Microcontroller

Lecture 1: Introduction

HOCHSCHULE RHEIN-WAAL
Rhine-Waal University of Applied Sciences

Content of the course

According to our module description, content is defined as following:

- *General about Microcontrollers*
- *Data representation in bits and bytes*
- *Princeton and Harvard architecture*
- *CPU components*
- *Instruction coding and addressing*
- *Data storage*
- *Input and output systems*
- *Development tools*

Integral part of this course are **laboratory assignments**, without which you can't take exam. In laboratory assignments we will program in C and gradually build up experience to accomplish simple control tasks (light LEDs, emit sound, use buttons to get input from the user and many other tasks).

Course Overview (1)

- From the curriculum Electrical Engineering

3. Semester														
EL 3 2014	Interkulturelles Management und Kreativität	4	2			2			x		5			4
EL 3 2306	Mikrocontroller	4	2				2		x	x	5			4
EL 3 2307	Felder und Wellen	4	2			2				x	5			4
EL 3 2308	Signalübertragung	4	2			1	1			x	5			4
EL 3 2309	Objektorientierte Programmierung	4	2				2			x	5			4
EL 3 2901	Antriebe und Leistungselektronik	4	2			2				x	5			4

- From the curriculum Mechatronic Systems Engineering

3. Semester														
SE 3 2010	Dynamik	4	2			2				x	5			4
SE 3 2108	Werkstoffe und Werkstoffprüfung	4	2			1	1			x	5			4
SE 3 2306	Mikrocontroller	4	2				2		x	x	5			4
SE 3 2705	Kronstruktionstechnik	4	2			2				x	5			4
SE 3 2708	Thermodynamik	4	2			1	1			x	5			4
SE 3 2901	Antriebe und Leistungselektronik	4	2			2				x	5			4

- Mandatory 8 laboratory experiments = Attestation (T)

  - successful completion of 8 mandatory laboratory experiments (otherwise failed)
  - once finished the labs (e.g. previous years, but failed the written exam) you may directly participate in the written exam

- 10 Bonus points (of 100)

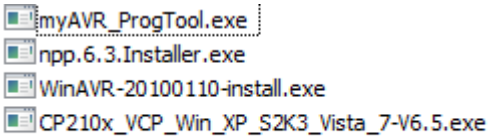
  - successful completion of all labs **with optional parts until 14.12.2018**

- Written module examination (120 min) = Graded (P)

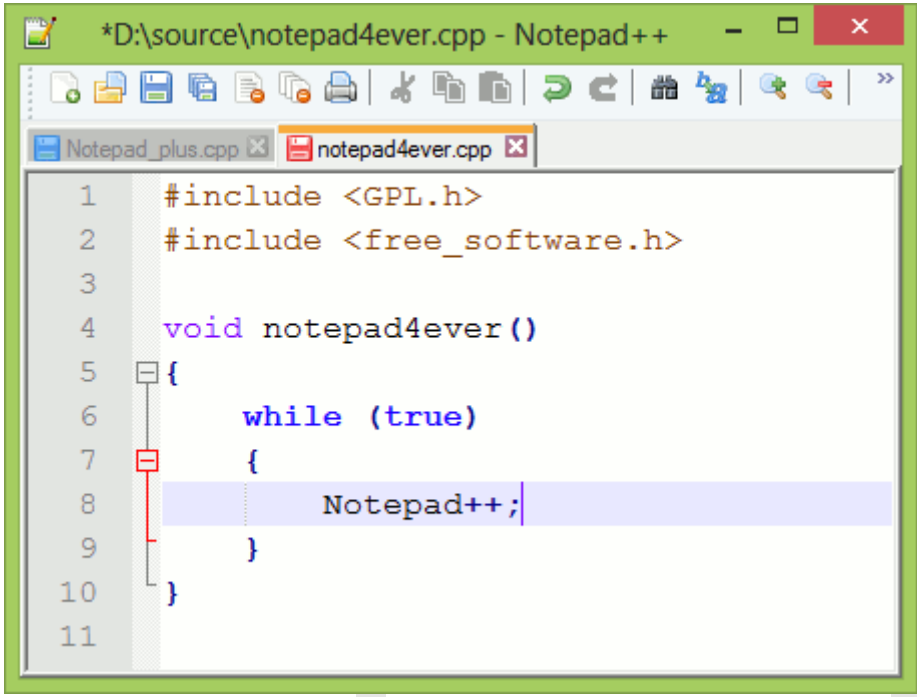
Course Overview (2)

Tools used in this course (actual version available on moodle):

- Software is available – WindowsAvrTools.zip
  - Programming tool myAVR-ProgTool.exe



  - Notepad++ Text Editor
  - WinAVR Toolchain
  - *Optional USB driver CP210x-VCP…



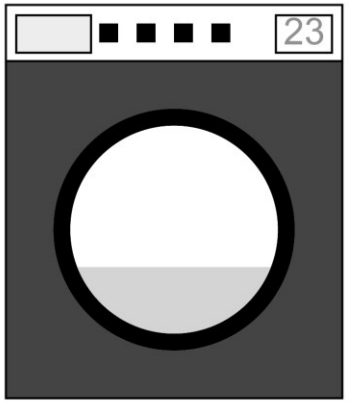
Overview – Microcontrollers

- Take a general view of functions provided by a microcontroller
- Examine memory, which is central to any computer system:
  - _ how it is arranged
  - _ how we communicate with memory
  - _ types of memory (RAM and ROM)
  - _ different architectures
- Operating system in a typical small microcontroller
  - _ there isn't one!
- Programming a microcontroller
  - _ different languages and approaches
  - _ how we communicate with memory

Product with a microcontroller: washing machine

What does the microcontroller need to handle?

- Inputs:
  - _ buttons on control panel (on/off — digital)
  - _ water level (digital?)
  - _ water temperature (continuous — analog)
- Outputs:
  - _ display on control panel
  - _ heater, water valves,… (on/off — digital)
  - _ motor (may appear to be analog, but usually digital using ‘pulse width modulation’)
- Also needs:
  - _ timer to control washing programme
  - _ memory for program and variables (state of washing programme), clock generator, logic to start machine up correctly…

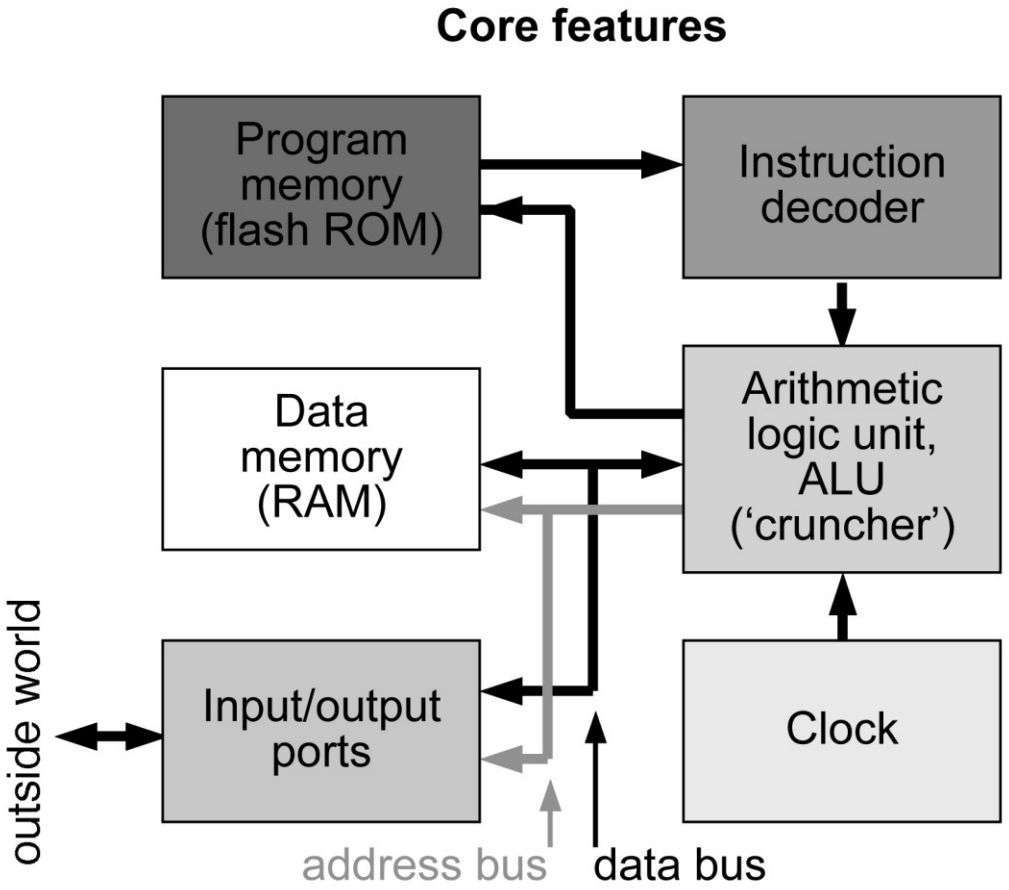


What is inside a microcontroller? (1)

What do we need inside a microcontroller?

- Arithmetic logic unit (ALU), which is/includes control unit
  - _ perform arithmetic and other manipulations on data
  - _ must be non-volatile: retain program even while power is off
  - _ read-only memory (ROM)
- Memory for data
  - _ may be volatile: doesn't matter if contents are lost while power is off
  - _ random-access memory (RAM)
- Communication with outside world
  - _ input and output 'ports'
  - _ may need to handle both digital and analog signals
- These need to be connected so that they operate together

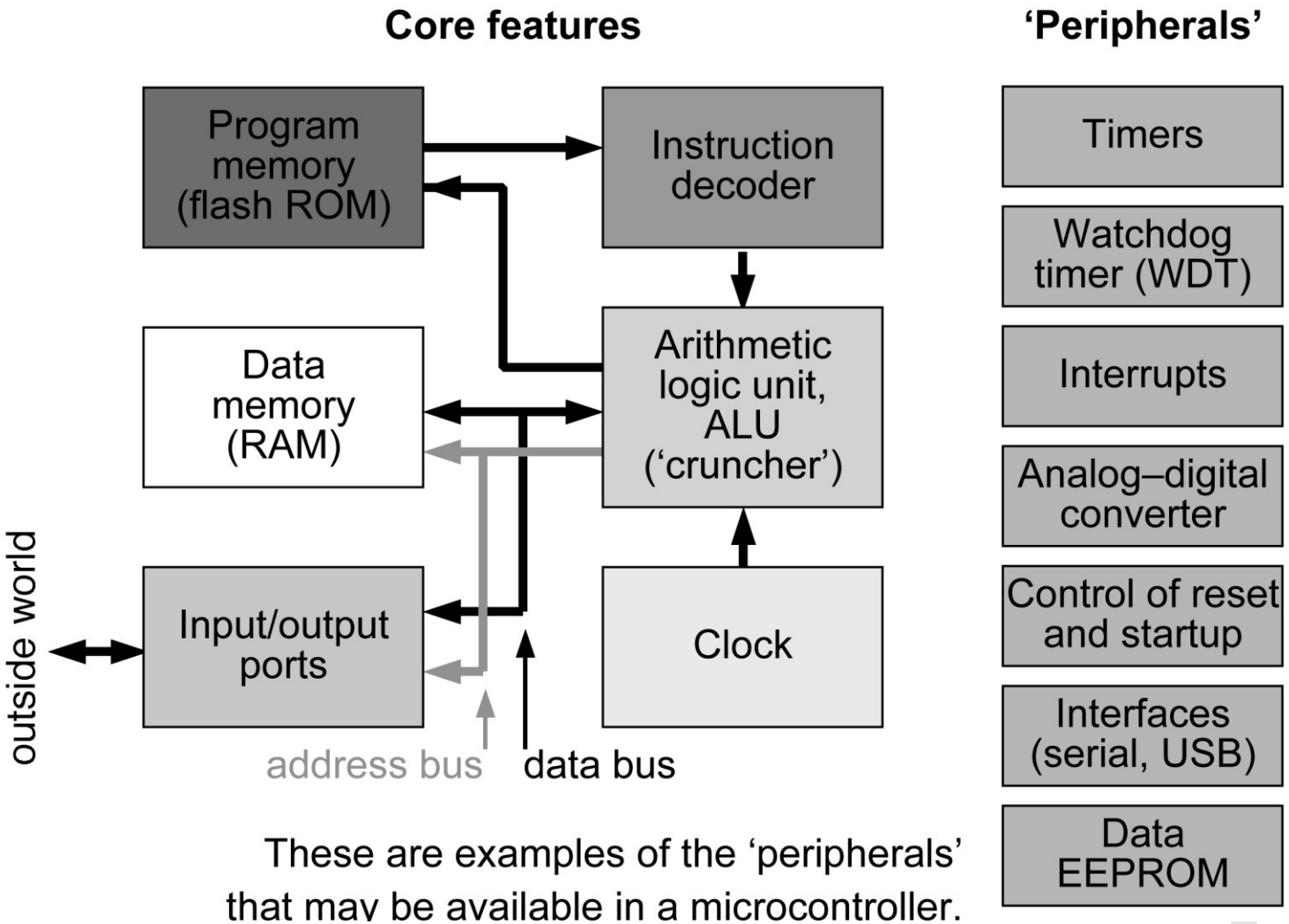
What is inside a microcontroller? (2)



The components are joined together by **buses**.
They are shown as single lines but are actually sets of 8 or more wires. Details later.

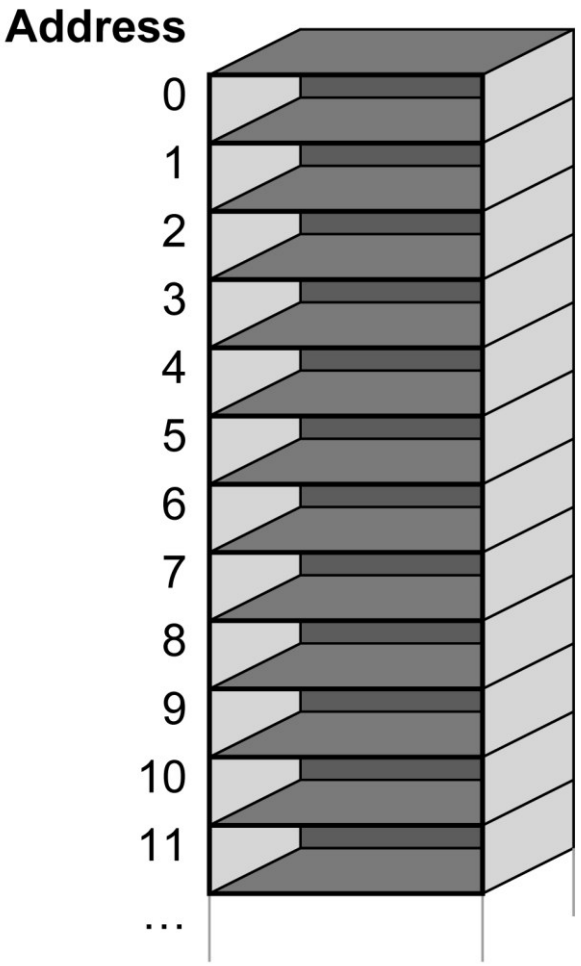
The **clock** keeps all parts synchronized.

What is inside a microcontroller? (3)



These are examples of the 'peripherals' that may be available in a microcontroller.

Memory — the heart of a computer



Memory in a computer is just like a (tall) stack of pigeonholes. There may be several stacks for different types of memory.

Each pigeonhole is identified by ist **address**, which is a serial number starting from 0

The address is said to **point to** a memory location (register).

When we communicate with memory, we need to handle both the data and the address.

Communication with memory

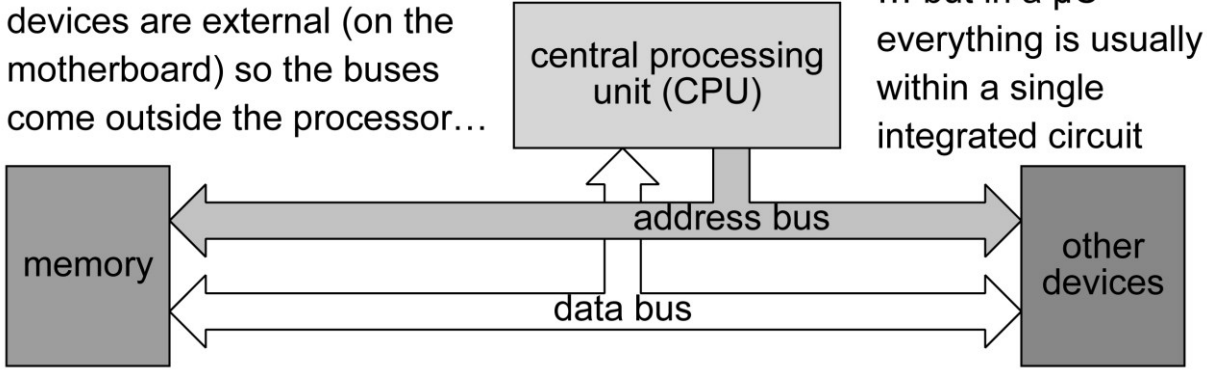
- Data is transferred between memory and the rest of the system using **buses**. These are shared sets of wires that join the components, something like a multi-lane highway. Several parallel sets of wires are needed:

  _ **address bus**, carries address (serial number) of pigeonhole

  _ **data bus**, carries byte either from the memory (read operation) or to the memory (write operation)

  _ **control lines** are also needed to synchronize timing, select read/write, ensure that only one device tries to use the bus at once, … ignore!

In a PC the memory and devices are external (on the motherboard) so the buses come outside the processor…



… but in a µC everything is usually within a single integrated circuit

Types of memory (1)

- Microcontrollers have two types of memory:
  - **volatile** — contents lost when power is removed, commonly known as **RAM** — random-access memory. Used for variables (not many).
  - **non-volatile** — contents retained even when power is removed, commonly known as **ROM** — read-only memory. Used for program.
- Unfortunately the common names are obsolete and misleading:
  - access to both RAM and ROM is equally random
  - most modern ROM is of a type called **flash**, which can be written and erased (although the process is much slower than with RAM)
- ROM used to live up to its name — it was 'written' when the chip was fabricated and could never be changed again.

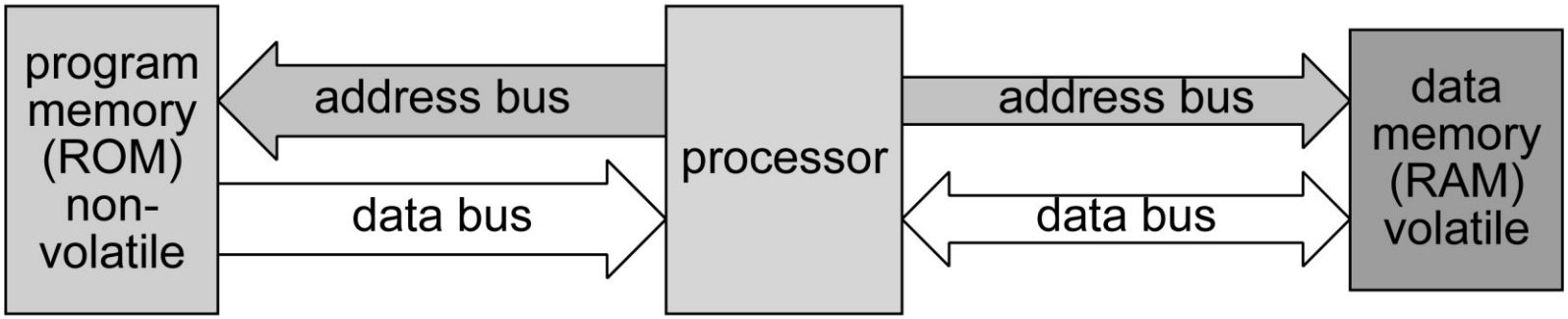
Types of memory (2)

- Flash memory (or EEPROM — electrically erasable, programmable ROM) can be erased and rewritten electrically — much more convenient, and enables upgrades to be distributed to the end user.
- Almost all memory in a PC is RAM. Each program must be read into RAM from non-volatile memory (usually disk) whenever it is needed. Similarly the operating system must be loaded into RAM whenever the system boots.
- In contrast, microcontrollers execute only one program, which can thereforebe stored in ROM and is available instantly.

Harvard and Princeton architectures

- The two types of memory that we have just seen,
  - _ **non-volatile** (ROM for program)
  - _ **volatile** (RAM for variables)
- can be treated in two general ways:
  - _ two completely separate memory systems: each has its own data and address bus — **Harvard** architecture, used in most microcontrollers.
  - _ single memory system — **Princeton** architecture, (almost) universal in general-purpose computers, but employed in some microcontrollers.
- Look at both briefly, as both are used in microcontrollers.

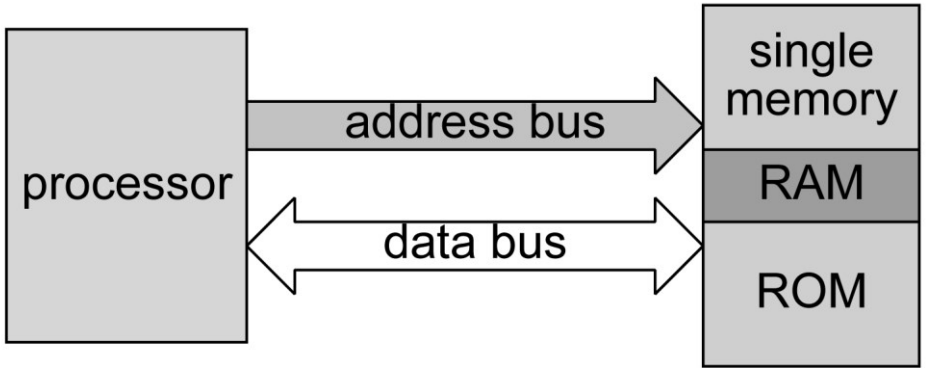
Harvard architecture



- More efficient use of memory:
  _ program and data can be accessed simultaneously
  _width of address and data buses can be optimized for each memory Most microcontrollers use this architecture, notably the **PIC** families from
- **Arizona Microchip**:
  _we shall explore the **PIC 16F676**
  _also has **reduced instruction set (RISC)** — see later

Princeton architecture

in Germany also known as “von Neumann model” (or architecture)



- More versatile architecture, almost universal in general-purpose computers:

  _several bus cycles are needed to get a complete instruction, including the data needed, so intrinsically slower

  _like a stack of different colour pigeonholes: some locations are ROM and some are RAM. Needs care! Study the memory map carefully.

- **Motorola** microcontrollers use this architecture:

  _the **Nitron (68HC908QT/Y)** range is very similar to our PIC

  _they use a **complex instruction set (CISC)** — see later

Operating system — desktop computer

- When you buy a desktop computer, it comes with an ‘operating system’ (linux, MacOS, Windows…).
- The original purpose of many operating systems was to keep track of files on disks (DOS = disk operating system).
- Modern operating systems provide a vast range of services to the applications that run on them.
- Suppose that you press the key ‘a’ on a keyboard. Then (in brief!):

  _the operating system reads a byte from the keyboard (itself an embedded system!)

  _it informs the application that an ‘a’ has been pressed

  _the application requests than an ‘a’ be drawn on the screen in a particular font at a given location

  _the operating system retrieves the pattern of dots (pixels) needed, and draws them on the screen to form the character

Operating system — small microcontroller

You are on your own!

A small microcontroller usually has no operating system at all.

- The development software may help you, but basically you must do everything yourself. For example, you have to:

  _configure the microcontroller when it starts up (which pins are inputs and outputs, for example)

  _make sure that your program starts at the right place

  _keep track of your variables and ensure that you don't run out of space

  _react to inputs when they occur

  _keep track of different tasks if you have more than one running (as is usually the case)

The development of software is a heavy, one-off cost.

Programming a microcontroller

There are several ways in which the program for a microcontroller can be written, depending on the size of the project and how critical the performance may be.

(1) **Assembly language**. This is no more than the instructions understood by the µC but written in a form that is easier for humans to understand. I shall give some examples because it shows precisely what the µC does. This is useful for small projects but rapidly gets incomprehensible.

(2) **C**. This is the language of choice for most projects on small µCs. It is close enough to the hardware to be efficient, but is much easier to understand than assembly language.

(3) **Java/C++**. These are 'object-oriented' languages and are increasingly used for major projects. Many mobile phones contain interpreters for Java, for example (so that they can run games!). These need much larger µCs than I shall describe.

Summary – MCU

Looked at major systems to be found inside a microcontroller.

- **ALU, memory, communication with outside world**
- **may be many 'peripherals' on-chip to simplify overall system**

Memory can be visualized as a stack of pigeonholes with addresse given by serial numbers

- **non-volatile ROM for program and volatile RAM for data separate (Harvard) or common (Princeton) address space most µCs, including the PIC use Harvard model although Motorola uses the Princeton model**

There is usually no operating system!

- **you must write everything yourself**

It is important, but particularly difficult, to develop a good interface for users — a major discipline within computer science.

Inputs and outputs for a microcontroller

- Power supply

  -very brief — will explore this in more detail later
- Digital inputs

  -how to connect switches and pushbuttons

  -need for pullup (or pulldown) resistors

  -switch bounce
- Digital outputs

  -how to connect light-emitting diodes (LEDs) value of resistance in series

  -what can the PIC drive directly?

Power supply

This is rather easy with modern µCs. They don't care too much about the precise voltage, nor do they draw much current, so batteries are ideal.

Most current designs work at 3 V, which you can easily get from two cells in series (2 AA for example). We'll look more at power supplies later — how you can get the system to work from a *single* AA cell, for example.

A couple of points:

- *V*SS is negative (ground), *V*DD is positive (supply)
- It is good practice to connect a **decoupling capacitor** between the pins for VSS and VDD as close to the µC as possible. The value is usually about 10^{-7} F = 0.1 µF = 100 nF. This is because all digital circuits produce electrical noise as they switch. The capacitor 'shorts this out' and prevents it affecting other components.

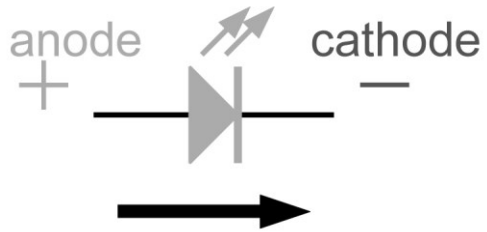
Look for the capacitor on the Stamp boards!

Output: Light-emitting diodes (LEDs)

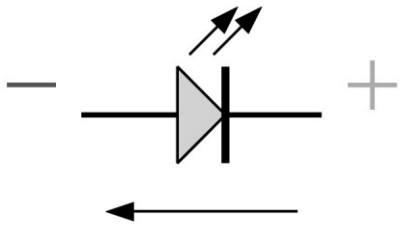
Light-emitting diodes (LEDs) are the simplest type of output that can be driven by a microcontroller. From their name, they are diodes (p–n junctions in a semiconductor) that emit light.

The important feature electrically is that they are **diodes**. This means that they pass current only in one direction. Almost no current flows if they are connected the other way around.

The symbol for a diode is basically an arrow: current flows in the direction of the arrow. (Remember that current flows -conventionally- from + to –.)



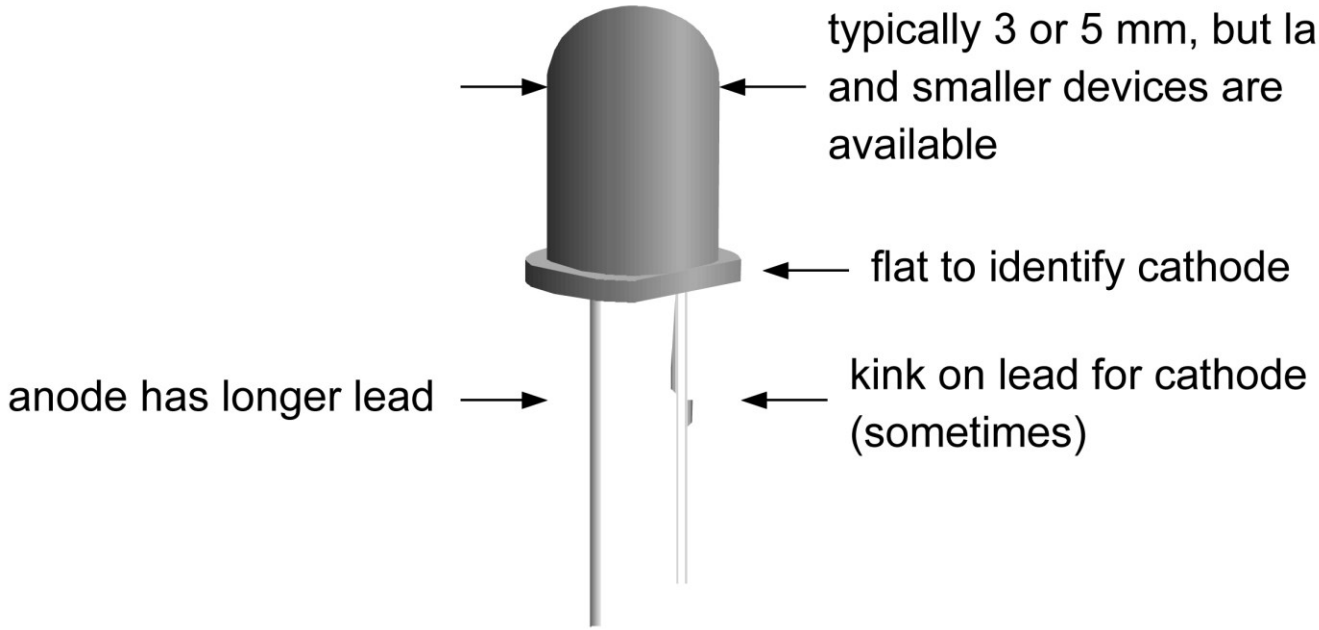
large current flows, emits light



small current flows, dark

Sketch of a LED

A wide variety of LEDs is available. For many years they were red, yellow or green, but blue and white LEDs are now available. They are increasingly efficient too (need less current). The design below is the most common but there are many others.

