1 in pos. 5
volatile char *status_reg = (volatile char *)0x34c;
//0x34 is address of status reg.

if (*status_reg & OK_MASK) // read register and mask out bit
    // bit 5 was set in status register
else
    // it was not
      
```

## Examples

- Setting a certain bit (memory-mapped I/O)
 

```

#define OK_BIT 5
#define OK_MASK (1 << OK_BIT)
volatile char *reg = (volatile char *)0x34c;

*reg = *reg | OK_MASK;
// read reg. and set bit, write back register
      
```
- Note that single bits can't be written - all 8 bits of a byte must be written at once

## Examples

- Testing a certain bit (separate I/O bus)
 

```

#define OK_BIT 5
#define OK_MASK (1 << OK_BIT)
#define STATUS_REG 0x34c

if (in8(STATUS_REG) & OK_MASK)
    // bit 5 was set in status register
else
    // it was not
      
```

## Examples

- Setting a certain bit (separate I/O bus)
 

```

#define OK_BIT 5
#define OK_MASK (1 << OK_BIT)
#define RW_REG 0x34c

*out8(RW_REG, in8(RW_REG) | OK_MASK);

```

## Note:

- There's nothing that guarantees that reading and writing at the same address will refer to the same register!
  - completely up to the hardware
  - can do whatever it wants with value
- For example, some devices map a status register (for reading) and a command register (for writing) to the same address:
 

```

#define IS_READY (1 << 5)
#define CONVERT (1 << 5)
#define STATUS_REG *((char *)0x34c)
#define CMD_REG *((char *)0x34c)

if (STATUS_REG & IS_READY) CMD_REG = CONVERT;

```

## Shadowing

- The correct way to update such overlaid registers is to use a shadow variable:
 

```

#define CONVERT (1 << 5)
#define CMD_REG *((char *)0x34c)
char cmd_shadow;
...
cmd_shadow = cmd_shadow | CONVERT;
CMD_REG = cmd_shadow;

```
- It's important to let all changes to `CMD_REG` be reflected in `cmd_shadow` as well
- [Fortunately, all ports in the AVR are read/write]
- Tip: C provides a convenient assignment syntax
  - `var op= expr;` means `var = var op expr;`

## Single write

- In many cases, output ports are written as single values, so reading of the same port is not required:
 

```

#define CTRL (1 << 3)
#define SIZE1 (1 << 4)
#define SIZE2 (2 << 4)
#define SIZE3 (3 << 4)
#define EXTRAFLAG (1 << 6)
...
CMD_REG = EXTRAFLAG | SIZE2 | CTRL;

```

## Reading multi-bit values

- Masking out a multi-bit value, and then shifting:
  - get two-bit value in bits 4-6:
 

```

#define SIZEMASK (3 << 4) // two 1's shifted 4 bits
#define DATA_REG *((char *)0x34c)
...
//switch on value of the two bits
switch ((DATA_REG & SIZEMASK) >> 4) {
  case 1: ...
  case 2: ...
  case 3: ...
}

```

## A slightly bigger example

- In the port pointed to by `ptr`, set `n` bits starting at bit number `p` to the value of `x`

```

void setbits(int *ptr, int n, int p, int x) {
  int mask = ~(~0 << n) << p; // the mask
  int data = (x << p) & mask; // data to be set
  *ptr = *ptr & ~mask; // clear current bits
  *ptr = *ptr | data; // OR in the data
}

```
- [Notice the assumption here that `ptr` is read/write]

## A slightly bigger example

assume p=2 and n=3

```

~(~0 << n) << p
76543210
0      00000000
~0     11111111
~0 << n 11111000
~(~0 << n) 00000111
~(~0 << n) << p 00011100
    
```

assume x=5

```

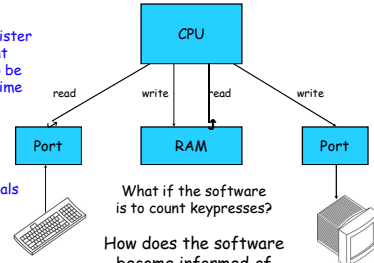
(x << p) & mask      76543210
x                   00000101
x << p              00010100
x << p & mask      00010100
    
```

```

void setbits(int *ptr, int n, int p, int x) {
int mask = ~(~0 << n) << p; // the mask
int data = (x << p) & mask; // data to be set
*ptr = *ptr & ~mask; // clear current bits
*ptr = *ptr | data; // OR in the data
}
    
```

## Back to I/O synchronization

Port I/O register allows current key status to be read at any time



Keyboard electronics signals all key changes to the port

What if the software is to count keypresses?  
How does the software become informed of changes in key status?

## Poor man's solution

- Read the key status register over and over again, until two subsequent values are not equal
 

```

int old = KEY_STATUS_REG; int val = old;
while (old == val)
    val = KEY_STATUS_REG;
... // status has changed
            
```
- This technique is called **busy-waiting**, because it causes the program to "wait" although the CPU is still busy
- Note that the program has no control over when to exit the loop - this is in the hands of the **external world**, which has its own means of affecting KEY\_STATUS\_REG
- [Replace KEY\_STATUS\_REG with an ordinary variable, and the program is stuck forever!]

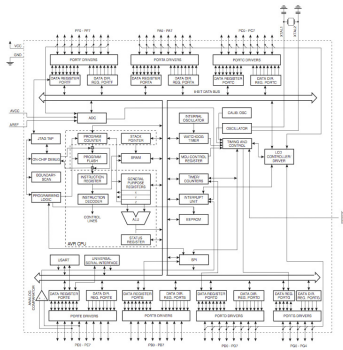
## Busy-waiting

- Busy-waiting as a synchronization technique has a lot of drawbacks, we will discuss them in depth in subsequent lectures
- However, busy-waiting has one striking benefit: it allows the programmer to think of I/O in the familiar terms of reading and writing program variables
 

```

int getchar() {
    while (KEY_STATUS_REG & PRESSED);
    while (!(KEY_STATUS_REG & PRESSED));
    return KEY_VALUE_REG;
}
            
```
- We will explore busy-waiting extensively in lab 1, so that we may safely drop it in later work!

## The ATmega169 microcontroller



## Some important ports

- #include <avr/io.h>
- LCDCR, LCDCRB, LCDCRA, LCDFR, LCDDRn
- TCCR1B, TCNT1
- DDRB, PORTB, PINB
- CLKPR
- See online manuals ATmega169.pdf and AVR065.pdf for detailed information

## Atmel AVR

- Much simpler than in general case
  - no need for explicit pointers and volatile
    - most of the time
- Writing to a port:
  - `<portname>=<value>`
- Reading from a port:
  - `<variable>=<portname>`
- Example:
  - `LCDCCR=controlBits;`

## Next lecture

- concurrency and mutual exclusion