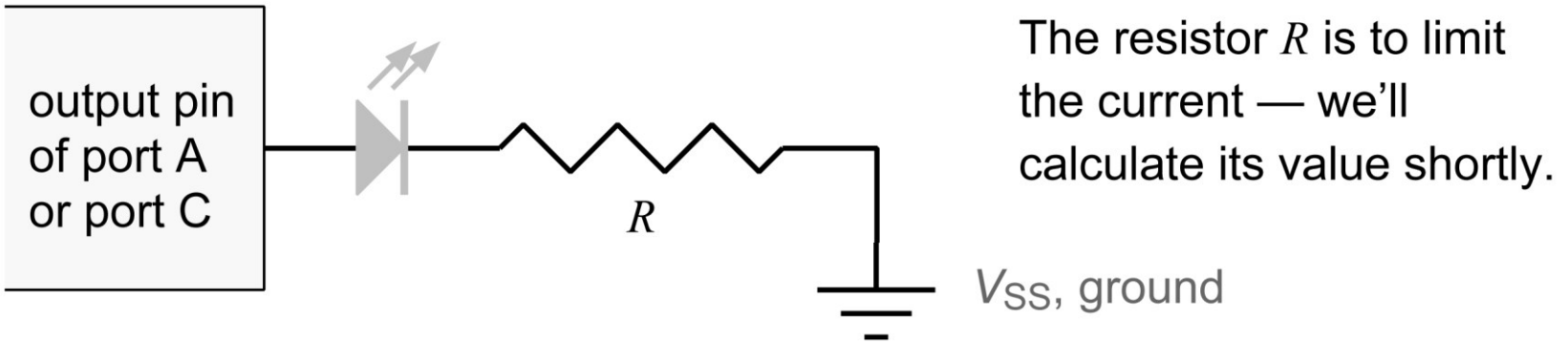


Basic LEDs cost about 5 Cent each in bulk. You can get bicolour, tricolour, different shapes, flashing…

How to connect LEDs (1)

Remember that LEDs are **diodes**, so we must connect them the correct way around! There are two ways in which this can be done.

1. Suppose that we want the LED to illuminate when the logical value of the output is high (1) — **active high**.
   Thus the LED should light when the pin is at *V*DD (supply, positive, logic 1) but not when the pin is at *V*SS (ground, negative, logic 0).
   We therefore connect the LED between the pin and *V*SS.
   Conventional current flows from the pin to *V*SS with this connection.



The resistor R is to limit the current — we'll calculate its value shortly.

How to connect LEDs (2)

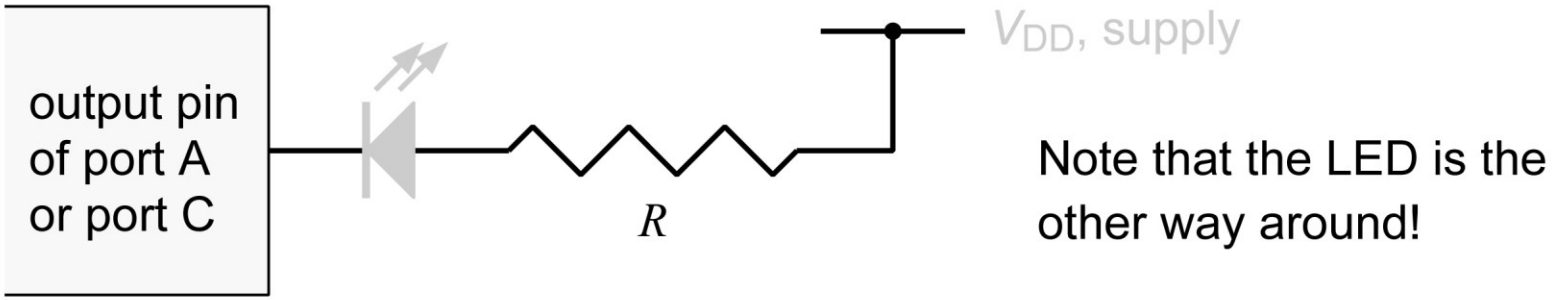
It might seem obvious that you would always want to connect an LED like this but there is a second, complementary way in which this can be done.

2. Suppose that we want the LED to illuminate when the logical value of the output is low (0) — **active low**.

Thus the LED should light when the pin is at *V*SS (ground, negative, logic 0) but not when the pin is at *V*DD (supply, positive, logic 1).

We therefore connect the LED between the pin and *V*DD.

Conventional current flows from *V*DD to the pin with this connection.



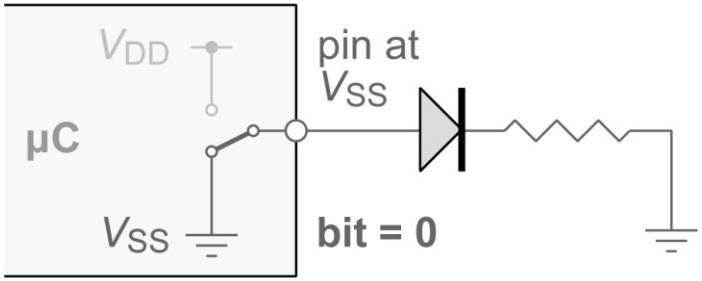
This connection was more popular in the past.

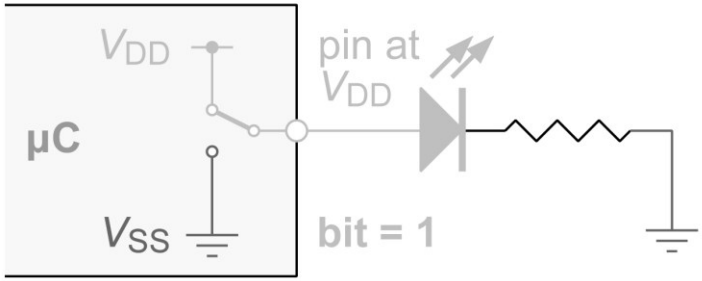
Operation of ‘active high’ LED in detail

Remember how the output works for each bit of port A or port C:

logical 0 causes the µC to drive the pin low, to V_{SS}, ground, negative

logical 1 causes the µC to drive the pin high, to V_{DD}, supply, positive





Both ends of the LED are at V_{SS} so no current flows

The anode of the LED is at V_{DD} and the cathode is at V_{SS}.

A forward current can flow (in the direction of the arrow) so the LED illuminates

(What would happen if the LED were connected the other way around?)

Resistor in series with LEDs

A resistor is usually needed to limit the current through the LED.
A small modern LED (red, yellow or green) needs about 3 mA for reasonable brightness and drops about 1.8–2.0 V in forward bias.
(Blue/white LEDs need 3.6–4.0 V, which can be a problem.)
Suppose that the supply is +3 V. What value of resistor is needed?

Next slides will help you with Practicals (incl. two theoretical annexes)

Output: Using LCD

LCD can be very useful for debugging and reading any sensor values. Luckily, there are libraries you can use, which makes using LCD easy.

Place files lcd.c and lcd.h in your working folder.
In the code: add

#include "lcd.h“

It is easiest to print strings on LCD. In C language it means an array of characters.

IMPORTANT: YOU HAVE TO DISCONNECT THE LCD IF YOU ARE USING INTERRUPTS (NEXT LABORATORY EXPERIMENT!!!)

Using LCD 2

A good way to form a string you need is `sprintf` command.
It has usual `printf` syntax, however, the output goes to a char array.
Example:

```
char str[16];
sprintf(str, "v: %d; r: %d", OCR1A, defVal);
```

Only what is in the “ “ will be put in the string. However, %d is a placeholder, so two of them will be replaced by the values of variables OCR1A, defVal. The result will be put in str.

There are more placeholders for other variables, including strings, so you can for example insert a name of a person in a predefined text.

Using LCD 3

So you have the string you want to print. What's next?

```
lcd-clear();        //remove all previous info from LCD
lcd-string(str);    //print the string "str"
```

And if you want to use both lines of the LCD display:

lcd-setcursor(uint8-t x, uint8-t y); //experiment with it if you are interested.

Don't forget that you will need four lines in this case, e.g.

```
lcd-setcursor(x1,y1);
lcd-string(str1);
lcd-setcursor(x2,y2);
lcd-string(str2);
```

Input: ADC initialization

Now we will learn ways to get information into our microcontroller. **ADC** (Analog to Digital Converter) is the part of uC used to represent analog voltage value in terms of digital value (there will be some proportionality between digital value and voltage, eg. 1 unit digital might correspond to 1 mV).

ADC has many properties, which can be configured, so starting it will take a bit more then one line of code.

We will need:

1) Set **Free-running conversion mode** (convert all the time, not wait for a signal to make new **ADC** conversion)

2) Choose on **what pin to read**

3) Define **prescaler** (how often the **ADC** conversion must be performed, in some cases you can save a bit of power and more)

ADC initialization 2

ADMUX |= ??????????; //choose which pin to use as input (see ATmega88PA documentation, page 248)

```
ADMUX |= (1 << REFS0);    //our board, AREF = Vcc
ADMUX &= ~(1 << REFS1); //our board, AREF = Vcc

ATmega8
ADCSRA |= (1 << ADEN);  //ADC enabled
ADCSRA |= (1 << ADSC); //Do first conversion (in free running mode you don't
have to start it again

ADCSRA |= (1 << ADFR); //Free running mode on
ADCSRA |= (1 <<ADPS1) | ( 1 << ADPS0);
ADCSRA &= ~(1<<ADPS2); // prescaler = 8 for ADC, meaning ADC clock is 8 times
slower when for the whole board
```

ADC initialization 3

ADMUX |= ??????????; //choose which pin to use as input
(see ATMega8 documentation, page 199
or ATmega88PA documentation, page 248)

```
ADMUX |= (1 << REFS0);    //our board, AREF = Vcc
ADMUX &= ~(1 << REFS1); //our board, AREF = Vcc
```

ATmega88PA

```
// ADCSRA
ADCSRA = (1 << ADPS2) | (1 << ADPS1); // Select ADC Prescaler

// Start conversion, Free running, Enable ADC
ADCSRA |= (1 << ADSC) | (1 << ADATE) | (1 << ADEN);
```

ADC precision

Imagine your documentation says you have 12-bit ADC.
Your input changes in the range from 0 to 5 Volts.

What does this mean?

It means that ADC will generate 12-bit variable (from 0 to 2^12-1=4095), and one unit of this variable will correspond to 5V/4096,
or roughly 1,22 mV.

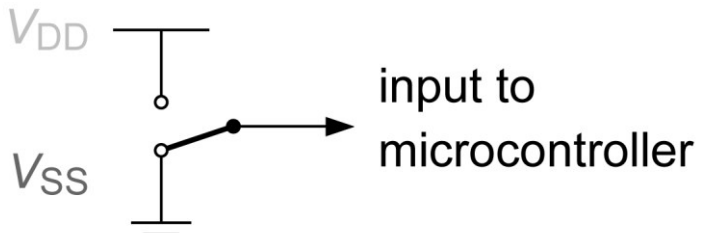
Now if you want for example to split your possible values to 4 value regions, LOW, MID-LOW, MID-HIGH, HIGH you would have to divide all your possible 4096 values of the output into four parts..

In the laboratory experiment you have 10 bit precision and need to divide it in 3 value regions.

Input: How do we connect a switch? (1)

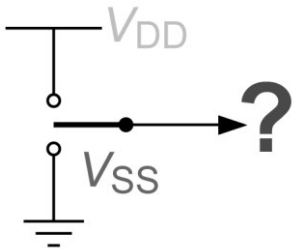
Suppose that we want to connect a simple on–off switch or pushbutton as an input to a µC. How should this be done?
The input should be either:

- VDD (supply, positive) for logic 1
- *V*SS (ground, negative) for logic 0



This may be the obvious approach:
However, this is not used, for several reasons

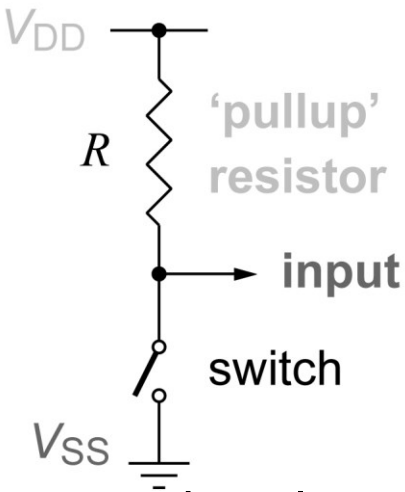
- it needs an expensive switch (3 terminals instead of 2!)
- it needs 2 or 3 wires
- **what happens while the switch is being operated?**

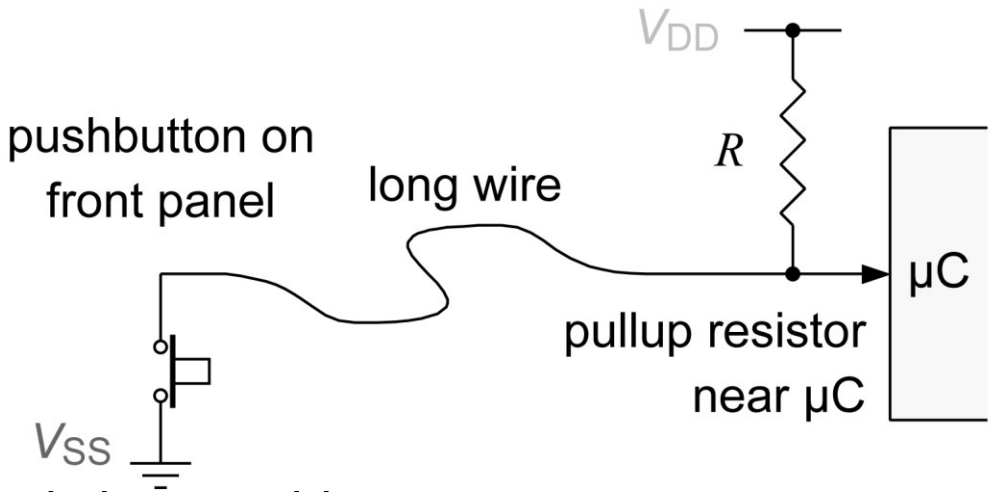


Digital circuits must never be left with their inputs 'floating' (not connected)! This leads to unpredictable behaviour and possibly severe problems, very difficult to isolate.

Input: How do we connect a switch? (2)

Instead, this is the circuit that is usually used. Traditionally the **pullup resistor** is around R » 10 kΩ.

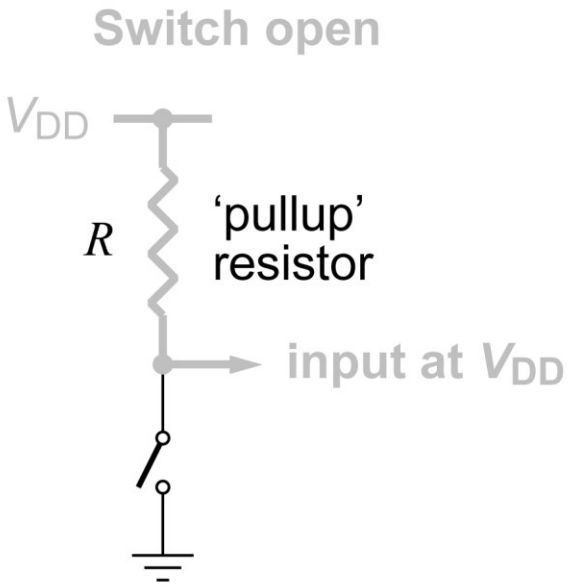


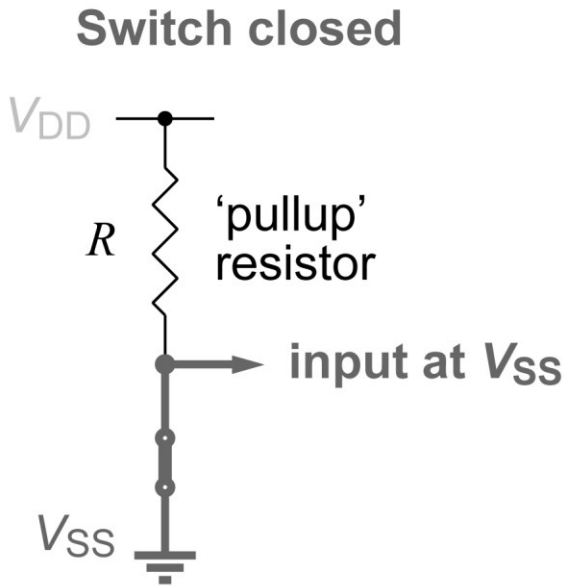


Ad

- needs only a simple switch or pushbutton
- only one wire to the switch is usually needed, because a ground connection can usually be found nearby
- the input is always connected to something

How does this work?





The input is connected to V_{DD}, logical 1, through the pullup resistor when the switch is open.

The input is connected to V_{SS}, logical 0, through the switch when the switch is closed.

A wasted current flows through the pullup resistor from V_{DD} to ground.

Input is 'active low'
– goes from logic 1 to 0 when the switch is closed

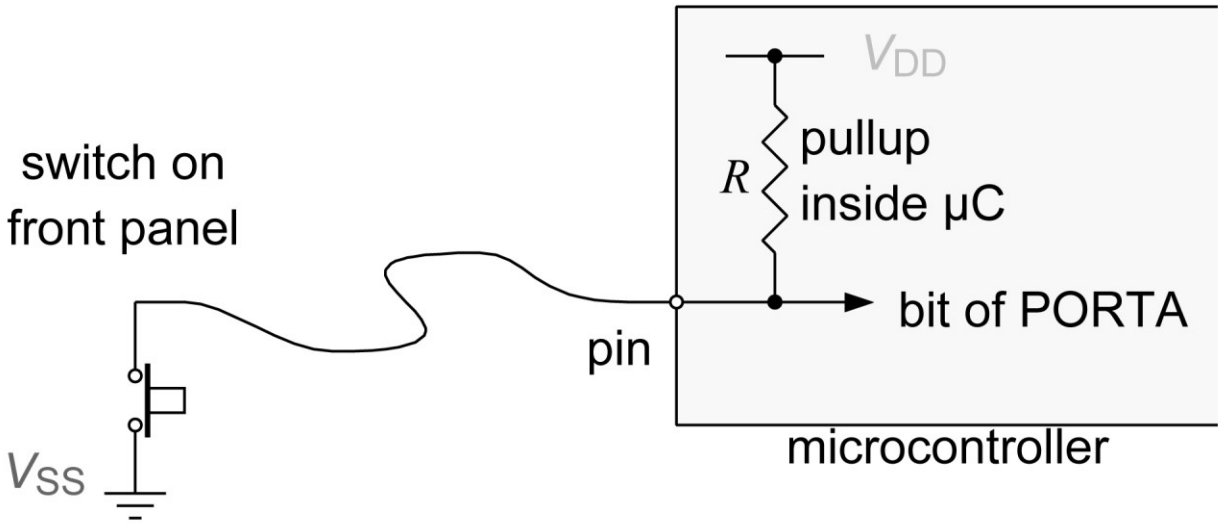
How much current is wasted?

A typical supply voltage is 3 V and pull-up resistors are often 10 kΩ. How much current is wasted through the pull-up when the button is pressed, and how does it compare with the operating current of the microcontroller?

It is equally possible to make an ‘active high’ input with a pull-down resistor, but the pull-up is standard. Partly this is because it is usually easier to find a connection to ground near the switch.
E.g., **the myAVR boards have active high inputs**.

Internal pullup resistors

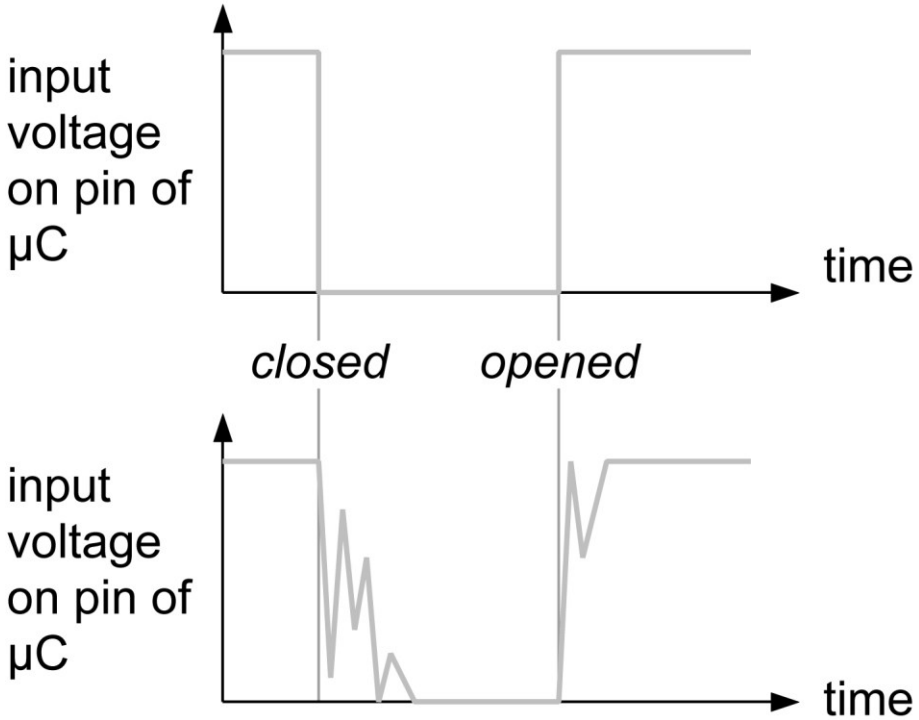
This configuration is so widely used for inputs that pull-up resistors are provided inside the microcontroller (but only for port A). This makes the circuit really simple.



They are called ‘weak pull-ups’ because their resistance is around 20 kΩ to save current.
The pull-ups can be turned on and off individually by setting or clearing bits in register WPUA (bank 1).

Switch bounce

We have almost finished with inputs but there is one unpleasant feature that you need to be aware of: **switch bounce**.
The contacts of a switch are mechanical. When the switch is operated they do not open or close cleanly but vibrate for a while. This causes the contacts to open and close several times, each time the switch is operated.



Ideal input signal

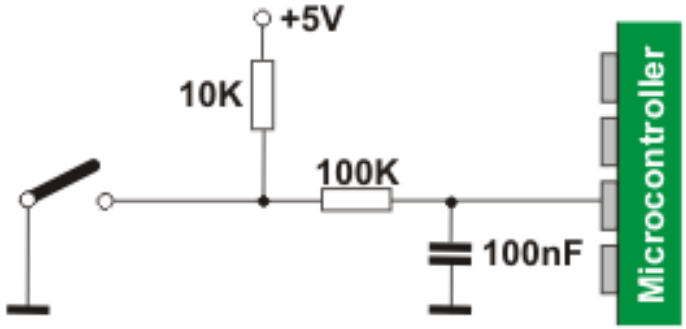
Input signal with 'bounce'.
This typically lasts for around 50 ms.
It can be eliminated by using extra components or (more usually) in software.

Debouncing

Problem: Buttons, as well as switches described on previous slide, are prone to signal spikes after the press. It can result in many readings per one actual click.

The possible hardware solution would be RC circuit:

Here the signal noise, which has high frequency, is filtered by capacitor.



However, it would be a lot easier if we can solve the problem in the software!

Debouncing

As said, the whole process is lasting up to about 50 ms (it depends on the quality of the used switch), so the simplest solution is to stop reading the signal for the 50 ms after the event.

What C command allows you to wait for 50 ms? When must we call this command?

Please write down your version of the code for this! (assume that button is connected to PB2, and per press you want to switch on and off the LED connected to PB3). If you do not know precise name for function that waits, write `wait(x ms);`

Debouncing

Example of the code:

```
if( ~PINB & 1<<PB2)
{
_delay_ms(50);
PORTB^= 1<<PB3
}
```

Conclusions (1)

Simple to drive LEDs
- can be connected to illuminate when output is 1 or when it is 0
- must remember that these are diodes and connect them the correct way around!
- usually need a resistor in series to limit the current

Need power electronics to drive heavy loads (heaters, motors,…)
- other courses like Power Electronics (= Prof. Schmetz)

Looked at connection of input switches and pushbuttons to a µC
- can be active low or active high
- need pull-up (usual) or pull-down (on PIC Stamp board) resistors so that input is well defined at all times

Conclusions (2)

We have learned about 2 ways to output information:

- **LEDs**
- **LCD**

Both are extremely valuable for debug (you need to know what exactly is happening inside the uC memory, therefore outputting your variables with LEDs and LCD is a great idea!)

Regarding input, we learned how to use 2 tools:

- **buttons/switches** and
- **Analog to Digital Converter**.

Many sensors are analog, meaning that knowing how to use **ADC** allows you to use great many measurement devices!

Appendix 1:
Arithmetic with Microcontrollers

Appendix is additional information to help you in the laboratories and promote understanding of microcontrollers.

- Representation of numbers is generally based on integers.
- Numbers are represented in binary notation.
- The number of bits used to represent data is limited. Thus, the range and resolution of the representation is limited.

Computer arithmetic

- The rules of arithmetic are the same in natural binary as they are in the more familiar base 10 system.

- Simplest operation is addition. Three rules (instead of 45 for decimal!):
  0+0 = 0
  0+1=1 / 1+0=1
  1+1=10 (0 carry 1)

- Example:

1100000	96
+0100101	+37
10000101	133

Signed integers

- Counting down, starting with 0111 (=7)
- When the bits reach <0000>, the next lower number is <1111> which is ‘one less than zero’, or -1 (like on a car’s odometer that is turned back from 00000 to 99999).
- The left most bit represents the sign (1=negative, 0=positive)
- This representation is called Two’s complement.

4bit binary	signed decimal
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

Two's complement

- The two's complement representation of signed integers has the valuable property that two such numbers can be added using the **same arithmetic rules** as for unsigned integers.

- This is important since we can use similar logical circuits for adding and subtracting numbers.

- Example:

0011	3
+1110	-2
1 0001	1

The fifths bit is the **carry-out** bit. If it's ignored we obtain the correct representation of the sum, +1.

Two's complement

- Changing the sign of a number is very simple in this representation:
  invert all bits
  add one to the result

Example:

+7	0111	
	1000	invert the bits
	1	add 1
	1001	-7

Two's complement: Theory

- Mathematically, we have defined that if A is an *n*-bit binary number, then –A is represented as 2*n*-A. Conveniently this is equivalent to inverting all the bits and adding 1!
- It can be shown that for the definition above the rules of subtraction and addition are valid.
- The same thing also works in decimal representation (base 10)!

Multiplication and Division

- Multiplication by the *n*th power of two is simply implemented by shifting the data left *n* places (<< denotes the shift left operator): 00101*(5)* << 01010*(10)* << 10100*(20)*

- Division by the *n*th power of two is implemented by shifting right *n* places (>> denotes the shift right operator): 1100*(12)* >> 0110 *(6)* >> 0011 *(3)*

Multiplication and Division

- Multiplication and division by non-powers of two can be implemented by a combination of shifting and adding

  For example:
  $(3x10) = (3x8)+(3x2) = (3x2^3)+(3x2^1)$

  00011 << 3 = 11000 *(24)*
  00011 << 1 = 00110 *(6)*
  11110 *(30)*

Overflow and Carries

- If an arithmetic operation results in a value which is outside the range which can be represented by the variable we are talking of overflow.

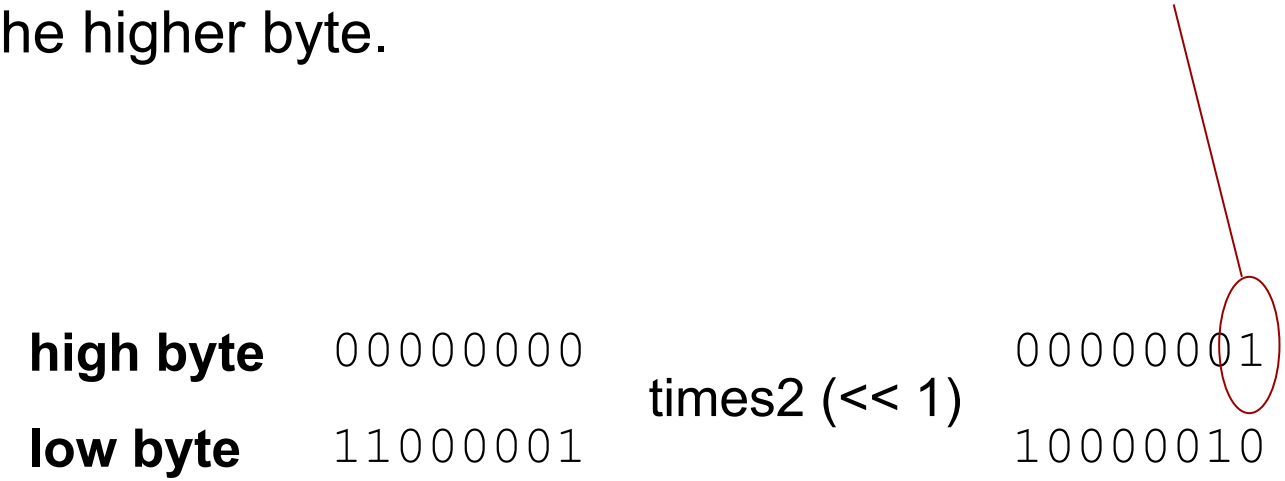
  Example:

```
   11100111   (231)
  +00011010   (26)
  ----------
  1,00000001  (257)
```

(1) in 8bit representation!

Overflow and Carries

- If a variable is represented by more than one byte (e.g., a *word*), overflow in the lower byte results in a **carry bit** being taken into the higher byte.



Scaling of Integers

- Although integer arithmetic can not represent fractions, decimal numbers can often be scaled to be represented by integers.

  Example: A thermometer should measure temperatures between -40 and +60 degrees. Since we are using an 8bit ADC, the measurements must fit into one byte. To represent this range internally, we map -40 to 0 and +87.5 degrees to 255:

degrees	-40	-39.5	-39	…	86.5	87	87.5
units	0	1	2	…	253	254	255

Fixed point numbers

- Fractions can be expressed using a fixed (binary) point notation: 000101.11 = $2^2+2^0+2^{-1}+2^{-2}$ = 5.75
- As the binary point is at a fixed location, the range and resolution are limited. For the example above, the range is (assuming unsigned numbers) 0…63.75 and the maximal resolution is 0.25 (2^{-2}).

Floating point numbers

- to cope with very large and very small numbers, floating point notation is more suitable. We may write

  11011100.0 as 1.1011100×2^7

  00011100.0 as 1.1100000×2^4

  0.00111010 as 1.1101000×2^{-3}

  Note that the first bit of the number is always 1 (normalised representation).

- Standard IEEE 754 provides a notation for floating point numbers:

$(-1)^S \times (1+F) \times 2^E$

where

S = sign

F = fraction

E = exponent

IEEE floating point formats

Description	S	E	F	Largest	Smallest
Single precision, 32bit	1bit	8bits	23bits	3.4×10^{38}	1.18×10^{-38}
Double precision, 64bits	1bit	11bits	52bits	1.8×10^{308}	2.23×10^{-308}
Double extended, 80bit	1bit	15bits	64bits	1.2×10^{4932}	

Most common format is single precision:

Sign S = 0 (positive), =1 (negative)

Exponent E=-126…127 (biased by -127)

Fraction $F=bit22 \times 2^{-1} + bit21 \times 2^{-2} + \ldots + bit1 \times 2^{-22} + bit0 \times 2^{-23}$

Floating point numbers

- Special numbers are:
  E=0 and F=0 is defined as zero
  E=Emax and F=0 is defined as infinity
  E=Emax and F≠0 is defined as Not a number (NaN)

Precision

- The main reason for using double precision (8 byte) numbers is not the increased range, but the improved precision.

- **Single Precision**

Fractional part originally 1+0
Next number $1+2^{-23}$
Precision is 2^{-23} in 1 i.e. 1 part in 2^{23},
i.e.≈ 1 part in 10^{7}

- **Double Precision**

Fractional part originally 1+0
Next number $1+2^{-52}$
Precision is 1 part in 2^{52}
i.e.≈ 1 part in 10^{15}

Appendix 2:
Interfacing with real world

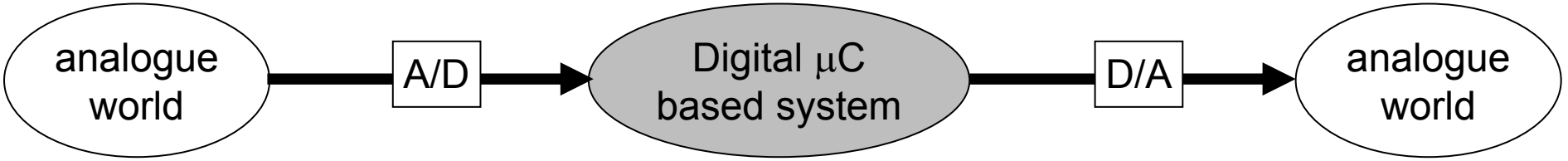
Appendix is additional information to help you in the laboratories and promote understanding of microcontrollers.

- Analog signal to digital signal (ADC)
  - Successive approximations technique

Interfacing with the Real World

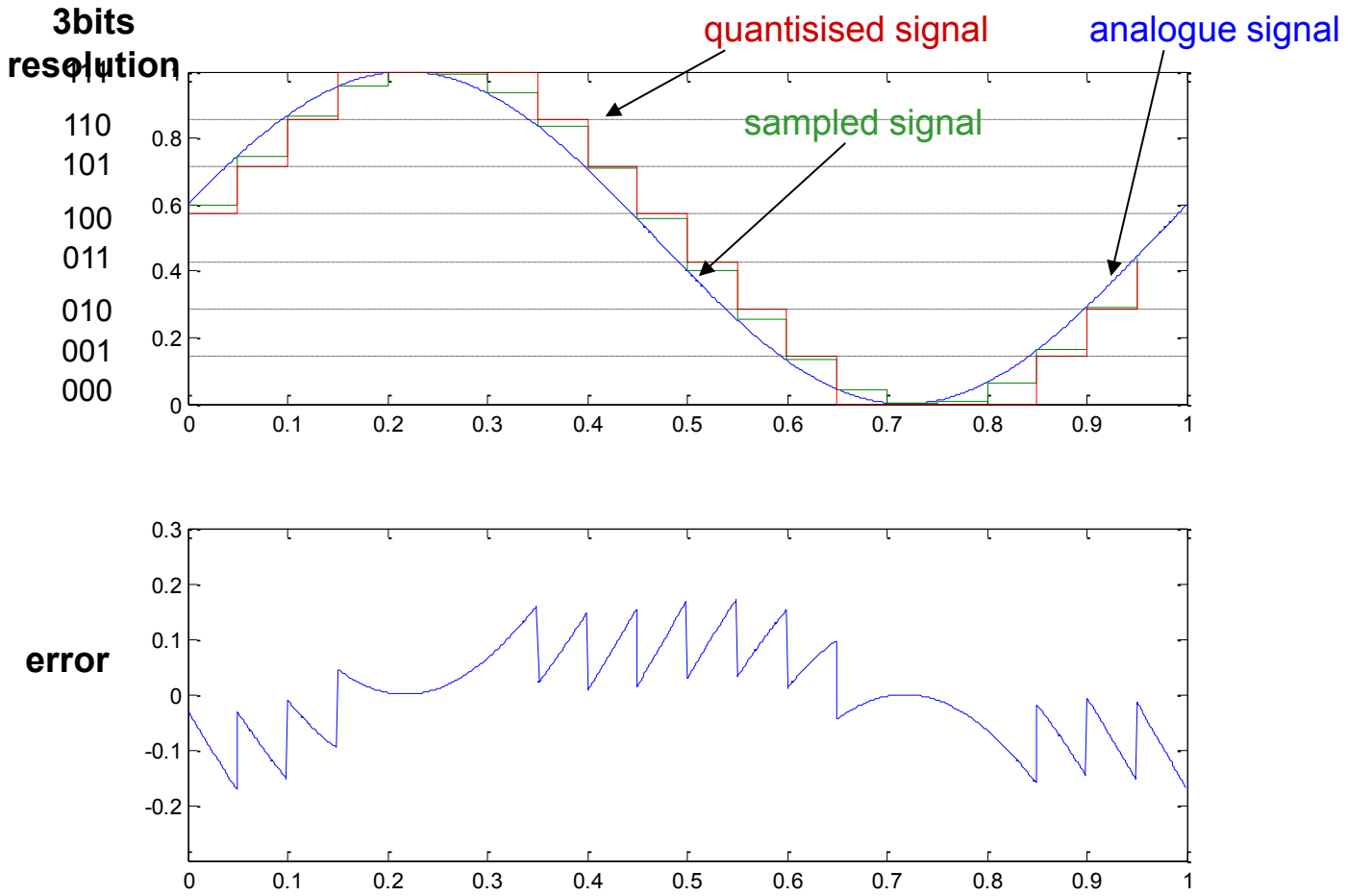
- Digital microcontrollers are in the business of monitoring and controlling the real environment – which is commonly **analogue** in nature.

- To interface with the environment, **analogue to digital conversion (ADC)** and **digital to analogue conversion (DAC)** is needed



From analogue to digital signal

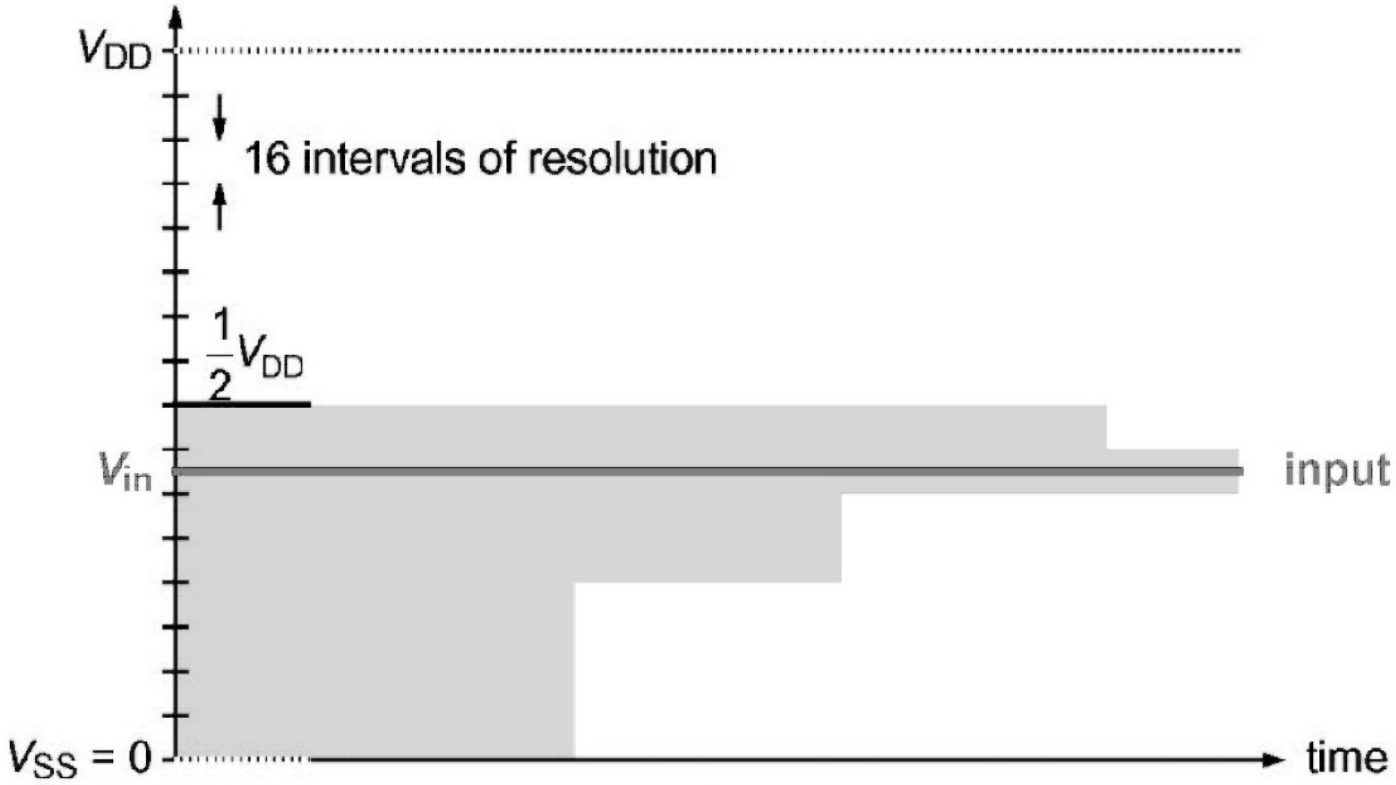
- Error mainly due to sampling and quantisation.



Analogue-digital conversion

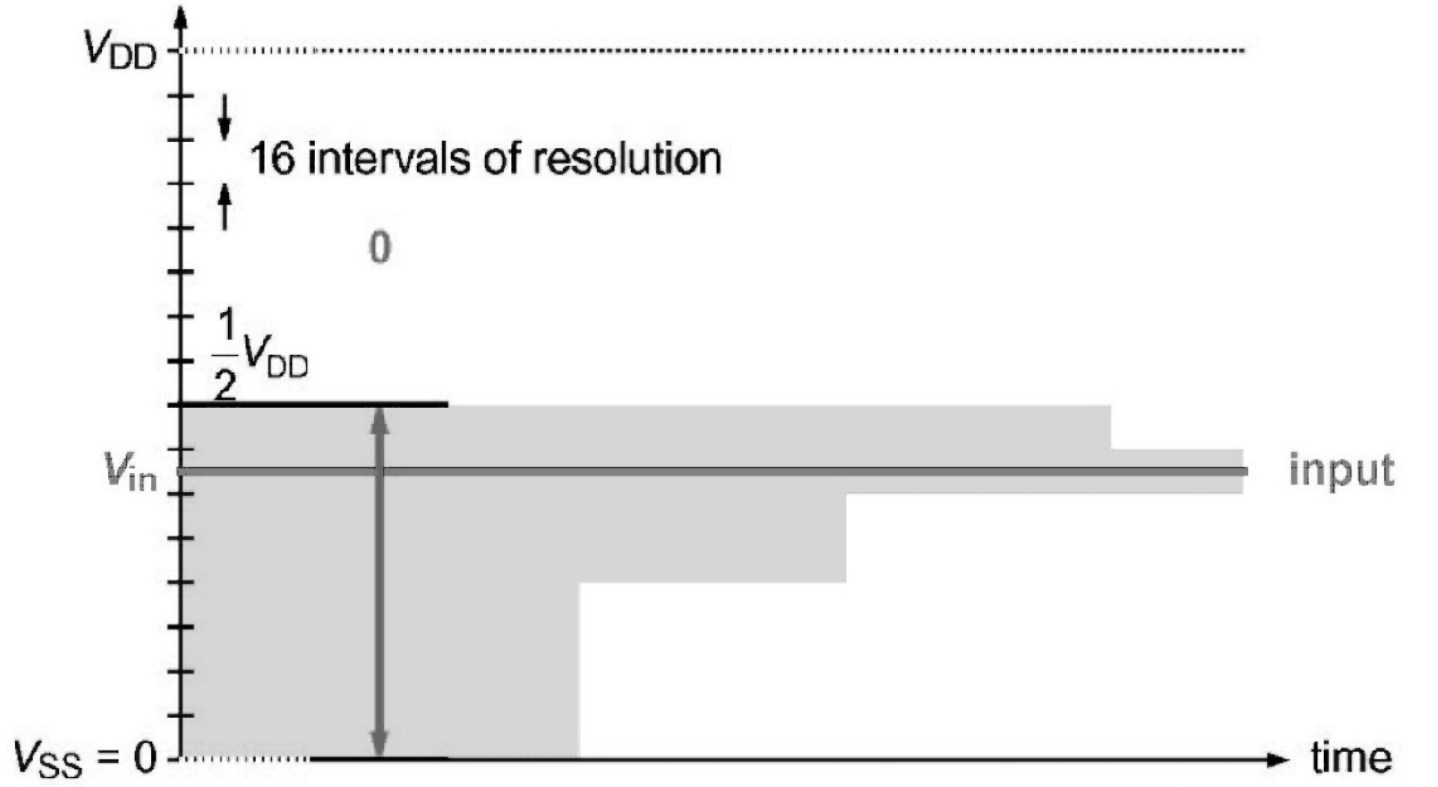
- although there are many different methods for ADC, by far the most common is the successive approximation technique:
  - In each step, halve the range within which the solution is known to lie.
  - Start by comparing V_{in} with $(1/2)V_{DD}$. This tells us whether the voltage is in the top or bottom half of the range — gives most significant bit (MSB) of converted value.
  - If it is in the bottom half (say), we next compare V_{in} with $(1/4)\ V_{DD}$. This tells us whether the voltage is in the top or bottom half of the subrange (i.e. first or second quarter) and gives the second most significant bit.
  - Carry on halving the range until enough bits have been generated.
  - Look at an example that generates four bits.

Successive approximation technique: 4 bit ADC



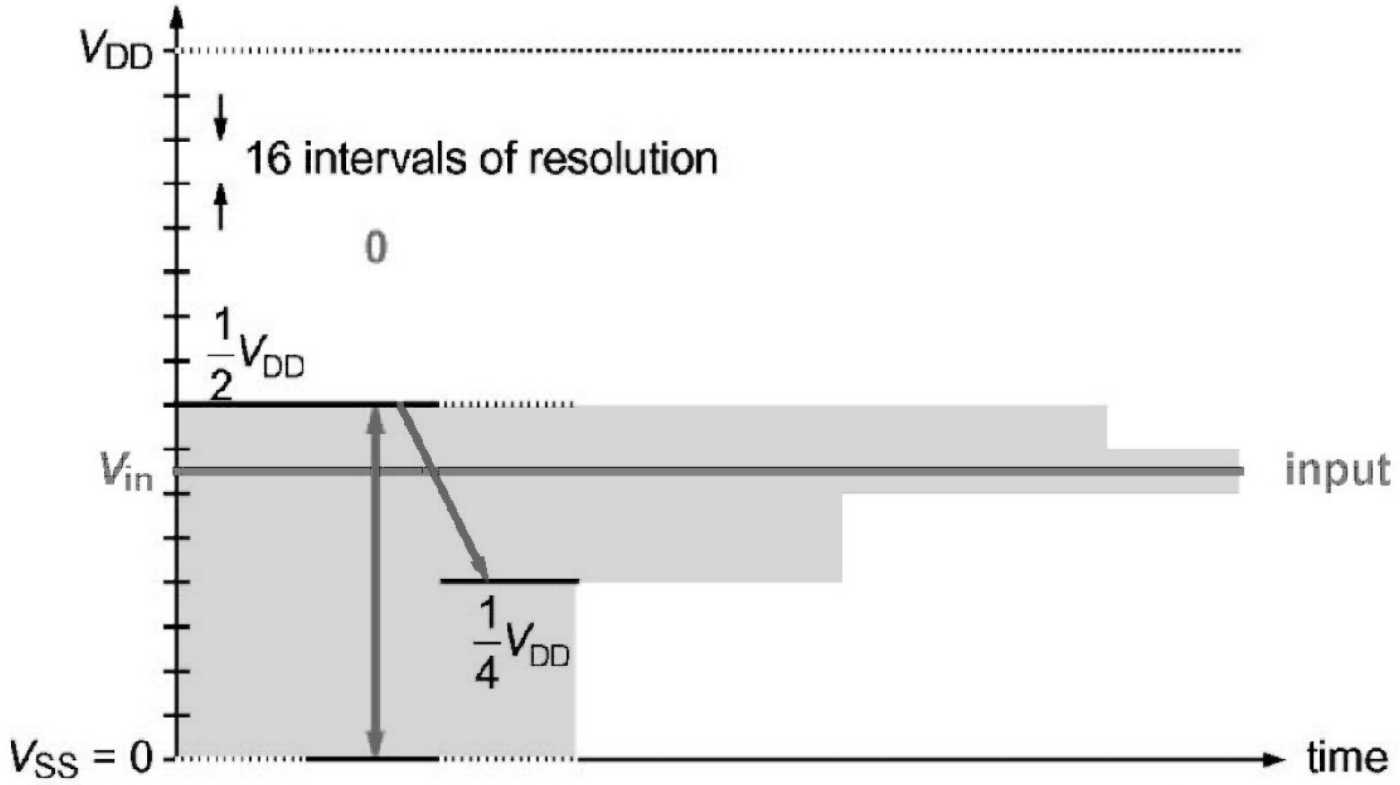
1. Full range: compare input with $(1/2)V_{DD}$.

Successive approximation technique: 4 bit ADC



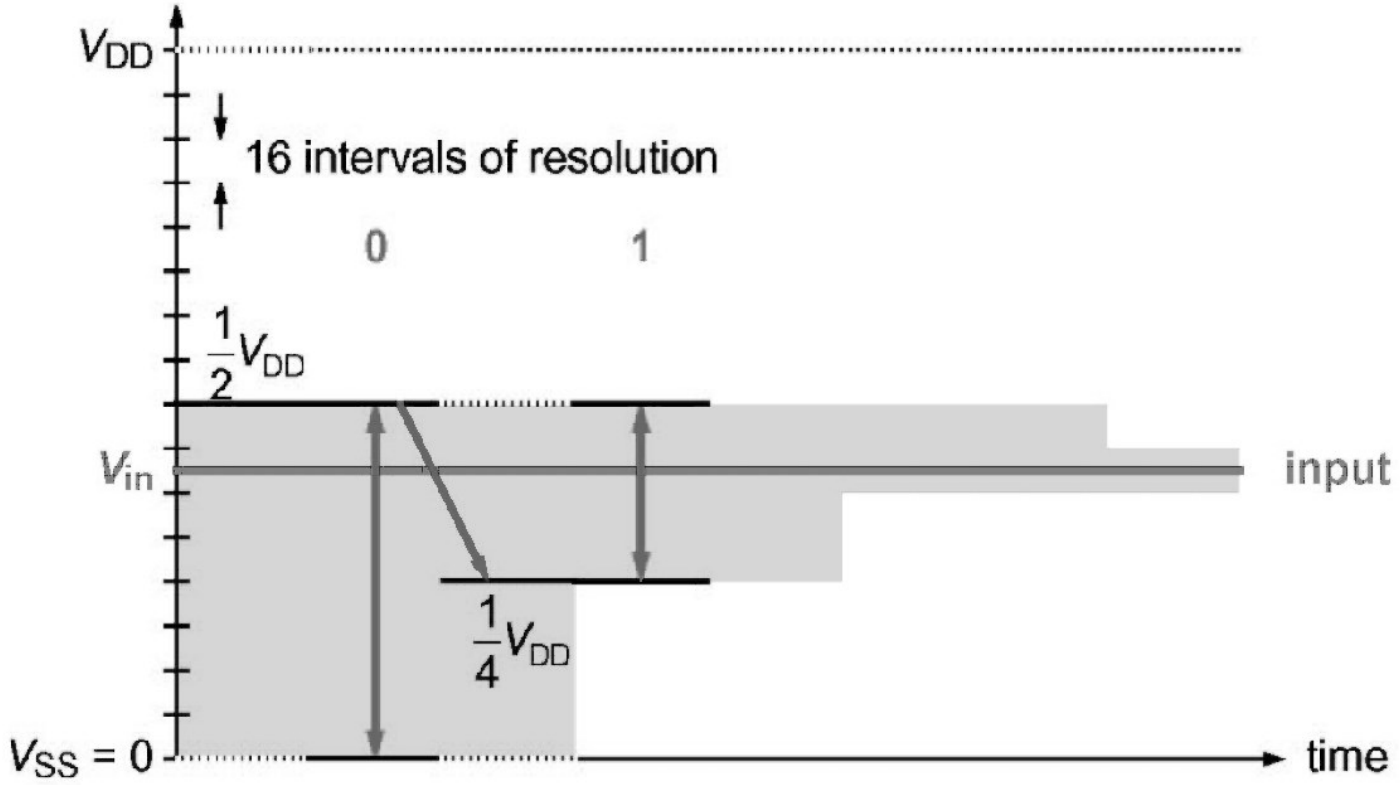
1. Full range: compare input with $(1/2)V_{DD}$. Find V_{in} smaller so bit = 0

Successive approximation technique: 4 bit ADC



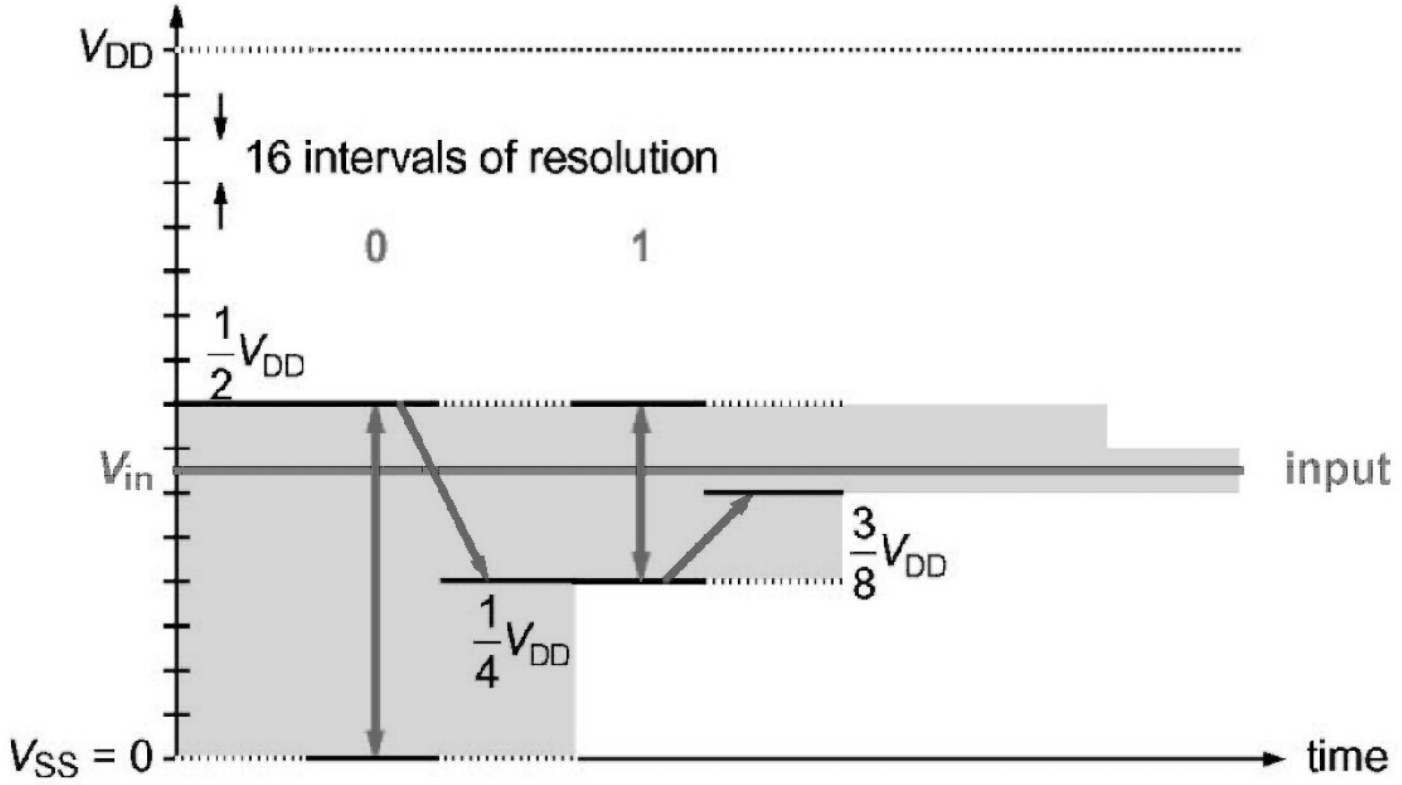
1. Full range: compare input with $(1/2)V_{DD}$. Find V_{in} smaller so bit = 0
2. Halve range: compare input with $(1/4)V_{DD}$.

Successive approximation technique: 4 bit ADC



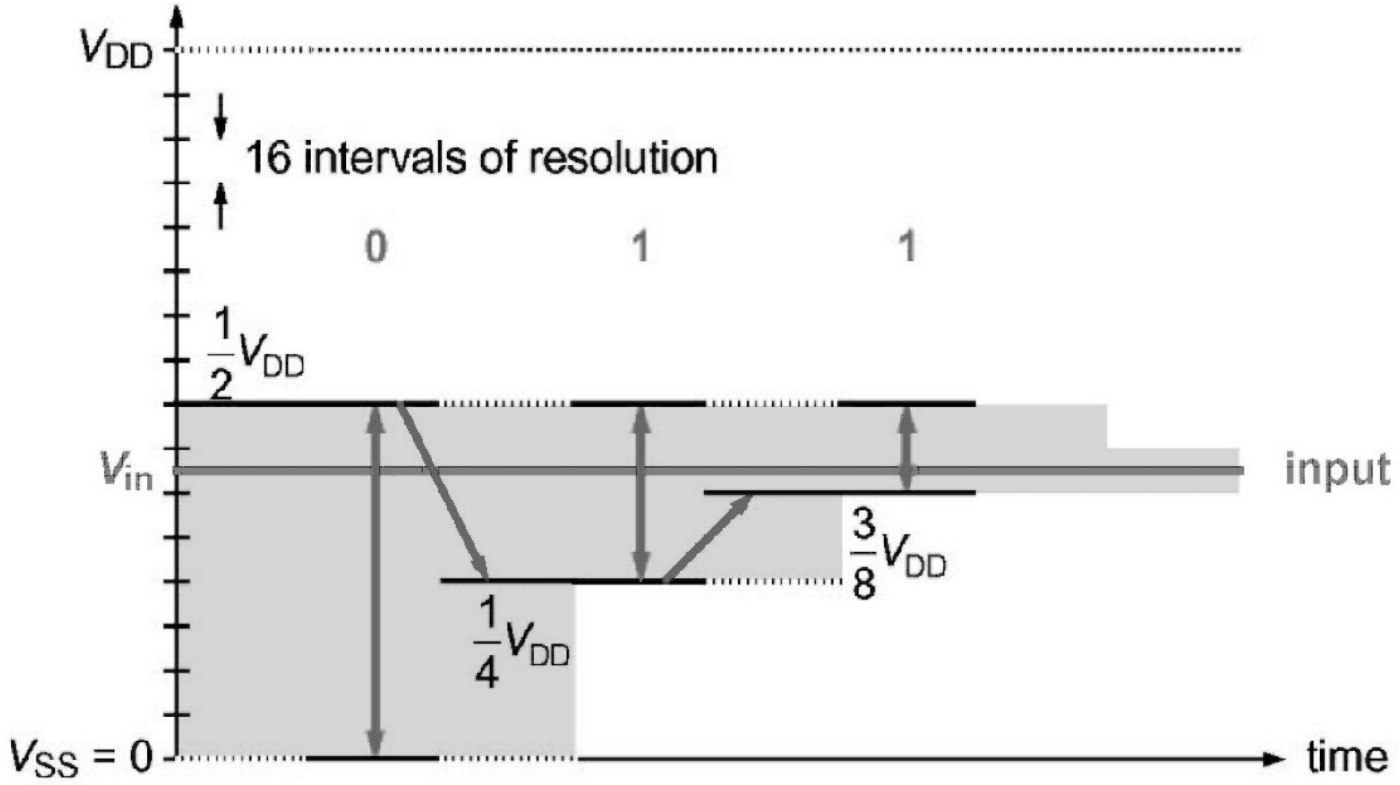
1. Full range: compare input with $(1/2)V_{DD}$. Find V_{in} smaller so bit = 0
2. Halve range: compare input with $(1/4)V_{DD}$. Find V_{in} larger so bit = 1

Successive approximation technique: 4 bit ADC



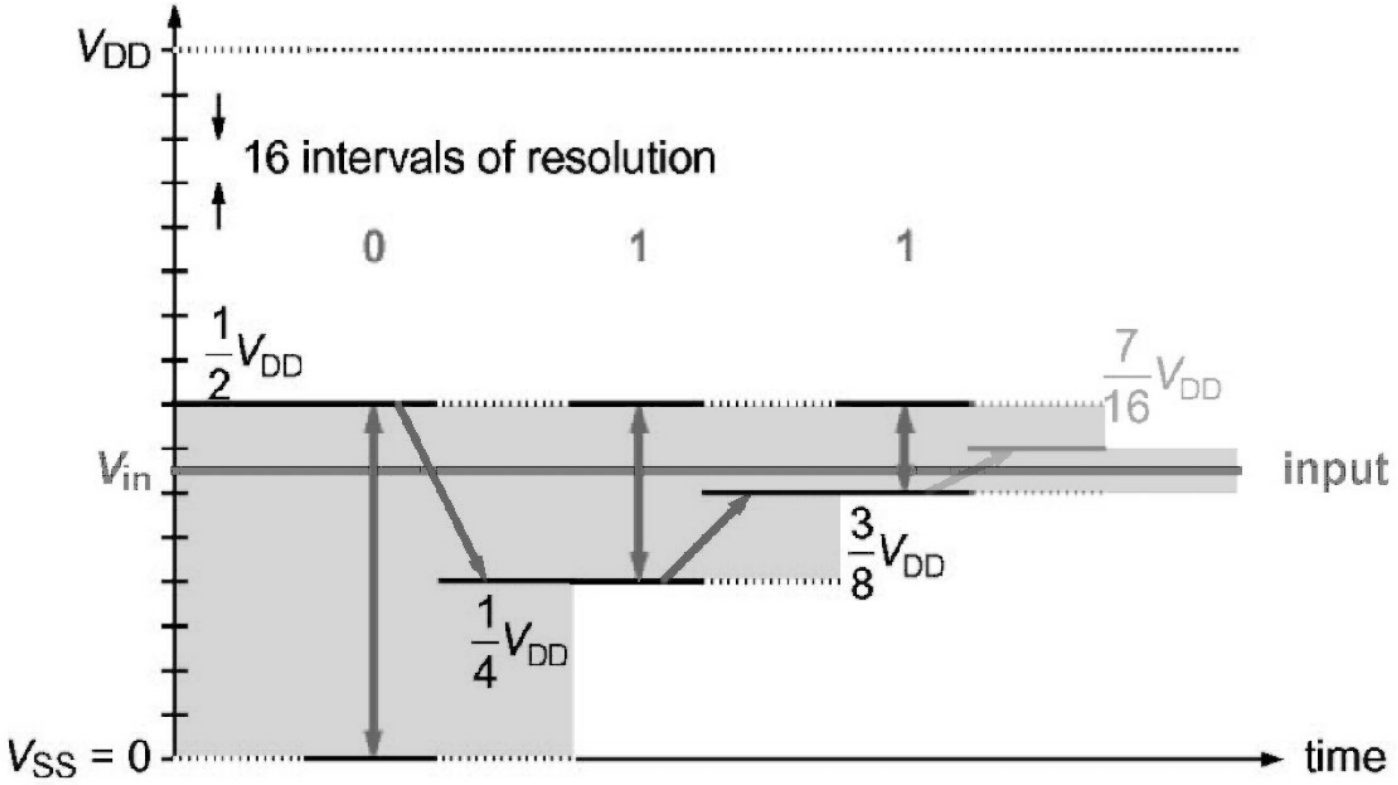
1. Full range: compare input with $(1/2)V_{DD}$. Find V_{in} smaller so bit = 0
2. Halve range: compare input with $(1/4)V_{DD}$. Find V_{in} larger so bit = 1

Successive approximation technique: 4 bit ADC



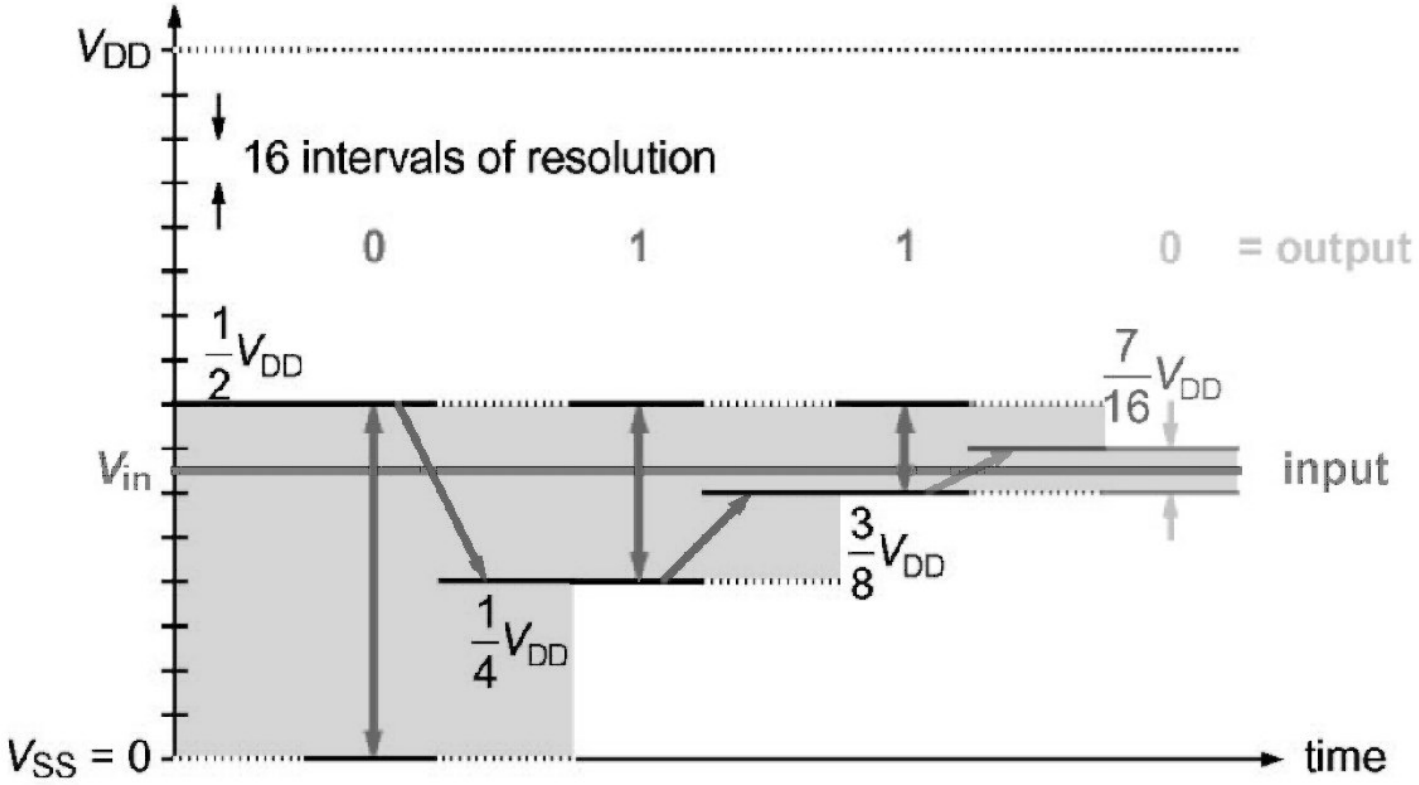
1. Full range: compare input with $(1/2)V_{DD}$. Find V_{in} smaller so bit = 0
2. Halve range: compare input with $(1/4)V_{DD}$. Find V_{in} larger so bit = 1
3. Halve range: compare input with $(3/8)V_{DD}$. Find V_{in} larger so bit = 1

Successive approximation technique: 4 bit ADC



1. Full range: compare input with $(1/2)V_{DD}$. Find V_{in} smaller so bit = 0
2. Halve range: compare input with $(1/4)V_{DD}$. Find V_{in} larger so bit = 1
3. Halve range: compare input with $(3/8)V_{DD}$. Find V_{in} larger so bit = 1
4. Halve range: compare input with $(7/16)V_{DD}$.

Successive approximation technique: 4 bit ADC



1. Full range: compare input with $(1/2)V_{DD}$. Find V_{in} smaller so bit = 0
2. Halve range: compare input with $(1/4)V_{DD}$. Find V_{in} larger so bit = 1
3. Halve range: compare input with $(3/8)V_{DD}$. Find V_{in} larger so bit = 1
4. Halve range: compare input with $(7/16)V_{DD}$. Find V_{in} smaller so bit = 0

Successive Approximation

- Electronic implementation uses a network of precision resistors or capacitors.
- They are configured to allow consecutive halving of the fixed voltage V_{ref} to be switched in to an analogue comparator.
- Sample and Hold process (capacitor network):

  _During the sample period, the network is connected to the analogue input voltage V_{in}

  _During the hold period, the analogue input is disconnected and the charge at each capacitor is examined by the comparator.

Successive Approximation - Summary

- The successive approximation method produces one bit for each comparison, starting with the most significant bit (MSB).
- Each step requires two clock cycles: one to change the reference voltage and one to make the comparison — 16 ADC clock cycles in all for an 8bit ADC.
- The conversion is done by an ingenious network of switched capacitors:
  - _ must be slow enough for charge to redistribute on capacitors
  - _ must be done quickly enough that charge doesn't leak away
  - _ hence ADC clock must be around 1 MHz, regardless of frequency of clock for rest of µC, which may run faster

Literature (1)

- **E. Williams: Make: AVR Programming**. ISBN 978-1-4493-5578-4. **00/TWG 33**
- S. Barret: Embedded Systems Design with the Atmel AVR Microcontroller, Morgan & Claypool Publishers. ISBN: 978-1-60845-127-2. **00/TWQ 82**
- R. Barnett: Embedded C Programming and the Atmel AVR, Cengage Learning. ISBN: 978-1-4180-3959-2. **00/TXU 41**
- J. Catsoulis: Embedded Hardware, O'Reilly. ISBN: 978-0-596-00755-3. **00/TWG 1**
- T. Floyd: Digital Fundamentals, a systems approach, Pearson. ISBN: 978-0-13-293395-7. **00/YGQ 6**
- T. Wescott: Applied Control Theory for Embedded Systems, Newnes. ISBN: 978-0-7506-7839-1. **00/TWQ 77**

Literature (2)

- J. Sanchez: Microcontroller Programming [The Microchip PIC], CRC Press. ISBN: 0-8493-7189-9. **00/TWS 5**
- F. E. Valdes-Perez: Microcontrollers [Fundamentals and Applications with PIC], CRC Press. ISBN: 978-1-4200-7767-4. **00/WGN 16**
- L. D. Jasio: Programming 16-BIT Microcontrollers in C, Newnes. ISBN: 978-1-85617-870-9. **00/TXU 31**
- J. Davies: MSP430 Microcontroller Basics, Newnes. ISBN: 978-0-7506-8276-3. **00/TWS 6**
- B. Schaaf: Microcomputertechnik, Hanser. ISBN: 978-3-446-43078-5. **00/TWG 13**

Microcontroller

Lecture 02: Programming

HOCHSCHULE RHEIN-WAAL
Rhine-Waal University of Applied Sciences

Programming the ATmega

- What is a program?
  - will look at instructions in both assembly and BASIC languages
- Look very briefly at instruction set of ATmega88pa
- Layout of assembly language
- Switch LEDs
  - all off (clear)
  - on or off individually (bit set and clear)
  - whole pattern simultaneously (move byte)

Choice of device

In the microcontrollers course we will work with **ATmega88pa**.

It will be a part of **AVR board** in most of our labs and projects, where additional hardware is mounted on the periphery of the **microcontroller**.

Important characteristics to understand:

- Memory for user programs: ***8 KBytes*** (we are talking about EEPROM, not available RAM). It is inside the ***ATmega88pa***.

  - datasheet

- RAM is limited to ***1 KBYTE***

  - datasheet

- Frequency: The processor supports frequency between ***1*** to ***20 MHz*** in most cases. AVR board will usually have ***8 MHz*** oscillator mounted on it.

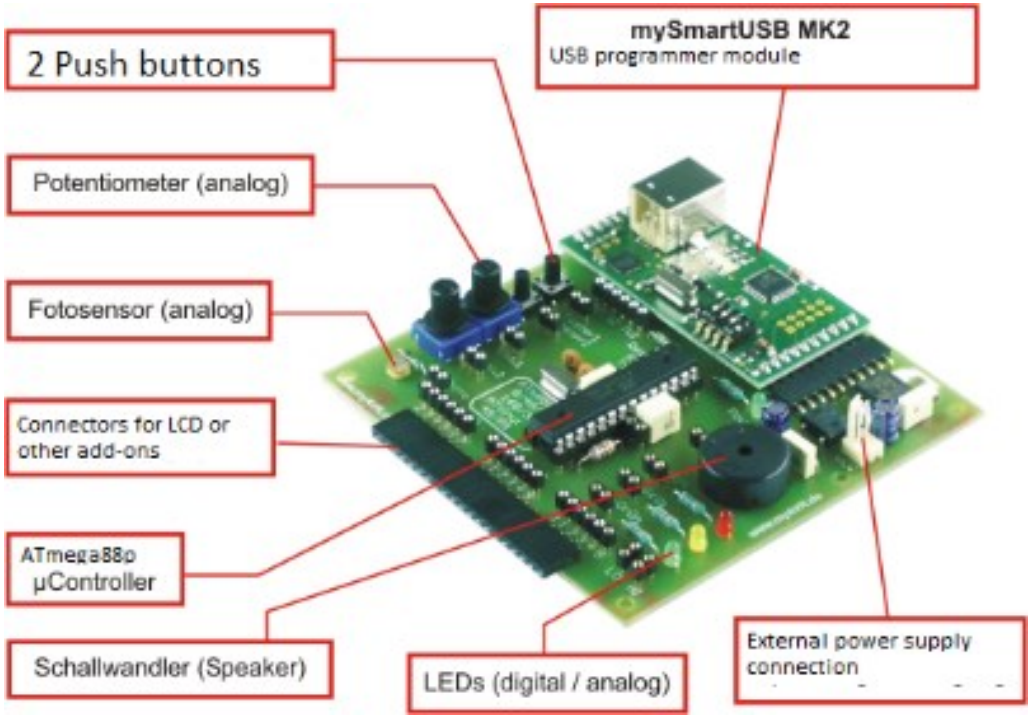
Remark: according to the info in datasheet ATmega88 should be able to go as low as 100 KHz and save some power at this frequency, but only with external oscillator.

Hardware

Before we can start programming, it is important that we know the **hardware around** the **microcontroller**.

Notice in the picture:

- 3 LEDs
- 2 Push buttons (we can program it to work with internal pullup)
- Potentiometers, Speaker (Buzzer)



What is a program?

What is a program?

A program is a sequence of instructions (‘code’) to be executed by a computer.

It may be organized in different ways (procedures, objects) and written in different languages (assembly, C, C++, Java),

All a processor can do, through , is execute instructions. It executes instructions one after another unless there is an explicit ‘branch’, ‘jump’ or ‘goto’ instruction. The complexity and “smart” behavior comes, therefore, from this sequence of instructions – **Your program**!

Why programming?

So why do companies create chips with complex processors instead of solving a particular task?

A great advantage of programmable device is the possibility to solve different tasks using same hardware! Changing software is ridiculously easy compared to creating new hardware.

Use this advantage – experiment with the software. If you do not know how something will work, most of the time it is safe to try! Only thing to watch out for is overcurrent – make sure outputs either have resistors or other limiting hardware connected. **For the rest – learn by trying!**

Assembly language

By the time a program is stored (‘burned’) in the program memory of a µC, it is just binary data (1s and 0s). It has a particular meaning to the µC and is called **machine code**.

Nobody in their right mind would want to use this (although some early computers had to be programmed in binary). Instead we use **assembly language** *(in practice, we will go one level higher – we will use programming language C).* This is just words (mnemonics) that can be translated directly into machine code. We shall look at a little of this to see how the processor executes a program.

An application called an **assembler** translates assembly language into machine code. It doesn’t do much else — substitutes constants and calculates addresses. It is feeble compared with a **compiler** for a higher level language, such as C, which does a much more thorough job.

Each family of processors has its own assembly language and uses a different notation — a nuisance. In contrast, higher-level languages are nearly independent of the platform used.

What can the processor do?

A feature of the ATmega88pa is that it has 131 instructions. It is still called a reduced instruction set computer (RISC). Instructions can be classified as follows.

Arithmetic — add, subtract, increment, decrement, clear

Logical — and, inclusive-or, exclusive-or, complement, rotate (shift)

Bit operations — set or clear individual bits in any file register

Subroutines — call and return

Move — move data between registers

Control flow (Jump, Branch, Call)— goto, skip instruction after testing a single bit in any file register, increment and skip, decrement and skip, no operation

and other (miscellaneous).

Writing assembly language

Assembly language has a rigid layout, which you must follow precisely! A typical line has four parts, separated by spaces or tab characters (to keep it neatly aligned and easier to read). Don't put spaces anywhere else.

- **Label** starting flush left. This can be omitted if you don't need a label but you must put the space or tab afterwards. A colon (:) is usually used to indicate a label. They are used for 'branches' in programs.
- **Instruction** (mnemonic for one of the 35 instructions)
- **Operands** (data needed), if any
- **Comment** — all text after a semicolon is treated as a comment and ignored by the assembler. Use comments freely because assembly language is incomprehensible without them!

Assemblers usually don't distinguish between UPPER and lower case.

```
lightUp:→    movwf→    7→    ; turn LEDs on in pattern
          tab          tab    tab
```

Light LEDs in a given pattern

We shall now write the core of a program to light LEDs in various patterns. Assume that we have LEDs on all 6 pins of port B (PB0–PB5). We'll do this in 3 ways:

- turn all LEDs off
- turn **individual** LEDs on and off
- turn **all 6** on or off in a given pattern

There are many ways of doing even a simple task like this!

NOTE — I assume that the microcontroller has already been configured correctly!

This includes:

- data direction registers
- port D set up as inputs — with pullups (or pulldowns) if needed
- port B set up as outputs

I'll also ignore what happens after the ATmega88pa has executed our instructions.

What happens if you write to an input?

There is nothing to stop you writing to a pin that is configured as an input.

The value does not appear on the pin (because it's an input) but is stored in an 'output latch'.

If you later reconfigure the pin to be an output, the value that was stored in the output latch will appear on the pin immediately.

So, it is in fact a good idea to write to the pin before it is made an output, so you know exactly what value will appear on the pin as soon as it becomes an output!

For example, you might want to clear all the output latches (set them to 0) before you configure the pins as outputs.

Turn all LEDs off (1)

It is a good idea to turn the LEDs off first — there is no guarantee that this happens automatically.

Remember that the LEDs are connected ‘active high’, so that for each bit:

- logic 0 output = LED off
- logic 1 output = LED on

The LEDs are controlled in assembly language by writing to the file register PORTB — memory mapped input/output.

We must therefore set all bits of PORTB to 0 to turn off all LEDs. The jargon is that we **clear** the register. There is an explicit instruction for this:

```
clrf f          ; clear file register f
```

Which file register? The memory map shows that PORTB has address 0x06 (hexadecimal!) so this will do the job:

```
clrf 0x06       ; clear PORTB, turn off all LEDs
```

Turn all LEDs off (2)

The PIC will be happy with this but it would drive a human mad if we had to use explicit addresses like 0x06 all the time.

Fortunately the assembler helps us here (one of the few ways in which it does): we can define names to stand for numbers, in the same way as **`symbol PORTB = $06`** in BASIC ($ for hex). Then we can write:

```
clrf PORTB    ; turn off all LEDs
```

Much clearer! Names like this are predefined for all common registers.

The assembler will:

- replace the symbol **PORTB** by the number **0x06 = 000 0110**
- replace the mnemonic **clrf** by its opcode **000 0011**
- write **00 0001 1000 0110** to one location of program memory, which is the full instruction that the PIC will process

In BASIC we can clear all the outputs at once by writing **let pins = 0**.

It is a good idea to do this before the pins are configured as outputs.

Light an individual LED — assembly language

Now suppose that we want to light a single LED — on PB0 for example. This means that we need to **set** bit 0 of PORTB (**make it 1**), leaving the other bits as they were. There is an instruction to do this:

```
bsf f,b         ; set bit b of file register f
bsf PORTB,0     ; light LED on RB0
```

That's it!

To turn the LED off we must **clear** the bit (make it 0), which needs

```
bcf PORTB,0     ; turn off LED on RB0
```

So this how the translation of your C code may look like before it is turned into 1s and 0s.

Light an individual LED — example

How would you light the LEDs in this pattern using the instructions for setting and clearing bits, either in assembly language or C?

Light a pattern of LEDs

We could light the LEDs in this pattern simultaneously by moving the correct byte into PORTB rather that setting each bit individually. Here is the pattern again.

Thus we need to move the literal binary value %00010101 into the file register PORTB.

In C, we can do it just by doing:

```
PORTB = 1 << PB4 | 1 << PB2 | 1 << PB0 ;
// if you are in a hurry, you could do PORTB = 21;
But why would it work?
Once again: 21 = 0001 0101 in binary representation
```

Assigning values to registers

So indeed,

```
PORTB = 1 << PB4 | 1 << PB2 | 1 << PB0 ;
```

And

```
PORTB = 21;
```

are obviously equivalent, as they result in the same value in the register. However, the first notation is preferable: it is much more explanatory, both to you and whoever reads your code. Somewhat better, but still bad: `PORTB = 0x15;`

As long as you are using gcc compiler (on our Ubuntu workstations), you can also do it like:

```
PORTB = 0b00010101;
```

But: Please describe in comments why you use a particular number!

```
(eg. // put 1s to bits PB4, PB2, PB0)
```

ATmega88p: Timer/Counter hardware

One of the tasks a microcontroller might have to tackle is timing signals / measurements.

By default, the only way a processor interacts with time is by clock pulses. So, to count time, you would want to count the clock pulses. The problem: it makes the processor **busy for any other tasks**!!

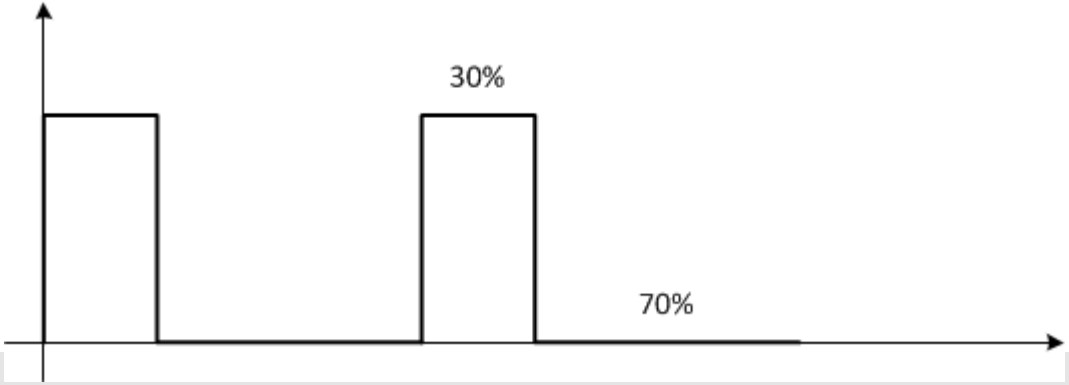
For this reason, separate hardware exists to count clock pulses on ATmega88p, as well as on many other microcontrollers.

Practicals: Timer / Counter 2

So far to manage time we were using delay. The most precise we got, however, was _delay_us(x) waiting for x microseconds.
This is not bad, but with frequency of 8 MHz we can actually work with units of 125 ns (1s/8M).

Furthermore, hardware has means to keep track of 3 different time values (timers). They can be incremented automatically, without taking processor time from the tasks you need to accomplish.

Task: Blink red LED with period of 425ms and green – 103 ms. Time on should be 30% of the period.



Task: Blink red LED(PB0) with period of 425ms and green(PB1) – 103ms. Time on should be 30% of the period.

>> can be done with delay and a long algorithm.

```
Int redOn, greenOn;              //flags to remember state of  LEDs
Int redCount, greenCount;        //counters to interact with time
While(1)

{
_delay_us(10);
redCount++; greenCount++;

If (redOn&&redCount==2975)     {PORTB^=(1<<PB0); redOn=0;redCount=0;}

If (greenOn&&greenCount==721)       {PORTB^=(1<<PB1); greenOn=0;greenCount=0;}

If(!redOn&&redCount==1275)      {PORTB^=(1<<PB0); redOn=1;redCount=0;}

If(! greenOn &&greenCount==309)       {PORTB^=(1<<PB1); greenOn=1;greenCount=0;}
}
```

>> Will solution precisely match the period required?
>> Can you run one more serious task on the uC in parallel to this? No! It will impact timings!

Right solution: Use timer/counter!

Example1:

TCCR1B &= ~(1 << CS10 | 1 << CS12); //setting to 0 is optional, as 0 is default.

*TCCR1B |= (1 << CS11); //Timer 1 on, using prescaler of **8**.*

*What will happen now? The register **TCNT1** will be incremented once per **8** cycles (according to the prescaler value, 8/8M = 1 us period).*
It will happen independent of any other operations you may be running.

Next slide will show how to use it to do periodic tasks.

Periodic tasks

Assume your task is: Send signal every 113 us.

```
If(TCNT1>=113) {TCNT1=0; PORTB^=(1<<PB0);} // >= is used in case you don't check timer for more then 2us and it goes from 112 to 114 without being checked.
```

Attention:

What would happen in this case if we use == operator?

Timer might tick to MAX value(65535), and overflow to 0 again. And … you will not send any signals!

What would happen without `TCNT1=0;` statement?

The timer would keep running up to overflow, potentially confusing your >= condition into triggering every loop!

Example2: Generate sound of freq. 500 Hz for 3 seconds, then 200 Hz for 2 seconds.

500 Hz implied a period = 2ms, during which we want high and low voltage on our piezoelectric buzzer. Therefore we change it every 1ms for first 3 seconds, then T2=5ms, i.e. we change it every 2500 us for the next 2 seconds.

We need one timer to measure duration in seconds, it is convenient to use Timer1 – it is 16 bit, and with max. prescaler of 1024 it takes 65535*1024/8M = 8,38848s to overflow.

So what value will refer to 3 seconds in TCNT1? One time unit is 1024/8M seconds, i.e. 128us. 3M/128 = 23437,5; now we have to round it, e.g. 23437 => 23437 * 128us = 2,999936 s.

Please, find TCNT1 value for 2 seconds!

Generating 500 Hz beat

Since Timer0 shares prescaler with Timer1, we have to use Timer2 to generate 500 Hz sound. With max value of 256 on 8 bit counter needing to match 1 ms, we may use prescaler of 64 (look up TCCR2 in atmega8 documentation):
Then 1 ms refers to 8000 processor cycles, 8000/64 =125.

How to initiate Timer2 with prescaler of 64? (atmega8 p.116)
What code can you use now to switch from high to low on the PB1 output every 1 ms? (remember to set TCNT2 to 0!)

What prescaler and TCNT2 value would refer to following times:

1) 13 us?
2) 400 ms?
3) 3,5 ms?

Microcontroller

Lecture 03: Control Flow

HOCHSCHULE RHEIN-WAAL
Rhine-Waal University of Applied Sciences

Flow of control in a program

- The fragments of programs that we wrote to light LEDs in different patterns were simple, linear sequences of instructions
  -did not react to inputs
  -what happens after the μC has executed the small number of instructions that we have given it? (Usually the μC is put to sleep after your last instruction to avoid any problems.)
- Flow of control in a program
  -**goto**, branch or jump instruction
  -loops
- Read input from a pushbutton
- Switch LED on when button is pressed
  -need a decision — **if** statement
- Design of a program — flowcharts

Educational objectives

During this lecture you will learn

- Is an infinite loop a disaster to be avoided?
- What is a flow of control in a program
  _**goto**, branch or jump instruction
  _loops
  →in Assembler language = the code which is processed by a MCU
- How to read input from a pushbutton
- How to switch LED on when button is pressed
  →corresponding instructions in Assembler language
- Design of a program — flowcharts, differences between
  →Polling
  →Interrupts

Arithmetic with MCU - recap 1
Logical & Bit operations

Bit operations:

- | = logical OR
- ~ = logical NOT
- & = logical AND
- ^ = logical XOR (exclusive OR)

Exercises (4 students): X = 15; Y = 12

1) Z = X & 0xA	1) Z = Y & 0xD	1) Z = X & Y	1) Z = X & 0x5
Z =	Z =	Z =	Z =
2) Y = 0x4 | (~Z)	2) Y = X | (~Z)	2) Y = Z | (~X)	2) Y = 0xC | (~X)
Y =	Y =	Y =	Y =
3) X ^= Y	3) X = Y ^ 0x5	3) X ^= (Y^0x9)	3) X ^= (Y - 0x7)
X =	X =	X =	X =

Arithmetic with MCU - recap 2
Two's complement

The two's complement representation of signed integers has the valuable property that two such numbers can be added using the same arithmetic rules as for unsigned integers. Example:

0011	3
+1110	-2
1 0001	1

Exercises (same 4 students): 2's complement values are given in binary representation (8 bits). Calculate the corresponding values in decimal representation

```
11111110  =
00110010  =
10011100  =
10000000  =
```

Control flow

The components of a program having control flow in it are:

• **subroutines** — small units, each with a well-defined function. This makes the overall program easier to understand, more reliable, and more efficient to run. Subroutines can be stored in libraries and re-used.

• **simple branches** — the infamous **goto** statement. Severely frowned upon in high-level languages but unavoidable in assembly language because processors only execute simple instructions, like ‘goto’.

• **decisions** — “if–then–else” in a high-level language, but only “if condition then goto” in both BASIC and assembly language.

• **loops**, for operations that must be repeated

_for ever (infinite loop)

_until a condition is true

_a fixed number of times

The goto instruction (1)

This is the simplest instruction that changes the flow of control. It may be called a goto, jump or branch. Basically it looks like this:

```
goto n          ;  execute instruction in memory
                           location n next
```

Usually the processor executes instructions in program memory one after the other.

Here, the instruction in location 0x03 is `goto 0x06`, so the processor will execute the instruction in memory location 0x06 next, instead of that in 0x04.

It skips the instructions with addresses 0x04 and 0x05.

The goto instruction (2)

Instead of using line number in goto instruction, we can use a *label*. The instruction `goto label` means execution will continue from instruction immediately after `label`. Here is the equivalent of the previous example.

```
    instruction 0          ; tab at start of each line
    instruction 1
    instruction 2
    goto   jumpHere        ; jump to labelled instruction
    instruction 4
    instruction 5          ;
jumpHere:                  ; no tab at start!
                             colon to show that it's a label
    instruction 6
```

This works in exactly the same way in C.

[goto is considered a "taboo" in professional programming – it makes very hard to understand where your program will end up, especially if there are multiple goto instructions! You will soon learn about more safe ways to control execution flow]

The goto instruction (3)

The example that we have just seen looks fairly pointless, but becomes powerful when combined with an ‘if’ instruction later.

However, the **goto** becomes much more useful if it jumps backwards rather than forwards.

What does this do?

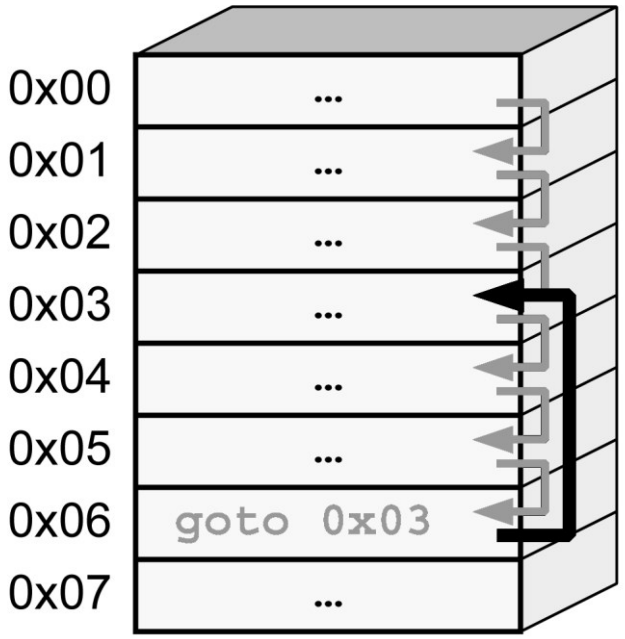
The processor executes the instructions in 0x00, 0x01, 0x02, 0x03, 0x04 and 0x05.

In 0x06 it encounters the `goto 0x03`.

It returns to 0x03, executes that instruction, and goes on to 0x04 and 0x05 again.

It then encounters the `goto 0x03` again and jumps back to execute the instructions in 0x03, 0x04 and 0x05 yet again…

It performs an infinite loop.



Infinite loops

Here is the assembly code for such an infinite loop.

```
    instruction 0
    instruction 1
    instruction 2
loopStart:
    instruction 3
    instruction 4
    instruction 5
    goto loopStart   ; back to beginning of loop
```

If you have written a simple program to run on a PC, you will probably think that an infinite loop is a disaster to be avoided — your program gets stuck in the loop and won't do anything else, so it has to be killed.

In contrast, a microcontroller keeps running at all times that the equipment is turned on (unless it is put to sleep), so programs always have infinite loops or they would run out of instructions. (This is also true of operating systems.)

Empty loops

You can even make completely empty loops, like this:

```
loopStart:
    goto loopStart ; back to start of loop
```

This simply repeats the **`goto`** instruction for ever, until some interruption occurs.

Why would anybody want to do this?

- It would be a simple ‘trap’ to put at the end of our simple programs to light LEDs — it keeps the processor executing well-defined (even if useless) instructions. Without this, the processor would step through whatever instructions were in its program memory until it hit the end (and crashes).
- Many respectable programs contain only an empty loop in their main routine (after initialization and so on). The processor sits in this (useless) loop until an **interrupt** occurs. It ‘services’ the interrupt, returns to the empty loop, and waits for the next interrupt.

How does a µC know when to respond to an input?

There are two opposite ways of handling this. Suppose that I am writing lectures at home one evening while waiting for my wife to return. (She has lost her key so I need to unlock the door for her). I could do two things:

- I go and check to see if she is waiting outside the door after every page of notes. This is called **polling**.
- I carry on writing lectures until **interrupted** by the door bell. I finish the sentence in my lecture notes so that I can resume them easily, and only then go to let her in. Afterwards, I return to writing lectures.

These two methods are widely used and have advantages in different circumstances. (The choice is clear for this example!)

- **Polling** is simpler to program because everything is predictable, and is more efficient for handling frequent events.
- **Interrupts** are tricky to program because we must 'tidy up' before handling the interrupt and resume normal working smoothly afterwards, but are more efficient for infrequent events.

Light an LED when a button is pressed - recapitulation

Here is the simplest task where a µC needs to respond to its input:

- **light an LED on pin PB1 when a pushbutton on PD2 is pressed**

You have done this in the laboratory with a few lines of C.

```
#define F_CPU 8000000 UL
#include <avr/io.h>

int main (void)
{
                    // PD2 as Input
                    // Pullup PD2
                    // PB1 as Output

      while(1)// Infinite loop
      {
             // Read pin status of PIN PD2
             if
             {

             }
             else
                          );
       }
return (0);
}
```

This is effectively polling the input every time the program goes around the loop.

Test input with an ‘if’ statement

We would like a program something like this:

```
infLoop:
    if (pin3 = 1) then      ; button is pressed
    let pin0 = 1            ; turn LED on
    else                    ; button is not pressed
    let pin0 = 0            ; turn LED off
    goto infLoop            ; back around infinite loop
```

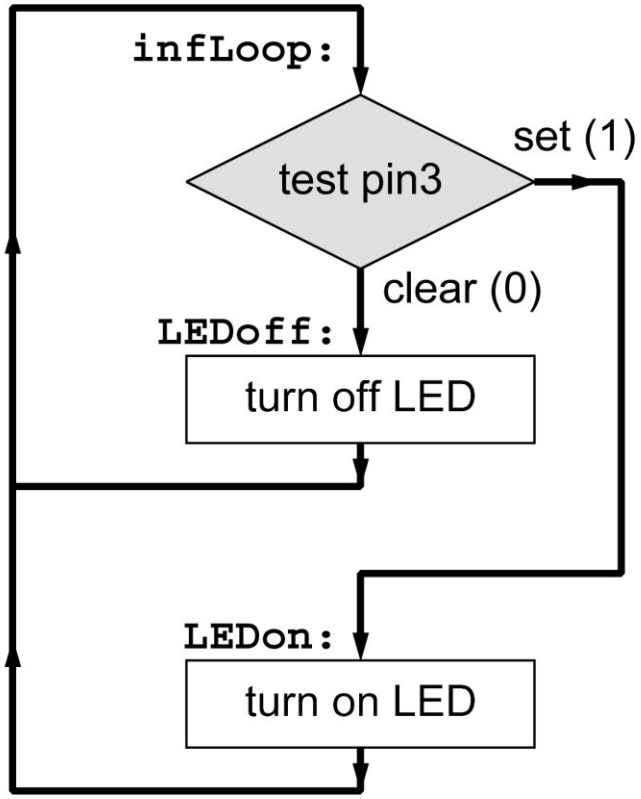
This would be fine in a higher-level language but neither PBASIC nor assembly language offers if–then–else. In fact the only statement allowed after an `if` in PBASIC is a `goto`. So we have to resort to this:

```
infLoop:
    if (pin3 = 1) then [goto] LEDon
    LEDoff:             ; button is not pressed
    let pin0 = 0        ; turn LED off
    goto infLoop        ; back around infinite loop
    LEDon:              ; button is pressed
    let pin0 = 1        ; turn LED on
    goto infLoop        ; back around infinite loop
```

(The label **LEDoff** isn’t essential but improves clarity — good idea!)

Test input with an ‘if’ statement — flow diagram

This example is trivial but things soon get complicated. A simple tool to help in the design of such programs is a **flow chart** or **flow diagram**.




```
infLoop:
  if (pin3 = 1) then goto LEDon
LEDoff:
  let pin0 = 0
  goto infLoop
LEDon:
  let pin0 = 1
  goto infLoop
```

Compare assembly and BASIC programs

The final programs are very similar (with a little sleight of hand!).

```
infLoop:
    if (pin3 = 1) then
                [goto] LEDon
LEDoff:
    let pin0 = 0
    goto infLoop
LEDon:
    let pin0 = 1
    goto infLoop
```

```
infLoop:
    btfsc  PORTA,3
    goto   LEDon
LEDoff:
    bcf    PORTB,0
    goto   infLoop
LEDon:
    bsf    PORTB,0
    goto   infLoop
```

The main difference is that the test really needs two lines of assembly language, and a lot of care is needed to get it the correct way around.

More complicated constructions are usually a lot simpler in BASIC or a high-level language.

Second approach — outline

There are always many ways of solving a problem, even as simple as the LED and pushbutton. Here is another approach, which is often useful.

- turn LED off
- loop (doing nothing) while button remains up

_fall out of loop when button pressed

- turn LED on
- loop (doing nothing) while button remains down

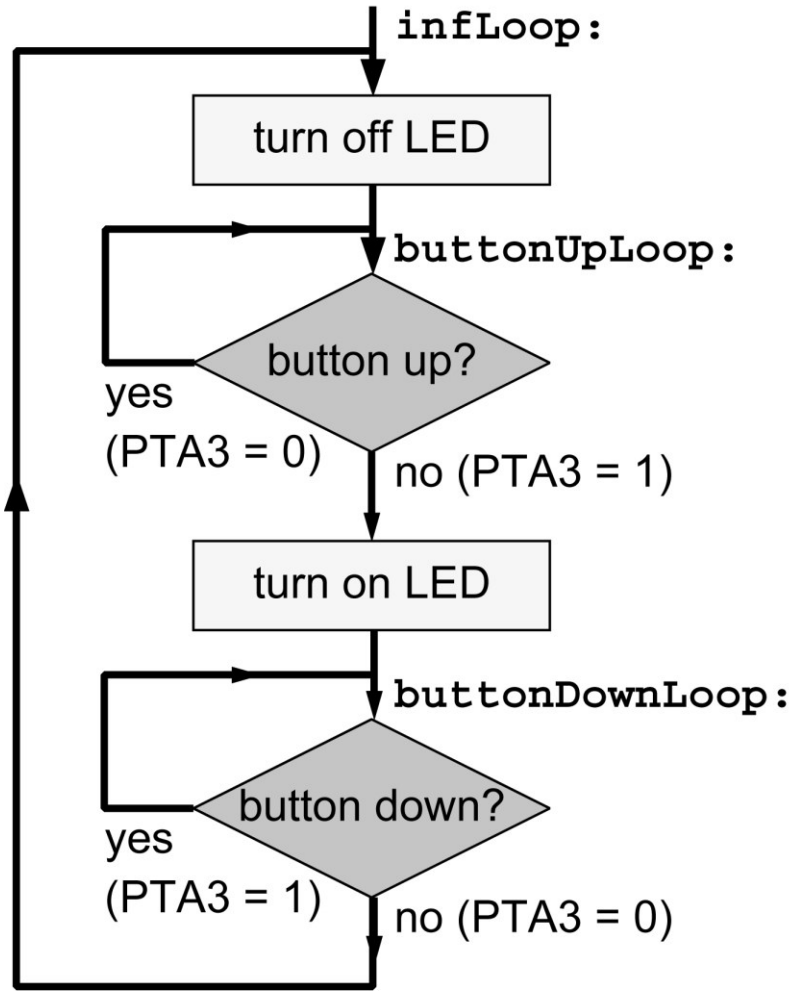
_fall out of loop when button released

- back to beginning

In this case the action takes place only at the **transitions**, when the button is pressed or released.

This can be programmed using the same instructions, but now the tests control the duration of the loops rather than the flow within the loop. The loops are no longer infinite — they are like ‘while’ loops.

Second approach — flow diagram and code




```
bcf     PORTB,0

PTA3 is low if button up,
skip loop when pressed (high)
btfss   PORTA,3
goto    buttonUpLoop

bsf     PORTB,0

PTA3 is high if button down,
skip loop when released (low)
btfsc   PORTA,3
goto    buttonDownLoop

goto    infLoop
```

Compare assembly and BASIC programs

The final programs are again very similar.

```
infLoop:
    let pin0 = 0
;   loop while button up
buttonUpLoop:
    if (pin3 = 0) then
      [goto] buttonUpLoop
;   button pressed, light LED
    let pin0 = 1
;   loop while button down
buttonDownLoop:
    if (pin3 = 1) then
      [goto] buttonDownLoop
;   button released, kill LED
    goto infLoop
```

```
infLoop:
    bcf    PORTB,0
;   loop while button up
buttonUpLoop:
    btfss  PORTA,3
    goto   buttonUpLoop
;   button pressed, light LED
    bsf    PORTB,0
;   loop while button down
buttonDownLoop:
    btfsc  PORTA,3
    goto   buttonDownLoop
;   button released, kill LED
    goto   infLoop
```

Conclusions

Seen how to control the flow of a program in assembly language
- relies heavily on the goto instruction
- infinite loops — always present

Contrasting approaches to reacting to inputs
- polling — simple but inefficient for infrequent events
- interrupts — complicated but may be more efficient

Decisions
- ‘test bit and skip’ instructions in assembly language
- equivalent of ‘if–then’ in BASIC
- easy to get it wrong in assembly language!

Constructed program to light an LED when a button is pressed
- can now read inputs and write to outputs of microcontroller
- these are the major hurdles: the worst is over!

What is an Interrupt?

Interrupts can be used when some immediate action is required from uC, or just as an convenient way to react without sampling for signal (what is optimal – checking if your friend has come every 5 minutes or waiting for the bell to ring?).

Every interrupt enabled must have corresponding instruction – Interrupt Service Routine(ISR).

In the code following syntax is used:

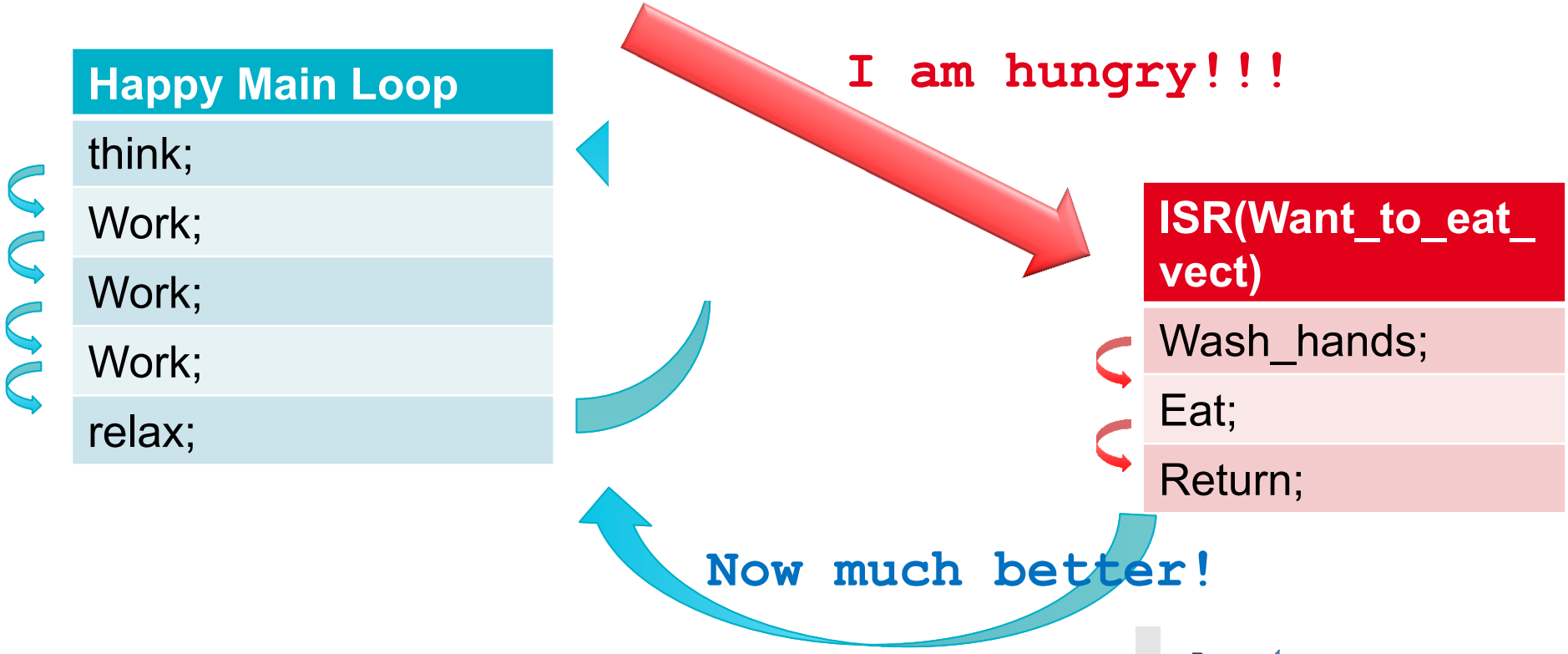
```
ISR(INTERRUPT_NAME_vect) // look up the name in atmega8 doc
{                                  // it is in the section “interrupts”, p.46
YOUR CODE;
}
```

This code is normally placed after the main function is closed.

What is an Interrupt (hardware level) ?

We know that processor / ALU goes through a series of instructions, and there is a control flow operations.

*** there are details. For example, max. amount of interrupts in memory is 8, meaning you should be processing them quicker than calling!*



What interrupts does atmega88p have?

Vector No.	Program Address	Source	Interrupt Definition
1	0x000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x001	INT0	External Interrupt Request 0
3	0x002	INT1	External Interrupt Request 1
4	0x003	PCINT0	Pin Change Interrupt Request 0
5	0x004	PCINT1	Pin Change Interrupt Request 1
6	0x005	PCINT2	Pin Change Interrupt Request 2
7	0x006	WDT	Watchdog Time-out Interrupt
8	0x007	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x008	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x009	TIMER2 OVF	Timer/Counter2 Overflow
11	0x00A	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x00B	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x00C	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x00D	TIMER1 OVF	Timer/Counter1 Overflow
15	0x00E	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x00F	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x010	TIMER0 OVF	Timer/Counter0 Overflow
18	0x011	SPI, STC	SPI Serial Transfer Complete
19	0x012	USART, RX	USART Rx Complete
20	0x013	USART, UDRE	USART, Data Register Empty
21	0x014	USART, TX	USART, Tx Complete
22	0x015	ADC	ADC Conversion Complete
23	0x016	EE READY	EEPROM Ready
24	0x017	ANALOG COMP	Analog Comparator
25	0x018	TWI	2-wire Serial Interface
26	0x019	SPM READY	Store Program Memory Ready

- **External voltage signal causes interrupt(PD2, PD3)**
- **Timer/Coutner interrupts**
- **Serial interrupts**
- **(no color) ADC, EEPROM and other…**

p. 57-58 of ATmega88pa datasheet

Examples on using interrupts

using interrupts in Timer/Counter:
So how to do something exactly at the moment timer scores 103 (or your target value)?

We need interrupt to react to such an event.

All timer/counters have compare registers. They can initiate interrupt upon timer reaching the value to which such register is set.

Example:

```
TIMSK1 |= 1<<OCIE1A; //enable compare interrupt, TIMSK1 means
Timer Interrupt MaSK for timer/counter 1
OCR1A = 125;

ISR(TIMER1_COMPA_vect)
{
PORTB^=1<<PB0;
}
```

Examples on using interrupts

Sometimes executing your normal program can be more important than your interrupts. To decide on this you have sei(); and cli(); statement.

You start with interrupts disabled.
sei(); enables interrupts.
cli(); disables interrupts.

//REMARK normal variables can not be changed in an interrupt. Please add volatile modifier in the declaration to enable this!

```
volatile int myNumber1=1, myNumber2=2;
```

More possibilities with interrupts

With the interrupt you can do more useful things, example:

- **Make phase-correct response to an event with unknown or changing period**
- **React quickly to an important external event**
- **Act at a precisely defined moment**
- **Read serial input when it is ready**
- **Or just use them to sequence your program to your preference..(but do not complicate your code for no good reason!)**

Please discuss which interrupts can be helpful in every case!

Microcontroller

Lecture 04: Communication

HOCHSCHULE RHEIN-WAAL
Rhine-Waal University of Applied Sciences

Communication

Most microcontrollers communicate with other digital devices

- range of requirements — may be on same printed circuit board (PCB), with a local PC, or across the world
- wide selection of approaches in both hardware and software

I^2C — inter-integrated circuit bus

- simple synchronous bus, often used to connect peripherals to a µC
- examine how it transmits a byte

RS-232 — ancient serial protocol

- asynchronous, often used to link equipment to a PC
- examine how it transmits a byte
- some practical issues, such as generating the weird voltages needed

Implementation of communications

- interfaces for both of these (and many others) available in µCs

Why communicate?

Almost all computers these days can communicate with other computers over networks — the internet.

Many (most?) microcontrollers also need to communicate with other digital devices.

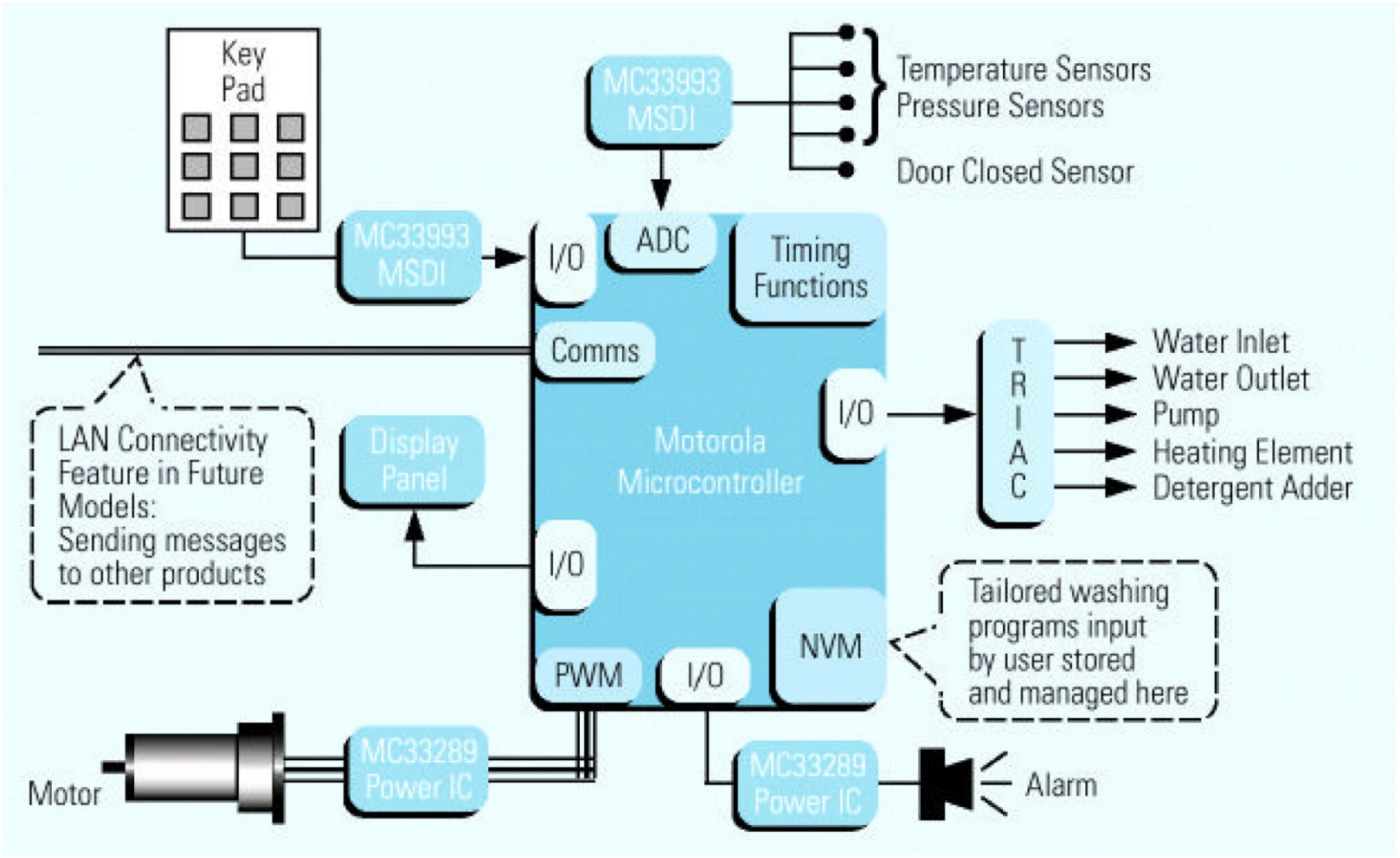
- in many cases this is the main purpose of the microcontroller: remote controls, keyboards…
- many microcontrollers are used primarily as interfaces to larger digital systems (the origin of PICs)

In many cases the scale of communication is modest

- keyboard and other human input devices: few bytes per second
- sensors are rarely called more than a thousand times per second, usually much less frequently: few kilobytes per second

Wide range of simple interfaces available.

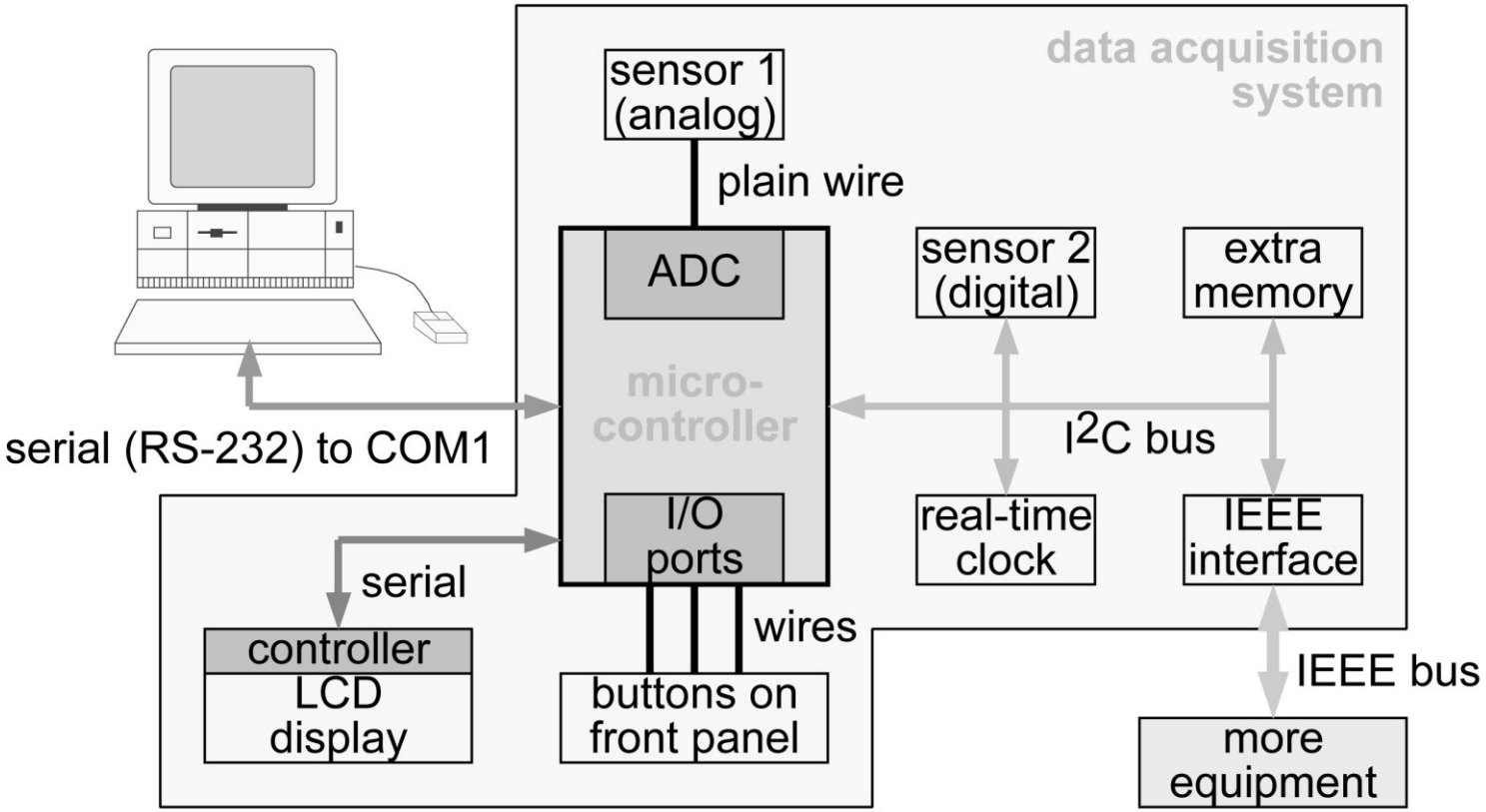
Washing machine (again)



http://e-www.motorola.com/files/shared/doc/selector_guide/SG2040.pdf

Example — data acquisition system

We require standalone equipment to store data from sensors and upload it to a PC.



Communication within a piece of equipment

In many cases, this means communication with other devices on the same printed circuit board (PCB). Examples:

- user interface (such as keypads, selector switches) — can easily run out of pins on the ports of a small µC
- sensors with digital interfaces
- additional memory, perhaps for data logger or processing
- displays more complicated that the µC can drive directly — such as LCD displays, which usually have their own µC built in
- other microcontrollers in a big system

There is a wide range of simple interfaces for dealing with these: **serial**, microwire, serial peripheral interface (SPI), inter-integrated circuit (**I²C**) bus, LINbus, one-wire bus….

Many microcontrollers have these interfaces built in, which makes them very easy to use. Usually all of them provides a serial interface.

We'll look at serial interface (RS232) and I²C in detail.

Communication between pieces of equipment

This covers a huge range of possibilities. Examples:

- components of a fancy hi-fi or home cinema system may be linked with a control system (as well as audio and video) — sometimes I^2C
- test equipment — often linked together with a specialized interface called IEEE-488, GPIB or HPIB (uses weird and expensive cables)
- control of a large building (heating, cooling, ventilation, lighting, fire,…) might be linked on a network — smart buildings

These have tended to use proprietary systems in the past but are increasingly based on the internet for larger systems, such as buildings.

Suppose that you want to run an experiment from home.

- In the past you might have used a modem, which looks much the same as a plain serial connection (except that it isn't always available)
- Modern equipment and the software for controlling it (such as LabView) often has a web interface — too complicated for us!

Cars

When I was a boy, the wiring diagram for a car would fit comfortably on a small page. However, the increasing complexity of the electrical system in a car could need up to 5 km wire, weighing 100 kg (what does a car weigh?):

- engine has sophisticated electronic control
- anti-lock braking systems, airbags and other safety systems
- ever more fancy entertainment and climate control systems
- even doors are full of electric mirrors, locks, windows, airbags…

This led to the development of a specialized network for cars — **CANBUS**.

- designed to be robust physically and electrically, with several levels of error checking
- needs a dedicated interface, either a separate chip or built into a μC

There are also many another networks.

Communication with a computer

This is a very common requirement and there are several connections on a PC that may be used:

- **serial port (COM1 etc) — review in more detail**
- parallel (printer) port
- keyboard and mouse ports
- universal serial bus (USB), a more modern interface
- FireWire, similar but more specialised
- the communication may be wireless, e.g. Bluetooth, but these often go via a serial port or USB
- for fast, fancy applications can use the bus inside the PC (e.g. EISA)

Of course the microcontroller was probably programmed through a connection to a PC in the first place! Traditionally this used the serial port but USB is now taking over.

Types of communication (1)

Serial or parallel

- serial — data is sent as a stream of bits, only one at a time
- parallel — many bits (usually 1, 2 or 4 bytes) are sent at the same time

Most communication is serial because it is simpler and needs fewer wires. The exception is when high performance is needed, such as for external hard disks (SCSI) — but these may now use FireWire, which is serial. An exception is the venerable parallel port on PCs.

May connect only two items of equipment (nodes) or many

The result is a network or bus if more than two nodes are connected, and most modern interfaces are of this kind.

- serial port (2 items only) superseded by bus (USB), up to 127 nodes

Duplex

- full duplex — can send data in both directions at same time
- half duplex — can send in both directions but not at same time

Types of communication (2)

Synchronous or asynchronous

- synchronous — a clock signal is sent along with the data
- asynchronous — the receiver must generate its own clock to match that in the transmitter

The 'clock' defines when signals contain valid data — see example of each.

Communications is a complicated topic!

The system can be broken into many levels from the physical (wires, optic fibre,…) to the high-level software applications (web browser…).

- 7-level ISO/OSI model
- can use different physical media for same logical signals (internet)
- can send different logical signals over the same physical medium (e.g. Ethernet can carry internet, AppleTalk, Novell, Token Ring,…)

Major topic: need to ensure that data arrives safely, messages don't collide when two nodes try to transmit at same time, and so on — **protocol**.

The I²C bus — a simple, synchronous interface

The **I2C** or **inter-integrated circuit** bus is a very widely used, simple, synchronous interface. It was introduced by Philips in 1992 and is often used to interface a microcontroller to:

- sensors, real-time clock/calendars, ADCs, DACs, extra I/O ports
- memory (RAM or Flash) — as on Stamp boards (possibly)
- interfaces to other systems such as displays, IEEE-488
- other microcontrollers, sometimes other equipment (far beyond the original intention, which is the mark of good design!)

It needs two wires and is often called the '2-wire interface':

- **SDA** — serial data
- **SCL** — serial clock

(of course it needs ground as well, so there are really 3!)
It was originally limited to 100 kbit/s but newer versions can run faster.
SMBus is virtually identical, and SPI and Microwire rather similar.

Why do we need a protocol?

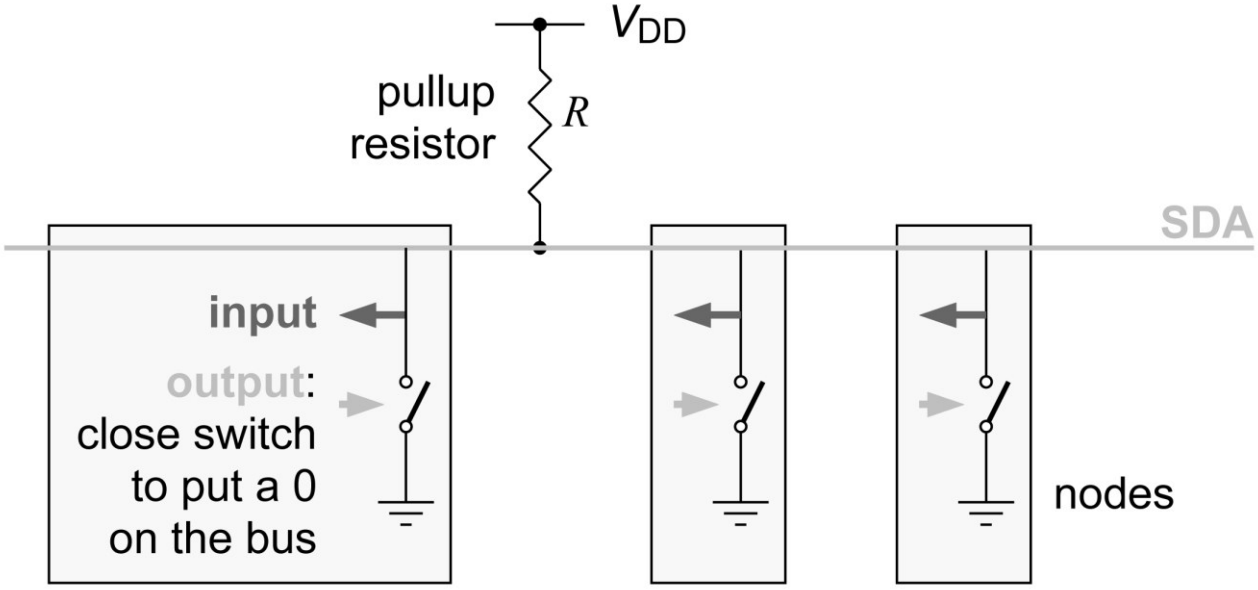
Any form of communication must follow certain rules if it is to work reliably: it needs a protocol. This is particularly important for a bus, such as I^2C, where many devices may be involved. For example, we need to define:

- **who is in charge** — what happens if more than one device wants to use the bus?
- **when does a new transmission start?**
- **precisely when is the value on the data line valid** and should be read by the receiver? (as opposed to when the transmitter is changing the value between bits)
- **when has transmission finished?**
- **has the message been successfully received?**

There is a **master** node in I^2C, which 'gives all the orders' to the other nodes and generates the clock. (Actually there can be several masters, but only one active at any time, with rules for resolving what happens if more than one tries to take control.)

How are the nodes connected together?

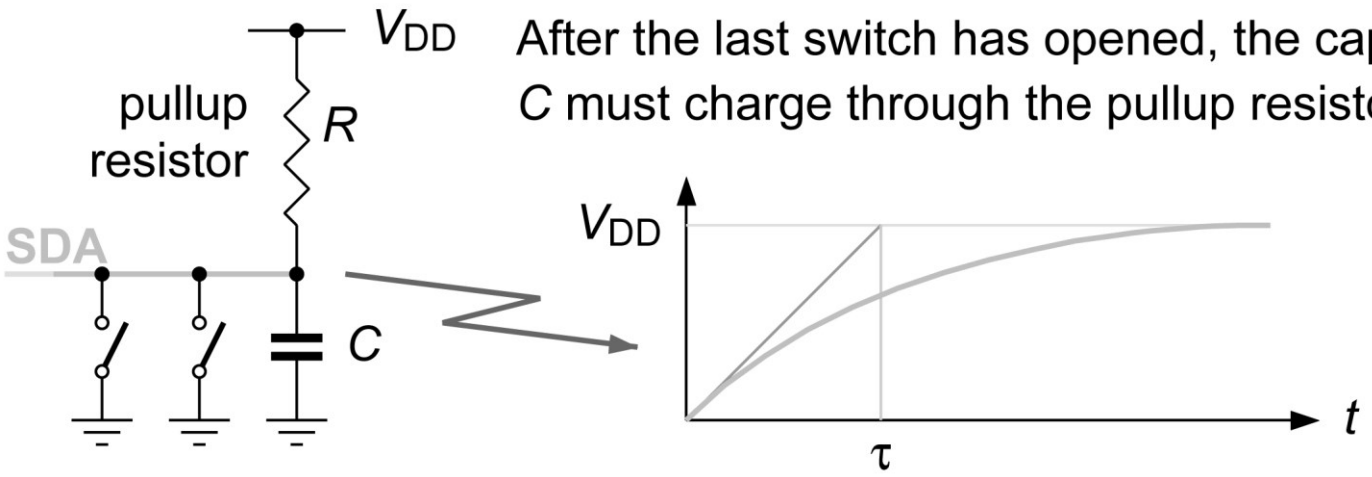
The nodes must all be connected to the SDA and SCL lines in such a way that no damage can arise if more than one node tries to drive the line. This is done in a very simple way, which you may recognise.



If no outputs are active, SDA is pulled up to *V*DD (logic 1) by the resistor. When one output is active and its 'switch' is closed, SDA is brought down to ground (logic 0). Further active outputs have no further effect, nor damage.

Aside —a little electrical engineering

Unfortunately SDA is not a perfect wire but, like all real components, has capacitance to ground. The circuit can be represented like this.



The time constant = RC and it takes about 3 for the capacitor to become 95% charged (Basics of Electrical Engineering OR Control Theory). Typically R = 5 kΩ and C = 200 pF, giving = 1 µs. This limits each cycle to about 10 µs, a frequency of 100 kHz (or 100 kbit/s).
This problem afflicts all digital circuits and limits their speed.

The I²C protocol

The basic rules of I²C are simple (shown pictorially next):

- Both SDA and SCL float high when idle (pullup resistor)
- SDA may change only when SCL is low (logic 0)
- SDA is valid when SCL is high
- the receiver sends a low 'acknowledge' bit to confirm receipt
- to start a transmission, pull SDA high to low when SCL is high
- to finish, SDA goes low to high when SCL is high

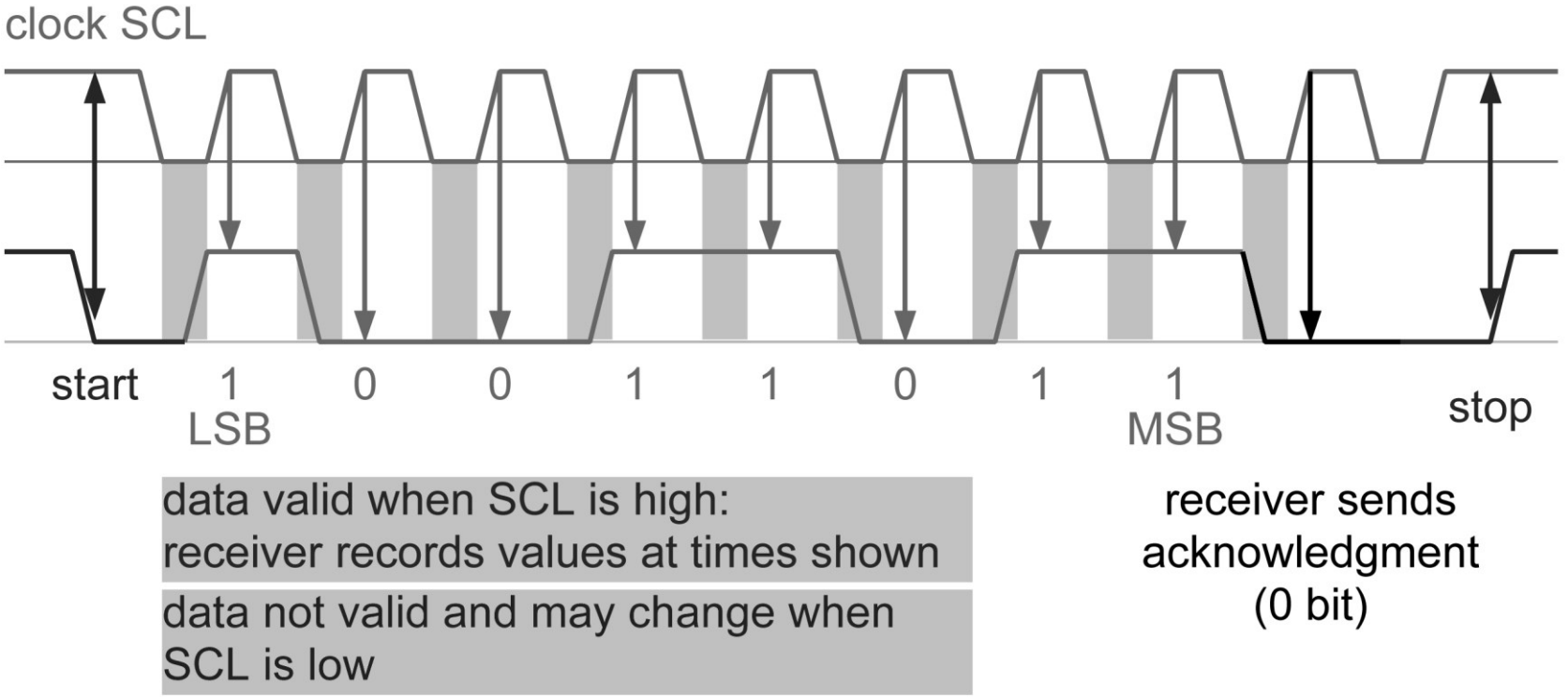
Data are transmitted in bytes (8 bits), each followed by an acknowledgement bit.

Usually, a transaction involves the master sending a command to a slave (such as a sensor), which then responds.

All messages start with the address of the target, which occupies at least one byte.

A one-byte message in I²C

Here is a message of a single byte for illustration (useless in practice).
Lower signal is data SDA, top is clock SCL.



This is the complete transaction: everything is clearly defined by the rules.
The clock makes it clear when SDA is valid and should be recorded.

Implementation of I²C

I²C is simple enough that it can be implemented in software: a programmer specifies the actions in detail and any digital I/O pin is used. This is called **bit-banging**. A µC with a 1 MHz instruction clock will be fully occupied in dealing with a transmission at 100 kbit/s (10 instructions per bit).

If this is unacceptable, many µCs have interfaces for I²C built in — look for a **synchronous serial port**. The main program moves the data to a special register and gets on with other tasks; the port deals with the detail of I²C and tells the main program when it has finished. (Hardware also filters any interference on the bus and ensures that signals follow the specification.)

You aren't expected to remember all the details of I²C (whew!) but should understand the principles: particularly that **the clock defines when data are valid**.

There are many other ways of transmitting data synchronously. Often there is no separate clock but the data are coded in such a way that the clock can be recovered from the data by the receiver. USB works in this way, for instance.

The traditional asynchronous serial port

This is the COM port, still to be found on most PCs (other computers, such as Macintoshes, have abandoned them years ago).

- asynchronous — no clock is sent along with the data
- connects two nodes only — not a bus
- data is sent in only one direction along a wire (but usually there are two wires to give full duplex communication)
- can also send data over a telephone circuit with a modem, or using infra-red (IR), radio or many other media
- ancient system, full of historical artifacts

Data are transmitted in small, separate packets, which I'll take to be bytes (8 bits) – almost universal now

- technique originated in teleprinters, which used only 5 bits in the distant past, Baudot code
- sometimes a 9th parity bit is sent (but rarely these days)

Form of data

The transmitter sends the logical value of each bit for a set period, so the core of a packet looks like this.

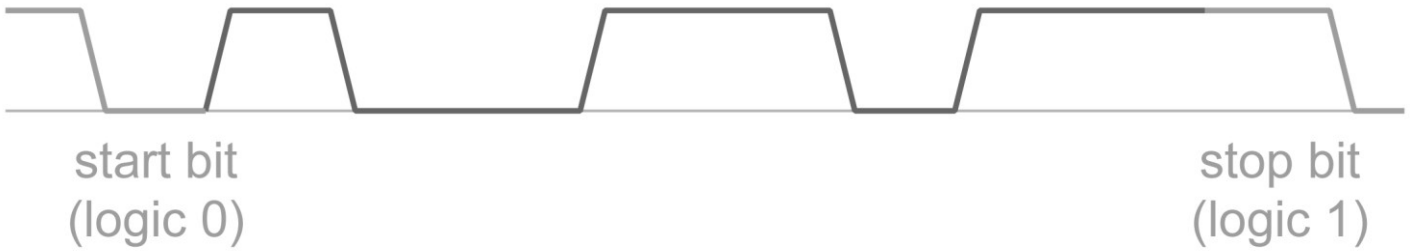
The receiver has to decode this reliably, so it needs to know

- the duration of each bit in time (reciprocal of baud rate)
- when each byte starts
- would be helpful to be able to check when each byte finishes

Each bit is transmitted for a certain length of time and the reciprocal of this is a frequency called the **baud rate**. For historical reasons the rate is usually 300 times a power of 2, often 9600 or 19200. (Was originally 110 baud – slow!) The transmitter and receiver must be configured manually for the same rate — you may have seen this on PCs.

Complete packet

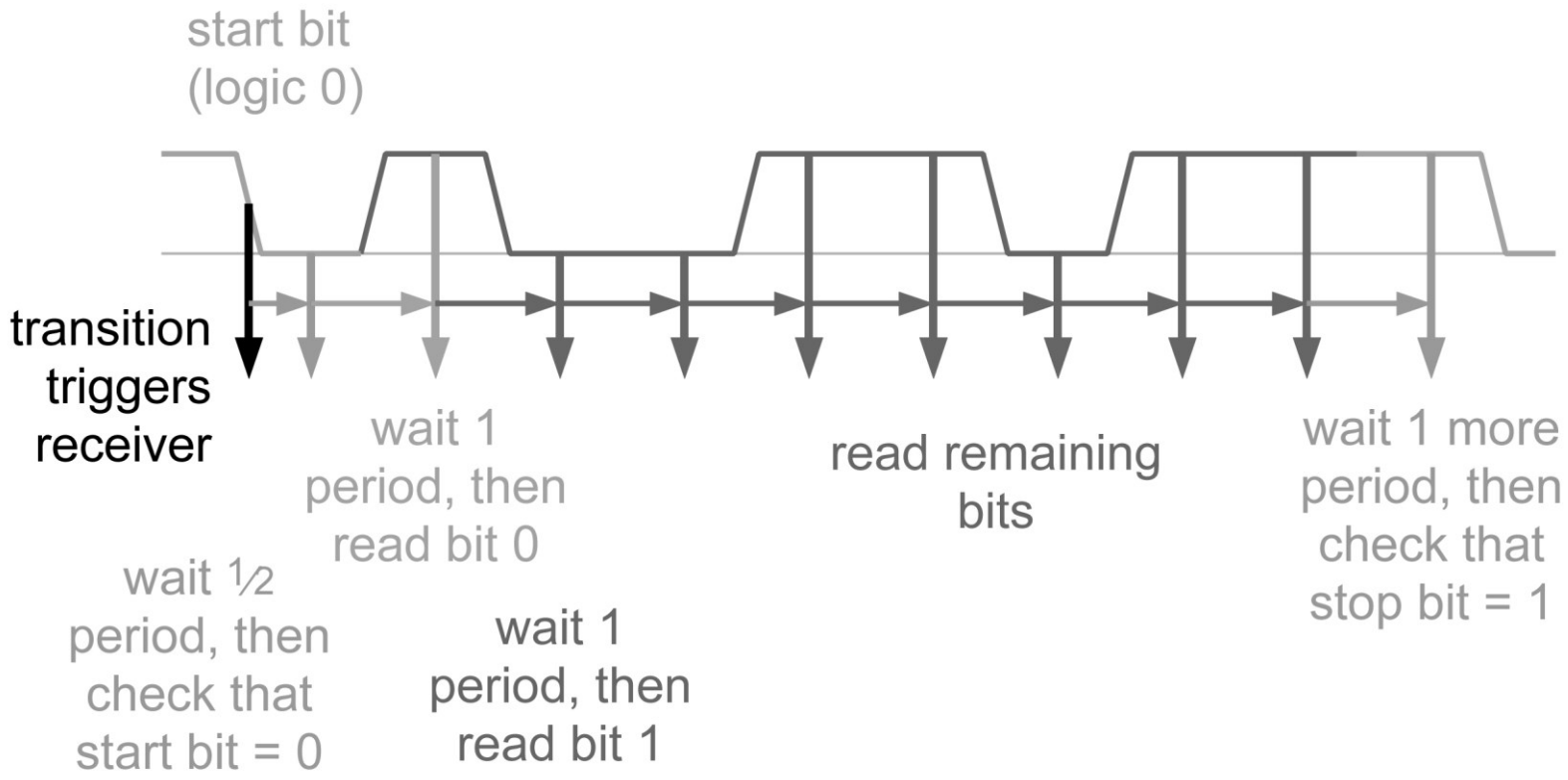
Start and stop bits are added to the ends of the packet to set the timing.



Rules:

- the line is held at logic 1 when it is idle
- a **transition from 1 to 0** signals the start of a byte and triggers the receiver
- the 0 is held for the usual period to form the **start bit**
- the **8 bits of data** are then transmitted
- finally, a **stop bit** of 1 is sent for the usual duration
- this may be followed by the start bit of the next byte

How the receiver decodes the packet



Disaster will strike if the receiver uses a different period from the transmitter. An error of about 4% is tolerable.

The receiver then re-synchronizes on the next start transition.

This looks like hard work

Asynchronous communication is hard work, particularly for the receiver. It can be done in software (bit-banging) but will occupy the µC exclusively, even for relatively slow baud rates.

PBASIC and other modules have **serin** and **serout** commands.

As for I^2C, many µCs have built-in interfaces to handle serial communication. They go by various names:

- **serial communication interface** or controller (**SCI** or **SCC**)
- **universal asynchronous receiver transmitter** (**UART**) — it may also handle synchronous signals (**USART**)

Again there are special function registers that must be configured with the baud rate, whether asynchronous or synchronous, and so on.
The interface will then handle the detail of sending or receiving each byte and inform the main program when the task is completed.

How is a serial interface implemented?

This is when it starts to get sticky! What are the allowed voltages for logic 0 and 1, for instance? Usually no problem between components within the same system, but what about communication with a PC?

The oldest standard, still used on PC serial ports, is called **RS-232C** (**EIA-232**). It specifies voltages of

+5 to +25 V for logic 0 (around +12 V is often used)
–5 to –25 V for logic 1 (around –12 V is often used)

This goes back to the very early days of computers, teletypes and modems.

- Special driver chips, such as the MAX232, generate these voltages (both positive and negative) from a standard +5 V or +3 V supply.

As well as the data lines (two — one for each direction) there is a bunch of control lines to supervise the flow of data. These are virtually obsolete.

Serial ports are being replaced by **Universal Serial Bus, USB**. This is much faster and also provides (limited) power via the bus — avoids the need for power supplies in every component.

Conclusions

Seen wide range of requirements and approaches used for communication with µCs

Looked in detail at two widely used methods

- I^2C — synchronous bus
- RS-232 — asynchronous (and not a bus)

Can be implemented in different ways

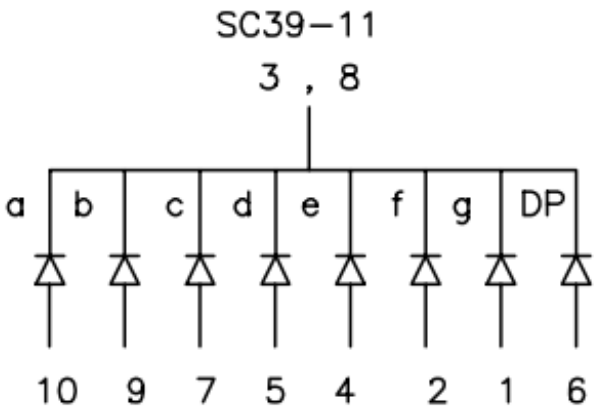
- purely in software, using general purpose I/O — bit-banging
- specialised interfaces — widely available in µCs for common protocols
- good idea to use interface chips when ‘high’ voltages are required, as in RS-232

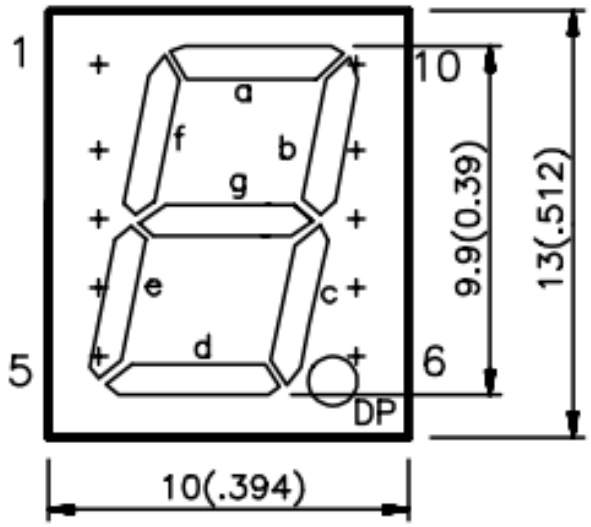
Lab hints: Seven Segment Display (1)

This section is about **lab 5: Seven Segment Display.**

This lab is based on **SC39-11 Kingbright® Single Digit Numeric Display**.

In principle, the **Seven Segment Display** is nothing more than **an arrangement of LEDs**. You can find the Datasheet of SC39-11 at the end of the lab guide. Notice the circuit diagram:



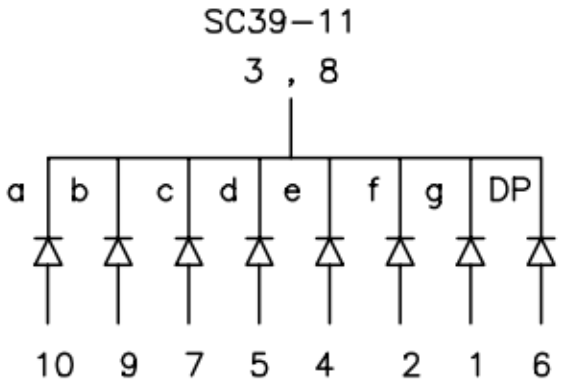


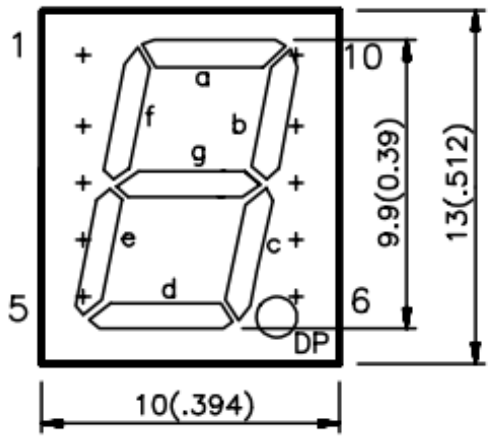
Lab hints: Seven Segment Display (2)

From the circuit diagram: If we connect either **3 or 8** to the **ground**, and **+5 V across a resistor** at point **1**, segment **g** will light up.

Now, if you want a particular digit or letter, you can calculate the needed combination of segments **a,b,c,d,e,f** and **g** to construct these symbols. For example, we can make capital **F** from segments **a, e, f** and **g**.

Remember to have a resistor in the circuit, as otherwise big current can damage your SC39-11.





Seven Segment Display: template (1)

In this lab, you are given a template that contains function
`void sevensegment(uint8_t value)`

This function will display the digit you give to it on SC39-11 provided your **1) electrical connection is correct** and **2) digit is between 0 and 9** (only digits 1 and 7 are implemented, you have to complete the function).

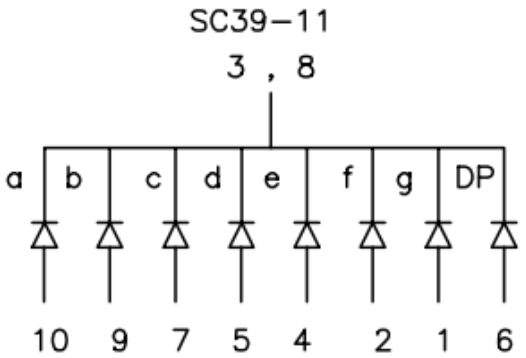
From this function, you can see that particular LEDs are expected on particular pins, for example LED “a” must be on PC2:

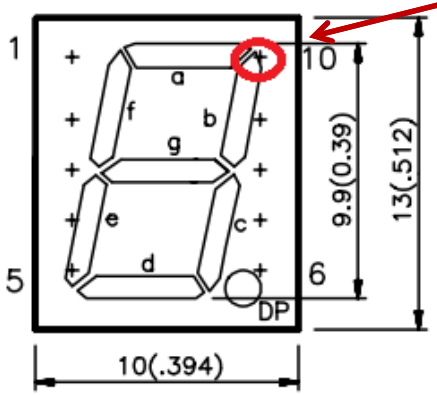
```
case(7):
        PORTC |= (1 << PC2); // A
        PORTC |= (1 << PC1); // B
        PORTC |= (1 << PC0); // C
        //PORTB |= (1 << PB0); // D
        //PORTB |= (1 << PB1); // E
        //PORTB |= (1 << PB2); // F
        //PORTB |= (1 << PB3); // G
break;
```

Seven Segment Display: template (2)

Now, 3 things must match: pin of microcontroller, letter code of an LED and location of the connection.
To properly connect **LED "a":**

- From the circuit on the left, to connect to **LED "a"**, you need to connect at location **10**.
- From the **diagram in the middle**, you can see where **10** is. It is **highlighted**.
- Lastly, **from the code** you see that you need to connect it to **PC2**.





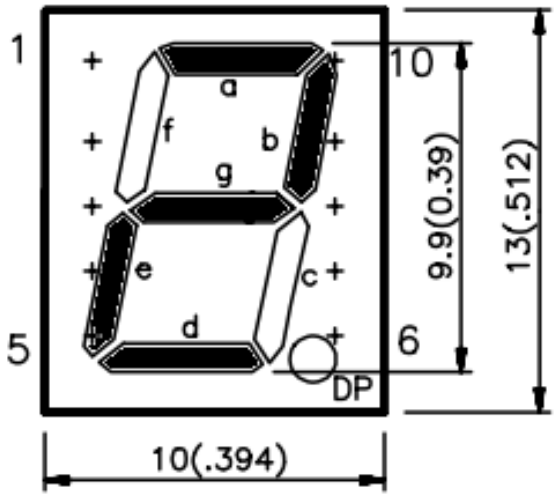

```
case(7):
        PORTC |= (1 << PC2); // A
        PORTC |= (1 << PC1); // B
        PORTC |= (1 << PC0); // C
        //PORTB |= (1 << PB0); // D
        //PORTB |= (1 << PB1); // E
        //PORTB |= (1 << PB2); // F
        //PORTB |= (1 << PB3); // G
break;
```

Seven Segment Display: template (3)

Repeat the procedure to find right connections for all 7 LEDs (Segments).

Now it is time to finish the code. You will need to copy the code for one of implemented digits, and change it. When a line corresponding to some LED is commented in this code, this LED is off. When you let the line execute (not commenting it), the LED will be on.

Here is a sample solution for number 2 (Segments A, B, G, E, D):




```
case(2):
        PORTC |= (1 << PC2); // A
        PORTC |= (1 << PC1); // B
        //PORTC |= (1 << PC0); // C
        PORTB |= (1 << PB0); // D
        PORTB |= (1 << PB1); // E
        //PORTB |= (1 << PB2); // F
        PORTB |= (1 << PB3); // G
break;
```

Seven Segment Display: completing the lab

Repeat this procedure for all other digits (**0,3,4,5,6,8,9**). Now you are ready to display any digit. Test it by changing the digit given in the main function.

To pass the lab, you need to have a circuit with AVR Board and SC39-11 which will **increase the current digit by 1 on every press of button 1**, and **decrease it on every press of button 2**.

To react to individual key presses, use **debouncing** (Lecture 1) and empty loop:

```
if(Button is pressed)
    Increment or decrement the value;
   while(Button is pressed)
       DO NOTHING
   Delay(50ms) //(Debouncing – happens on button release)
```

Microcontroller

Lecture 05: Driving External Loads

HOCHSCHULE RHEIN-WAAL
Rhine-Waal University of Applied Sciences

Driving external loads

How do we drive a load bigger than a LED?

- bipolar transistors, MOSFETs and H-bridges
- protection from inductive loads

Continuous variation of output

- digital–analog converters (DACs) — rare in microcontrollers
- Pulse Width Modulation (PWM) — covered already in Power Electronics 2
  - widely used for motors, heaters, and other loads that respond slowly
  - most microcontrollers have timer(s) to provide PWM outputs automatically, without intervention from the main processor

Remote control servos

Stepper motor

- widely used for precise, repeatable positioning

Driving a load

Our PIC can easily drive an LED or two directly but its maximum current is 25 mA from each pin with a 5 V supply, dropping to only 15 mA at 3 V. This is fairly typical of modern microcontrollers.

How can we drive

- loads that need more current?
- loads that need a higher voltage than the microcontroller?

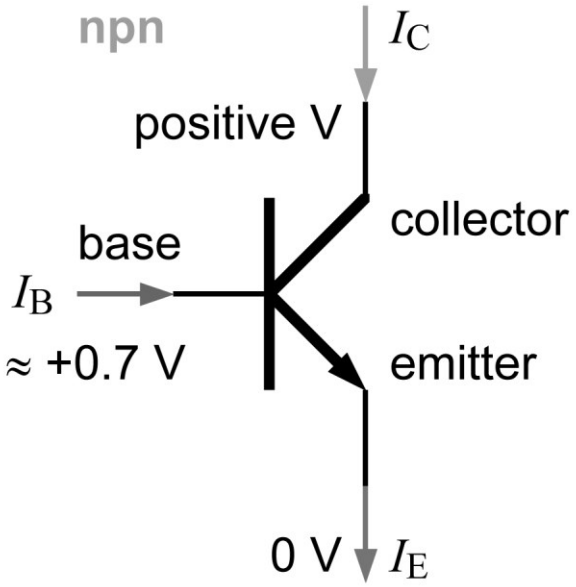
As usual there is a wide range of solutions from single transistors to ICs for common requirements. Let's look briefly at simple solutions:

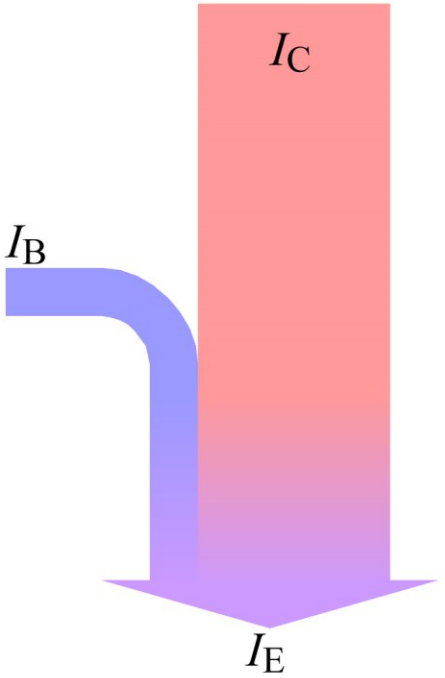
- single bipolar junction transistor (BJT)
- Darlington pair
- metal–oxide–silicon field-effect transistor (MOSFET)

In all cases, the aim is **to switch the load fully on or fully off** — not to vary its power continuously. We'll look at continuous variation later.

Bipolar junction transistors

These are often just called ‘transistors’. They come in two polarities, npn and pnp. The simplest picture is that a small current into the base causes a much larger current to flow into the collector.

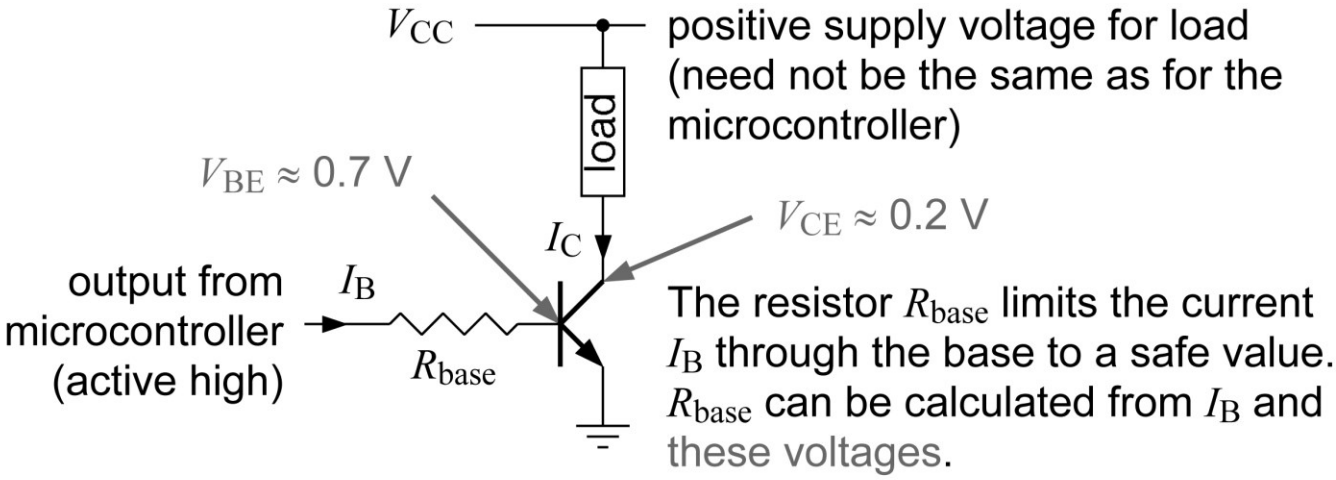




The simplest equation to describe the operation is $I_C = \beta I_B$, where $\beta \approx 50$ for a medium power transistor. The currents and polarities are reversed for a pnp transistor.

Bipolar junction transistor with a load

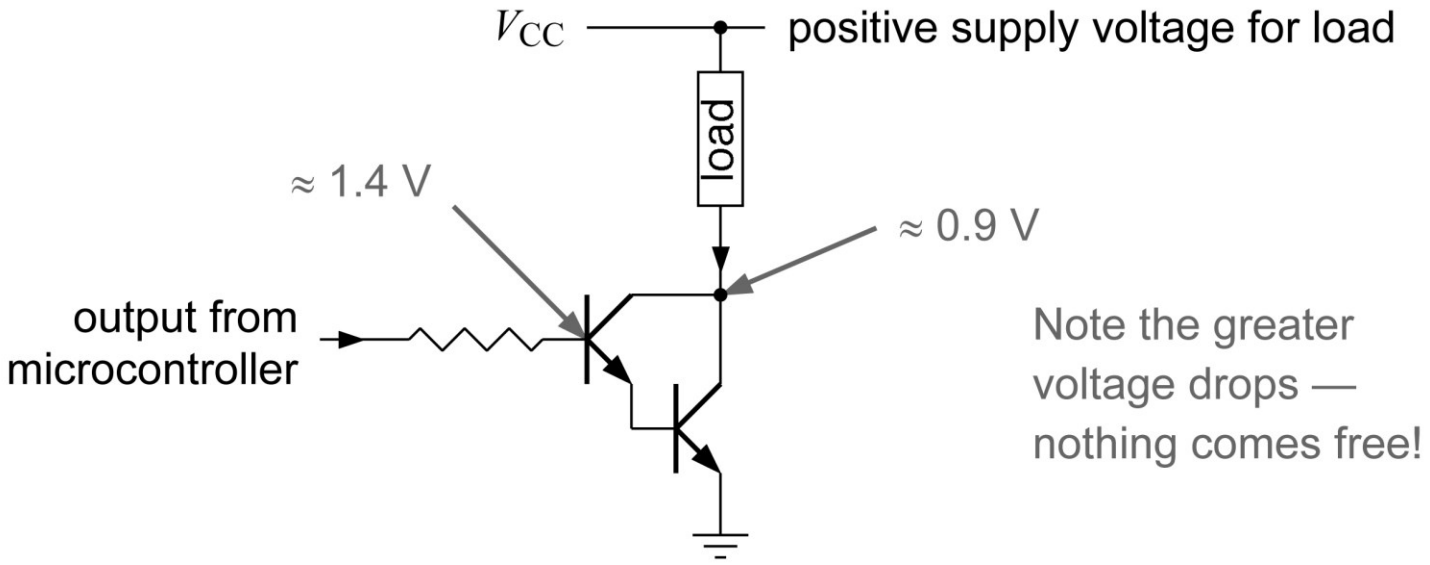
Here is the simplest circuit for driving a moderate load — up to roughly 500 mA — with a single transistor.
When the output from the μC goes low (0 V), the transistor is switched off and very little current flows through the load.
The transistor turns on when the output goes high.



The collector of the transistor is at about +0.2 V and the base is at about +0.7 V when the transistor is switched on (high output from μC).

Heavier loads — Darlington pair

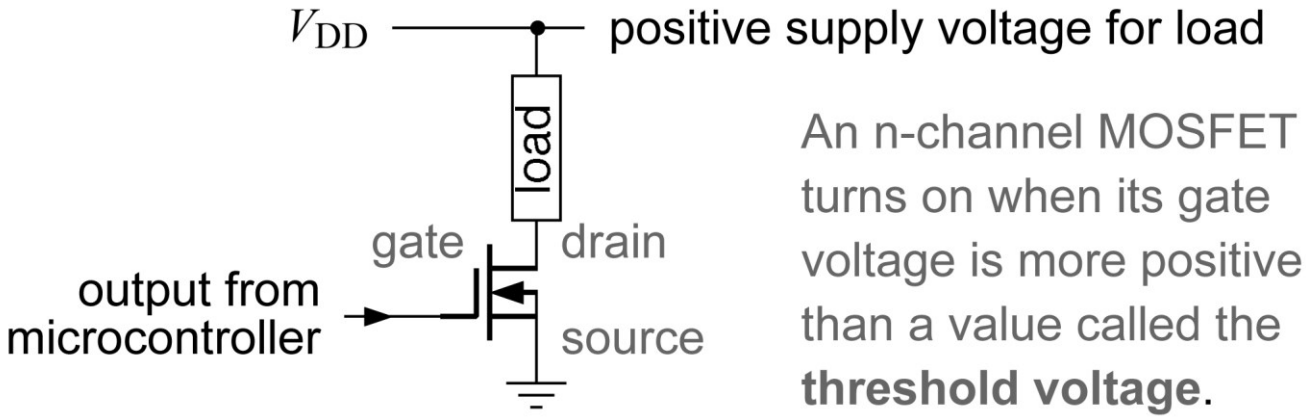
The base current becomes too large to be supplied by the microcontroller for currents greater than about 500 mA. Solution: two transistors connected as a **Darlington pair**.
This is used so commonly that special Darlington devices are available. Often there is more than one load, and ICs provide an array of Darlington drivers.



You may wish to seek expert advice from Mr. Grunenberg.

MOSFET with a load

MOSFETs are the second common class of transistor. Again they come in two polarities, n-channel (equivalent to npn) and p-channel.
These are the basic devices used to construct digital integrated circuits, such as microcontrollers, and bigger ones can be used to switch loads in the same way as bipolar transistors.



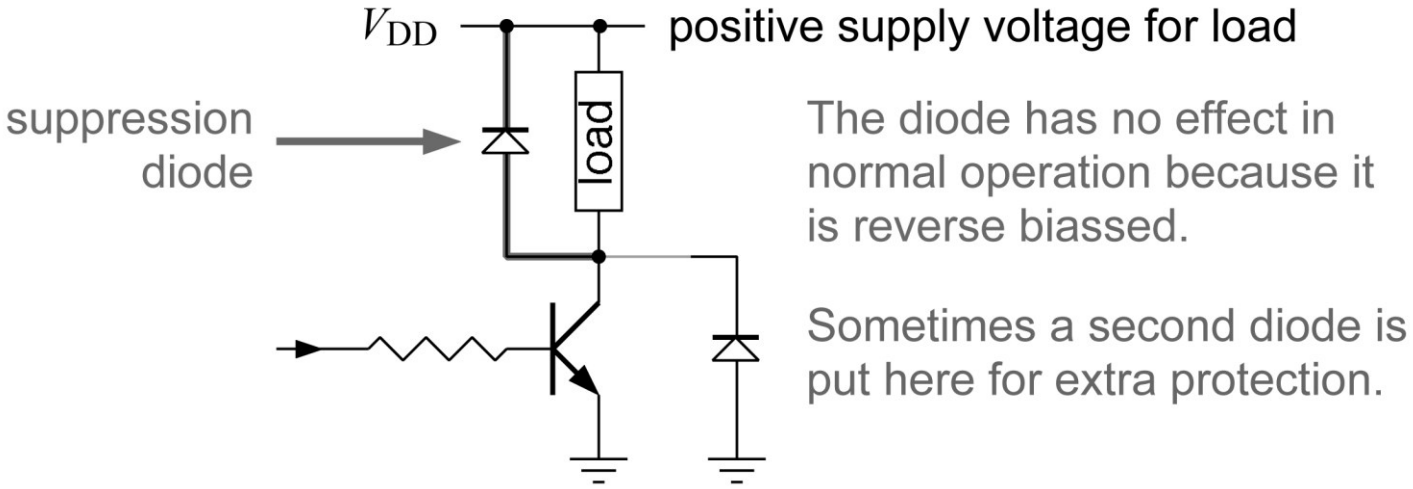
An n-channel MOSFET turns on when its gate voltage is more positive than a value called the **threshold voltage**.

MOSFETs are controlled by the voltage on the gate rather than the current: the gate is essentially one plate of a capacitor. Thus no current flows in a steady state, only when the voltage changes. This makes them very attractive devices to use, and a wide range of devices and ICs is available.

Watch out for inductive loads!

Many loads are **inductive** — motors, transformers, relay coils. These product a **back-emf** when they are turned off, a voltage of opposite sign to the supply voltage ($V = -L\mathrm{d}I/\mathrm{d}t$). **Without protection the back-emf may destroy the transistor and probably the microcontroller as well.**

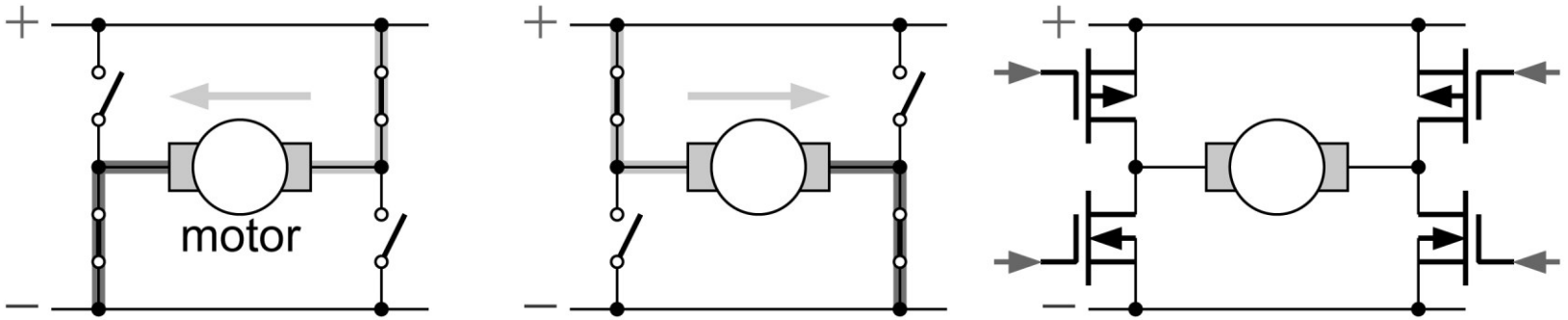
Solution: connect a diode to short out any reverse voltage across the load.



Protection diodes are built into most drivers, and sometimes the base resistor as well. These are sometimes called **digital transistors**.

What if I want a DC motor to go in both directions?

These circuits will provide only one direction of current through the load. The direction of rotation of DC motors can be changed by reversing the direction of current.
What we want is something like this (ignoring protection diodes etc).



This is called an **H-bridge** and is available as an integrated circuit or can be built from discrete devices.
The gates of the FETs (or bases of transistors) must be driven correctly to give the desired connections (and not short the power supply!). Special controller ICs do this and may provide modes for braking and sleep.

Continuous variation of output

All these approaches switch the load on or off — nothing in between. **How can we control the voltage or power in a load continuously?**
The obvious solution is a **digital–analog converter** (DAC). This is of course the opposite of an analog–digital converter: the input is a digital value and the output is an analog voltage:

output voltage = (digital input value / maximum value) × reference voltage

Similar issues arise as with ADCs: for example, there is little point in having a very precise DAC (lots of bits) unless the reference voltage is accurate and stable.
You may be surprised to hear that **few microcontrollers contain DACs** except in devices intended for particular applications, such as audio. There are only few in the whole range of PICs, for example. If you really need one, you will probably have to find a separate device and control it from your µC with I^2C or a similar interface.

Analog output — pulse width modulation (1)

Most microcontrollers provide only digital outputs (logic 0 or 1). Transistors are also much more efficient when acting as on–off switches, rather than producing intermediate outputs. They dissipate a lot of power when neither completely on nor off. **How can we control an output continuously by only switching it on and off?**

The common method of simulating a continuously variable output is **pulse width modulation** (PWM):

The duty cycle of a square wave is varied to change the average power delivered to a load.

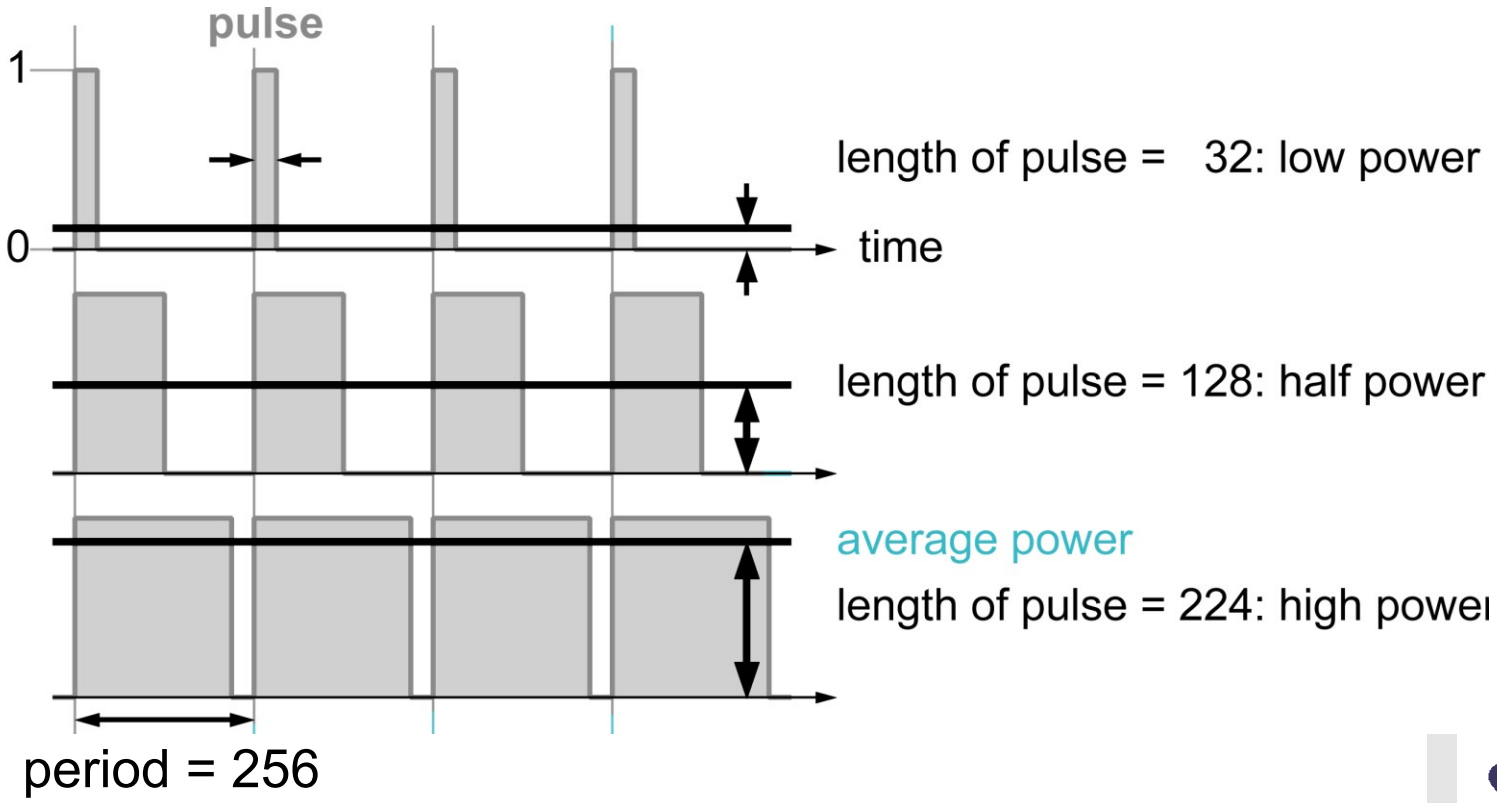
The frequency of the square wave must be high enough not to be noticeable.

- LEDs in demonstration are driven slowly so that PWM is visible!
- Many loads, such as heaters, response very slowly
- Inductive loads, such as motors, provide smoothing themselves
- A low-pass filter is needed for sensitive loads

Analog output — pulse width modulation (2)

For example,

- period of square wave = 256 units
- vary duration of pulse (output = 1) from 0 (fully off) to 256 (fully on)



How to generate PWM waveforms

This can be done in software using a loop with delays to switch the load on and off. For example, this will run an LED at 50% power.

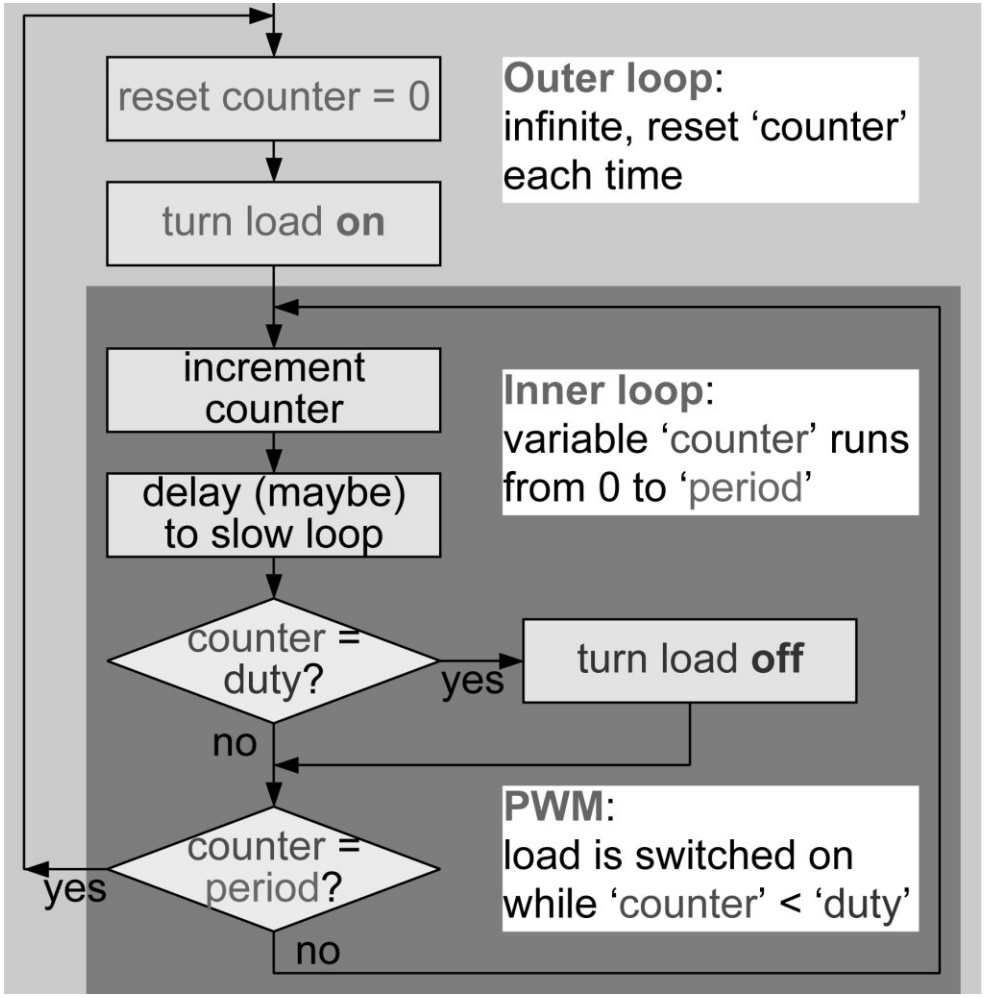
```
How to generate PWM waveforms
PWMLoop:
    high 0      ; turn on LED
    pause 1     ; wait 1 ms
    low 0       ; turn off LED
    pause 1     ; wait 1 ms
goto PWMLoop  ; back around infinite loop
```

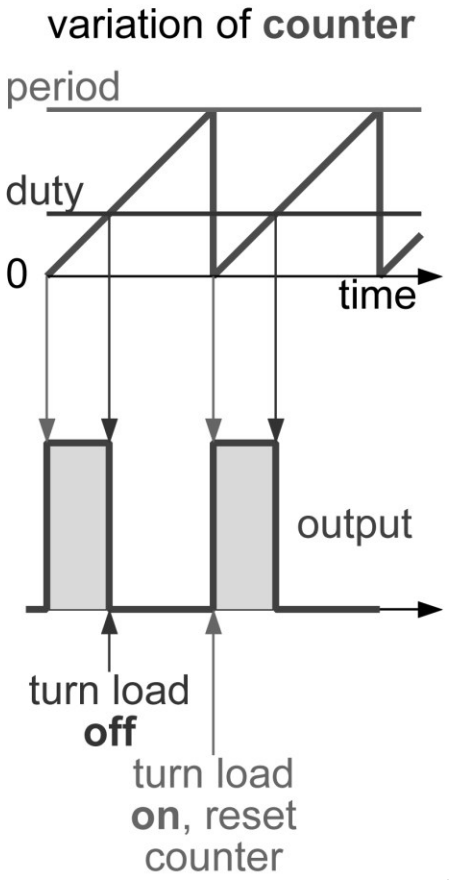
The average power could be varied by changing the two pauses, although you have to be careful about timings. For good control you really need finer resolution than 1 ms — it's too coarse. See **`pulsout`** later.

In general you might have a 'for–next' loop with a variable **`counter`**. This runs from **`0`** to **`period`** but the load is switched on only while **`counter`** lies between **`0`** and **`duty`**. The average power is then **`duty/period`**.

Some versions of PBASIC have a **`pwm`** instruction to do this automatically.

Generation of PWM waveforms in software





Generation of PWM waveforms in hardware

Many microcontrollers contain one or more timers to generate these waveforms: they run automatically after loading the period and length (duty) of the pulse. Look for a **capture/compare/PWM (CCP)** module in a PIC.
The 16F630 does *not* have a CCP module but the 16F627 does.
This works in exactly the same way as the software on the previous slides.
The module is based around a counter fed by the clock used by the rest of the microcontroller. This is equivalent to the variable **counter** in software.
Each time it is incremented, the counter is compared with two registers:

- when the counter reaches the value in the **duty** register, the load is switched off
- when the counter reaches the value in the **period** register, the counter is reset and the load is switched on (unless duty = 0)

The ratio of the values in the duty : period registers sets the duty cycle and hence the average power supplied to the load, just as in the software.
Some versions of PBASIC can set this up automatically: **`pwmout`**.

Perils of PWM

Because PWM is generated by a counter, there is a tradeoff between speed and accuracy.

- high accuracy means that ‘duty’ and ‘counter’ need a large range
- this means that ‘period’ must also be large and the loop will be slow

Example:

- Suppose that 256 possible values of the output are needed (8 bits).
- The period should therefore be 256 so that ‘duty’ can range from 0 to 255 (always off to always on).
- If the µC executes an instructions every µs, one period takes a minimum 256 µs ≈ ¼ ms, which corresponds to a frequency of 4 kHz.
- This is in hardware; in software, each iteration of the loop requires about 4 instruction cycles — even slower.

Care is needed when **`duty`** is updated to avoid glitches in the output. In hardware the CCP module should do this correctly for you.

Types of electric motors

Discussed in this lecture:

Classified by type of operation:

- Stepper motor
- Servo motor

Not discussed:

- Basic DC motor – continuous rotation:

Classified by commutation mechanism:

- Brushed motor
- Brushless motor

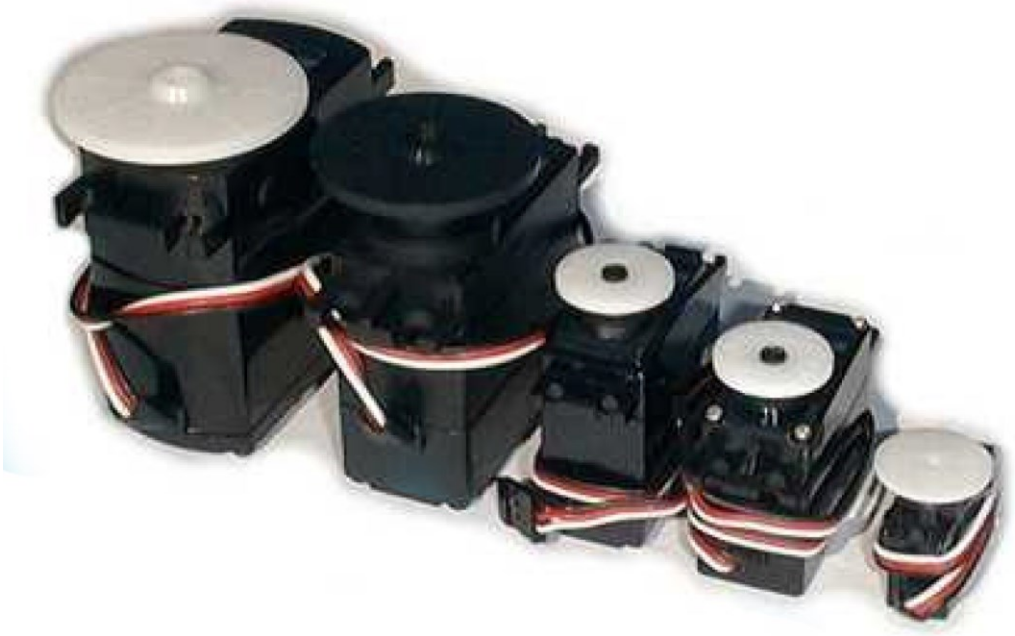
- Asynchronous / synchronous, other characteristics of motors.

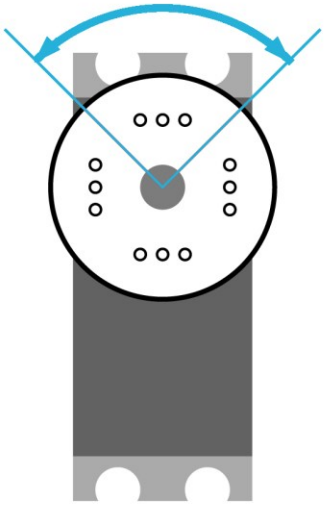
You will learn this information in Power Electronics with Prof. Schmetz.

You can find great explanation on basics online – eg. Youtube, "Learn Engineering" channel, https://www.youtube.com/watch?v=Vk2jDXxZIhs

Remote control servos

These are often used to control model aircraft, cars and the like. They do not turn continuously like a motor, but rotate their output through a range of 90°. They are controlled by sending a stream of pulses at 50 Hz.

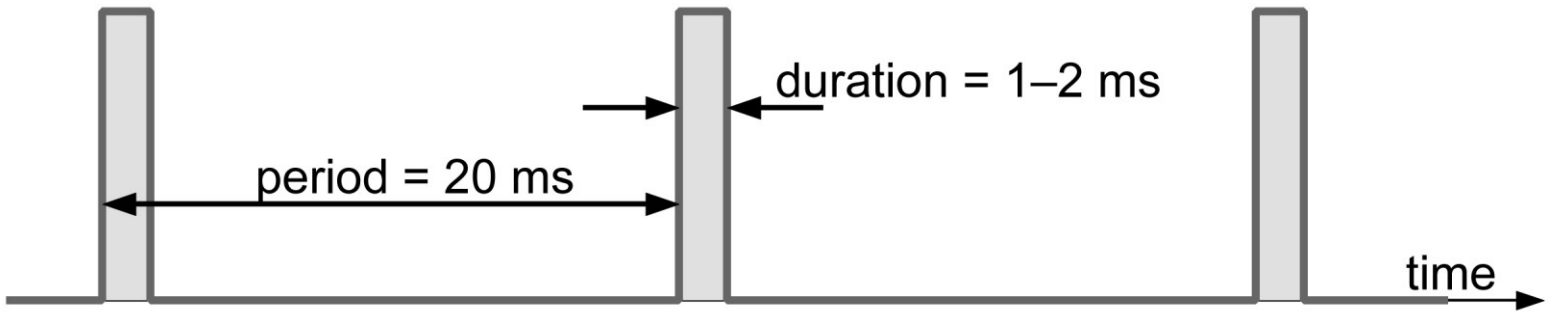




http://www.milinst.com/animatronics/animatronics.htm

Control of servos

Servos are controlled by a stream of pulses (roughly to scale):



The position of the servo depends on the length of the pulse, for example:

- 1.0 ms — far left (–45°)
- 1.5 ms — centred (0°)
- 2.0 ms — far right (+45°)

The **`pause 20`** instruction can be used to give the interval between pulses but is too coarse for the pulses themselves because the duration is in ms (milliseconds). Instead, use **`pulsout pin,time`**, where **`time`** is in units of 10 µs = 0.01 ms.
Some versions of PBASIC have a **`servo`** command.

Stepper motors

These are widely used where **precise, repeatable, open-loop control of position** is required — moving the head in an ink-jet printer is a typical application.

The rotor does not move continuously, as in a traditional DC motor, but in a sequence of steps controlled by pulses applied to the windings on the stator of the motor. Either the number of steps or the angle moved in each step is specified. A typical value is 7.5° or 48 steps. This discrete response goes naturally with a digital system.
The rotor may be either (or sometimes both):

- a cylindrical magnet, with alternate N and S poles around its circumference
- a shaped piece of soft iron (easily magnetized), which is attracted to the coils on the stator when they are active — reluctance motor

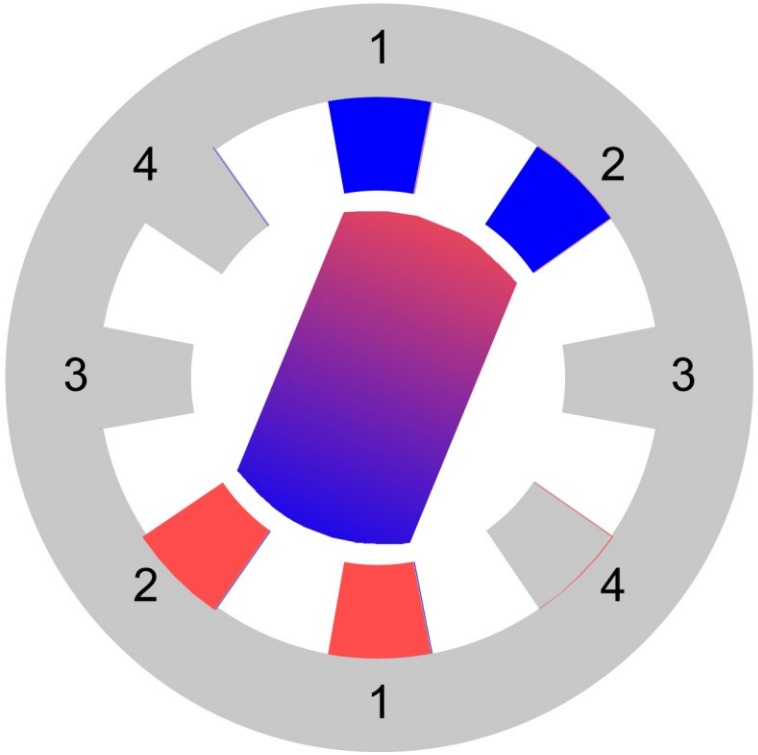
There are commonly 4 sets of coils, with 6 or more coils in each set.

Simple stepper motor

This is a highly simplified motor with 4 sets of coils (typical) but only 2 coils in each set, 8 in total (24 or 48 would be more usual). The projecting (salient) poles have coils wound on them, which are not shown. Instead, these poles are coloured red and blue to show when they are active.

Rotor moves through 45° in each step

(step would be smaller with more sets of coils)



Idle

Coils 1 and 2

Coils 2 and 3

Coils 3 and 4

Coils 4 and 1

Coils 1 and 2 (but polarity reversed)

Stepper motors

Stepper motors can also be operated in other modes.

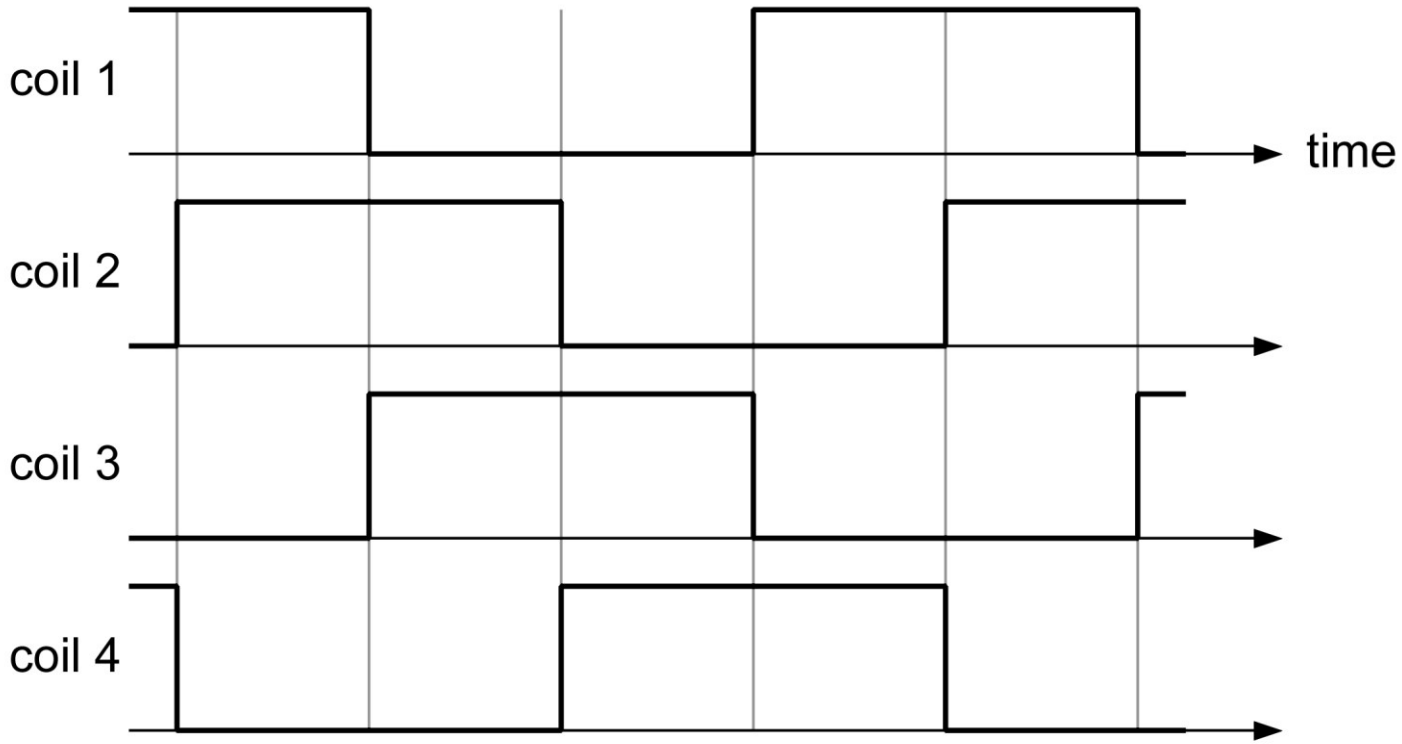
- Only one coil active at each time — reduced current but also reduced torque.
- Half-step — for example, a sequence of (1+2), 2, (2+3), 3, …. The rotor moves through only half the angle each time, 22.5° in the example shown in the sketches. A disadvantage is that the torque is different when 1 or 2 coils are active.
- The system in the sketch needs a bipolar supply — the coils must be energized in both directions. Other configurations need only a unipolar upply.

Steppers cannot be operated too quickly because it takes time for the motor and load to accelerate and decelerate.

They are also prone to mechanical resonance and these frequencies must be avoided!

Waveforms to drive a stepper motor

The following waveforms are needed to drive the stepper motor, assuming that it is unipolar rather than bipolar as in the sketches.



The motor can be driven in the opposite direction by changing the waveforms — how?

Control of stepper motors

There are three main aspects of driving a stepper motor:

- generation of clock frequency
- generation of waveforms to provide desired movement (translator)
- power electronics to drive motor (Darlington drivers, or H-bridge if bipolar supply needed — with protection diodes!)

As usual, specialized ICs are available to assist this task. For example, the interface between a translator and a microcontroller might require:

- clock waveform to define the frequency
- stop/go
- forward/backward
- half-step/full step

The translator IC would generate the waveforms required to drive the motor. A second IC might be needed to drive the motor itself or these functions might be combined.

A bit more on motors…

The usual electronic distributors include stepper motors in their catalogues:

- http://de.rs-online.com/web/
- http://de.farnell.com/

Here is a book that I have found useful. It includes a chapter on ‘Interfacing to external devices’ :

Fast and effective embedded systems design
Rob Toulson and Tim Wilmshurst
Newnes, ISBN 978-0-08-0977768-3 (00/TWI 3)

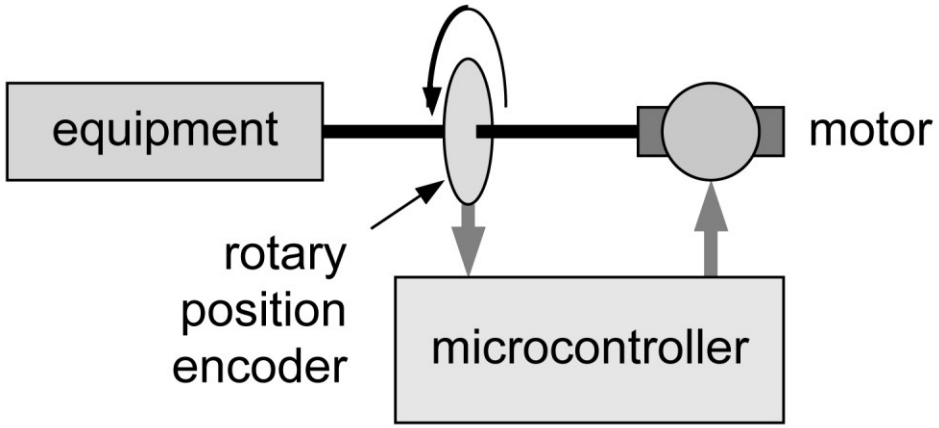
A couple of manufacturers’ web sites:

- www.thomsonlinear.com
- www.je-motion.com

Both have plenty of information.

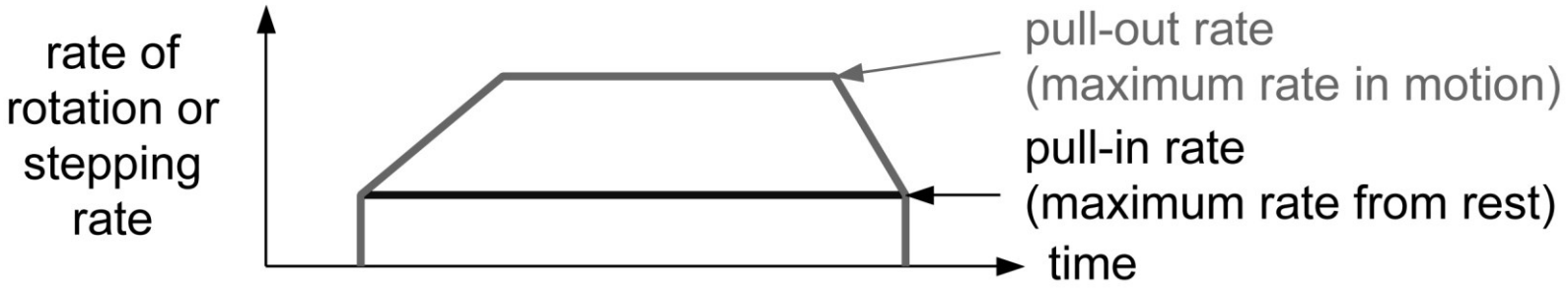
Steppers are not the only motors

Stepper motors are very convenient because of their natural ‘digital’ action but they are not the only type of motors available — far from it.
However, most other types of motor will require an additional sensor for position. Optically encoded disks are often used. The microcontroller must monitor this and drive the motor so that the system ends up in the desired position. (This is part of Control theory.)
You might find systems that include both the motor and its control.



Points to watch

It is not easy to get very small stepper motors because their construction is complicated. Simple DC motors can be made smaller (in model vehicles for instance).
Remember that most motors draw a relatively large current: make sure that the drivers and power supply can handle the load. Stepper motors draw a higher current when they are holding rather than rotating (no back-emf).
If you want to run them fast, stepper motors must be accelerated gradually because of the limited torque available. This is controlled by the stepping frequency, which must therefore be variable.
See the Airpax catalogue, which has a helpful introduction.



Conclusions

Seen how to control simple loads from a microcontroller
- much better to switch load on or off — not part way
- can use bipolar transistor or MOSFETs
- protection needed from inductive loads

Pulse width modulation (PWM) used as substitute for continuous variation of output
- can generate ‘in background’ using timer module(s) in most microcontrollers

Remote control servos

Stepper motors
- straightforward method of getting precise, repeatable positioning
- control through sequence of pulses
- special ICs available to carry out details (as usual!)
- may need ‘reset’ sequence to fix absolute position

Practicals:
Introduction to EEPROM

As you may already know, many devices need and use external memory that persists without power in the device.

Unfortunately, it is not a regular part of AVR laboratory board equipped with ATMega88pa…

Fortunately, we can connect an external one using TWI – two wire interface, a.k.a. I2C Bus.

Lets look into how it is done.

I2C – Integrated Circuit to Integrated Circuit communication

It is called TWI for a good reason – it consists of two wires, which work as buses and go through all the devices connected (see also our last lecture).

One is Clock, SCL
Another transmits information, SDA

Important features:

- **Devices only drive signal to Low, and if they don't (tristate, high impedance), signal is High due to pull up.**

- **Only one device can send data at a moment. To facilitate communication, some device must be a master and define order of interaction over I2C, and other devices are called slaves.**

Template functions

As you can see in i2c_master.c and i2c_master.h for most of I2C tasks there are functions prepared:

- **master_open_write(uint8_t adr); //open slave „adr“ for writing**
- **master_open_read (uint8_t adr); //open slave „adr“ for reading**
- **master_close(); //close current connection. Use after writing complete.**

- **master_write(uint8_t oneByte); //write oneByte to the active slave**
- **master_open_read_next(); //return next Byte, use if you will need to read more**
- **master_open_read_last();//read last Byte, use to end reading.**

Completing templates

However, some parts of code are incomplete to let you to complete those:

- **1) master_close(); //close current connection. Use after writing is complete.**
- **2) master_open_read_last(); //read last Byte, use to end reading.**
- **3) Main() function is not complete.**
  - Reading potentiometer value from ADC in main.c

ST24C02 EEPROM (1)

- **2 kilobit long = 2048 bits = 256 bytes**
- **Therefore, 256 memory addresses, 8 bit address size is enough.**

Supports TWI; To write:

1) Start communication with this device(1010 + 3 unique bits + 0 for write, 10100000 to 10101110 depending on jumpers)
"master_open_write (@decimal for 1010xxx0@); "

2) First byte send will be understood as address of the first byte of memory:
"master_write(123);" //will set start write position to memory byte #123.

3)Write as many bytes as you need.
"master_write(myByte);" //next time you use it, write operation chooses next memory slot automatically.

4)Close the connection.

** instructions are correct only for 4 byte writing or less. Refer to ST24C02 docs for more info.*

ST24C02 EEPROM (2)

To read:

1) Use prev. instruction to write one byte to the device: memory address to read. Close the writing afterwards.

2) Start read communication with this device(1010 + 3 unique bits + 1 for read, 10100001 to 10101111 depending on jumpers)
"master_open_read (0b1010xxx1); "

2) As long as you will need to read more:
"myByte = master_read_next();" //will set start write position to memory byte #123.

3)When you read last byte, use
"myLastByte = master_read_last();"

To complete read_last and close functions: understanding TWI on ATMega88pa

As described in the manual, ATMega88pa has following registers to work with TWI:

TWBR – determines the frequency, and therefore speed of TWI
TWCR – the control register
TWSR – the status register
TWDR – data register, contains last byte to read OR byte to send
TWAR – slave address register, however we play as Master ☺

Lets examine bits of the control register!!!

TWCR – the control register

Lets examine bits of the control register. (p.230-231 in datasheet)

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
TWINT	*TWEA*	*TWSTA*	*TWSTO*	*TWWC*	*TWEN*		*TWIE*

0 TWIE	Interrupt Enable. We will not be using interrupt.
2 TWEN	TW Enabled. Set to zero to terminate any ongoing operations, set to 1 to use TWI.
3 TWWC	Write Collision flag.
4 TWSTO	Causes AtMega88pa to send stop signal in current communication. TWSTO bit will be cleared upon completion.
5 TWSTA	If start is written to 1, device tries to become Master in current network
6 TWEA	Enable Acknowledgement: in the end of current operation acknowledgement bit will be sent. If TWEA is set to 0, device is virtually disconnected, it does not react.
7 TWINT	Calls the interrupt. Is written to 1 automatically when current task is done! CHECK FOR IT TO KNOW WHEN CURRENT TASK IS DONE

Completing templates (1)

So following should be done in control register:

- **master_close(): we need to write TWINT bit and TWSTO bit – TWINT says there is an operation to be done over TWI, TWSTO says to send STOP signal. To check that operation is complete, remember that TWSTO bit is cleared automatically when STOP signal is successfully sent.**

Completing templates (2)

So following should be done in control register:

- **master_open_read_last(): When we do not need to keep reading, we simply skip sending acknowledgement bit. See TWCR.**

- **Storing 16 bit analog value in memory: you will have to write separately two bytes of info. You can either use high and low parts of ADC value, or use /256 and %256 to get them.**
  Mind the sequence of this bits when reading!

Optional task: store many values in memory

Following algorithm can be used:
Button 1 >> @choose number of variable to store@ (let user increment the value of variable by pressing button 2, display current choice and instruction on LCD. Let user press button 1 again to confirm his choice.

Button 2 >> @choose number of variable to load@ (let user increment the value of variable by pressing button 1, display current choice and instruction on LCD. Let user press button 2 again to confirm his choice.

To save and load specific variable, use separate memory addresses. Remember you need two bytes for one value!

Microcontroller

Lecture 06: Power and other practical matters

HOCHSCHULE RHEIN-WAAL
Rhine-Waal University of Applied Sciences

Power and other practical matters

What power source should you use for your system?

- mains or battery, or something exotic?

Mains supplies

- need regulation, filtering and perhaps backup

Battery supplies

- wide range of different types of battery
- need to allow for discharge characteristics
- how can we get away with a single cell?

Other issues

- watchdog timer
- sleep mode and how to wake up
- other approaches to reducing power
- example of power consumption

Practicals: Introduction to RTC (Real Time Clock)

Sources of power

In many cases the need for portability (or not) will guide the choice of power source between the **mains** and **batteries**.

However, there are other possibilities: photovoltaic (solar) cells, wind power, fuel cells (becoming practical).

You first need to decide on the voltage used by your circuit. In the past, logic ran at 5 V. This was a nuisance if batteries were used because you either had to use a regulator with 4 cells (nominally 6 V) or a single 9 V battery, much of whose energy was wasted.

Modern logic often works happily from 2 cells, which provide a nominal 3 V — but it's not as simple as that!

A higher voltage may be needed for some parts of a system, such as LCD displays and headphone amplifiers. Much more power is needed for motors or other mechanical parts.

Mains power supply units

Your system probably works at 3 V or 5 V DC, but the mains supply is 230 V at 50 Hz in Europe, 120 V at 60 Hz in North America. You therefore need a power supply unit (PSU). There is a huge choice.

'Plug in the wall' supplies, can now be bought very cheaply (€3 and up).
They may be very poorly regulated,
meaning that the output voltage depends strongly on the load. You must therefore provide a regulator so that your system sees a steady 3 V or 5 V. There are plenty of integrated circuits to do this — follow the data sheet.

PSUs come in two major varieties.

Linear — the classical approach with transformers, rectifiers and smoothing capacitors. Electrically quiet but heavy, bulky and inefficient.

Switching — use power electronics, which gives a much smaller, lighter and more efficient product. However, they produce a lot of electrical noise. They are universal in computers and increasingly common in other products, such as battery chargers.

Features of mains supplies

Readily available in buildings — but limits portability, even indoors.

Cost — power from the mains is virtually free (compared with batteries): less than 30cent per kWh (even allowing for loss in the PSU).

Generally reliable — doesn't run out like batteries, but power cuts are not unknown, so you may need a backup supply. You may also wish to retain data or keep clock/calendar chips running when the equipment is switched off (as in computers).

Noisy — transient pulses are produced by heavy equipment switching on and off (especially in the laboratories, where they cause laptops to crash!). The PSU must filter these out. In more extreme cases, lightning strikes or major failures of the distribution system can cause surges.

Potential danger from high voltage
Will there be problems if the equipment is used in another country?

Features of battery supplies

Portable — but may be heavy, so use as few as possible.

Cost — very expensive! An alkaline AA battery costs about €0.75, produces 1.5V and its rate capacity is 2500 mAh. What would it cost to produce 1 kWh of energy with these cells?

Finite lifetime — run down and need to be replaced. How easy is it to purchase a new battery?

Characteristics change during lifetime — depends on type of battery: more of this later.

Little noise — no transients or switching noise from the battery itself (but you may get noise within your system).

Usually needs protection against reverse polarity — if user can insert batteries the wrong way round (seek a battery holder that prevents this).

Bad for the environment — especially if not recharged or recycled.

Choice of battery

There is a huge range of batteries available. They fall into two classes:

primary — zinc chloride, alkaline (manganese), lithium (lots of different types), silver oxide, zinc air, …

secondary (rechargeable) — nickel cadmium (NiCd), nickel metal hydride (NiMH), lead acid, Lion, …

How should you choose one?

- What capacity is needed?
- How portable does the product need to be?
- **Choose a battery that is easy to replace:**

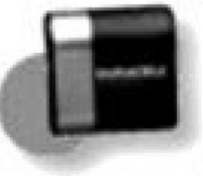
_alkaline batteries can be bought at any corner shop

_rechargeable NiCd and NiMH are also easy to buy, as are chargers

_some lithium coin cells and zinc air cells are fairly easy to find but exotic batteries are a nuisance, even if they have wonderful electrical properties!

Comparison of common batteries

Alkaline–MnO_2 — Popular, multi-use premium battery; good low temperature and high rate performance; good shelf life; sloping discharge characteristics.



Lithium–MnO_2 — High energy density; excellent rate capability and low temperature performance; excellent shelf life; relatively flat discharge characteristic.

Zinc Air — Highest energy density on continuous discharge; excellent shelf life (inactivated); moderate rate capability; limited shelf life when activated; flat discharge characteristic.

Silver Oxide — High gravimetric capacity; good shelf life; flat discharge characteristic.

Nickel–Metal Hydride — Rechargeable; high energy density; flat discharge; very high rate performance; fast charge capability.

from: www.duracell.com

Primary cells

Alkaline manganese are the ‘alkaline’ cells that you can buy anywhere. Their capacity is much greater than the cheaper ‘zinc chloride’ or ‘zinc carbon’ cells at large current drains; not so much for low currents. Nominally 1.5 V but with a strongly sloping discharge characteristic — see later.

There are many, many types of **lithium** cell, including the coin cells used in the desktop PC mainboards (dice). Usually around 3.0 V (depends on the chemistry used). Often used in cameras and for backup in computers — 3 V is convenient! A few sizes are readily obtainable. Small button cells are usually based on **silver oxide** (but can be alkaline). Expensive; numerous different sizes; can be hard to find. 1.5 V.

Zinc air cells are used in hearing aids and other applications with a low, steady drain. 1.4 V. (They have replaced obsolete mercury cells.)

Secondary (rechargeable) cells

Nickel metal hydride (NiMH) is probably the best choice at present. The cells have a higher capacity than the older nickel cadmium (NiCd) type, are less prone to show ‘memory’ effects, can be charged more quickly, and are less damaging to the environment. Nominally 1.2 V.

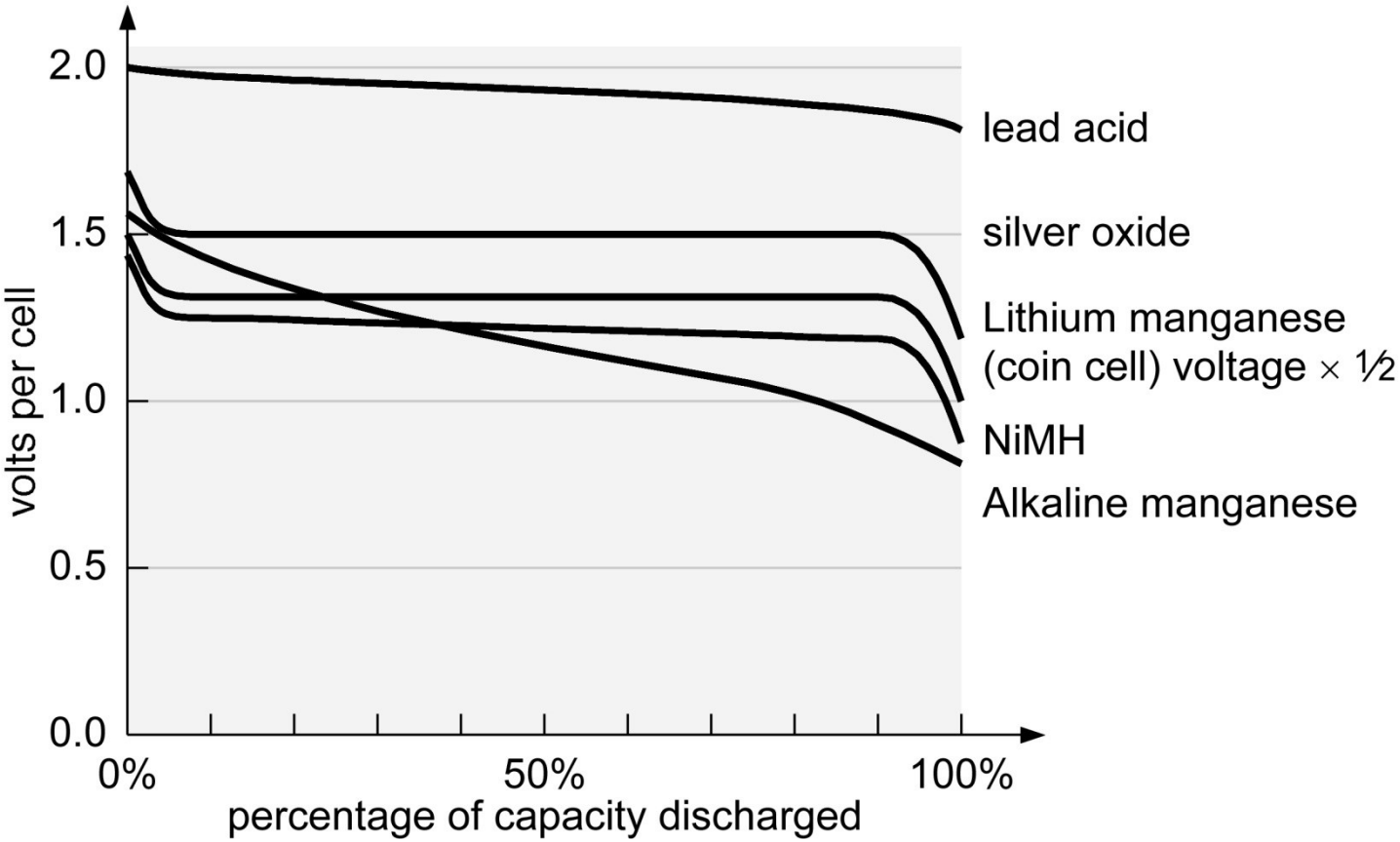
Lithium ion cells are now widely used in laptop computers. Tricky! — need specialized circuits to control them.

Lead acid is the technology used in car batteries. Modern batteries are sealed and resistant to abuse. Often used as backup in case of mains failure, kept trickle-charged while the mains is operating. Heavy! 2.0 V.

Special batteries are made for **backup in computers**. Secondary cells don’t last forever: about 5 years and 250–1000 charge–discharge cycles. They suffer from self-discharge if not used, especially NiCd and NiMH.
Each type of cell needs a different procedure to charge it correctly. You can buy complete chargers or integrated circuits to supervise this.

Discharge curves

These show the voltage produced by a cell as a function of its capacity as it is discharged. (The details depend on temperature, rate of discharge,…)



For what voltage should I design a circuit?

The discharge curves show that it is not easy to choose the voltage at which a circuit should operate! A common choice is 2 AA cells. This means that the supply is 3 V, isn't it?

- Alkaline cells produce over 1.5 V when they are new, but this drops steadily through their life (which makes it easy to determine how much energy is left). The end-point is often taken as 0.9 V, so 2 AA cells produce only 1.8 V at the end of their life!
- NiMH and NiCd produce about 1.2 V throughout most of their life, but the voltage plummets when they have discharged. In this case 2 AA cells produce 2.4 V.

Thus a well designed product that uses 2 AA cells should be capable of operating from 1.8 V if primary cells are permitted.

Some products now come with instructions forbidding the use of alkaline cells, perhaps so that they can be designed for 2.4 V rather than 1.8 V!

Capacities of different types of cell

The capacity of a cell is usually specified in ‘ampere hours’ or ‘milliampere hours’ (mAh).
For example, a capacity of 1000 mAh means in principle that the cell can supply 1 mA for 1000 h or 10 mA for 100 h and so on. In practice the capacity depends on the rate at which the cell is discharged, the temperature, and often on the history of the cell.
Here are some typical capacities and (very) rough costs. Coin and button cells come in many different sizes.

type	size	capacity (mAh)	rough cost
zinc chloride	AA	1000	40 cent
alkaline manganese	AA	2500	75 cent
NiMH	AA	1000–2000	200 cent
lithium (3 V)	coin	40–300	150 cent
silver oxide	button	10–150	150 cent

How can different voltages be generated?

For example:

- how can 5 V be provided to part of a system, if the rest uses 3 V?
- how can supplies of both signs be generated from a single battery (e.g. for a RS-232 transmitter)?
- how can I run a 3 V system, such as a pocket MP3 player, from a single AA cell?

Another important consideration is the sloping discharge curves of alkaline cells. If a circuit is designed to operate correctly at an end-point voltage of 0.9 V, it may waste much of the power when the cell can provide 1.5 V.

Ideally we need power supplies that can

- take an input of variable voltage
- produce an output of fixed, regulated voltage that may be lower than the input (the traditional supply), higher, or of the opposite sign

This sounds like a dream but solutions are readily available nowadays.

Charge pumps and switching supplies

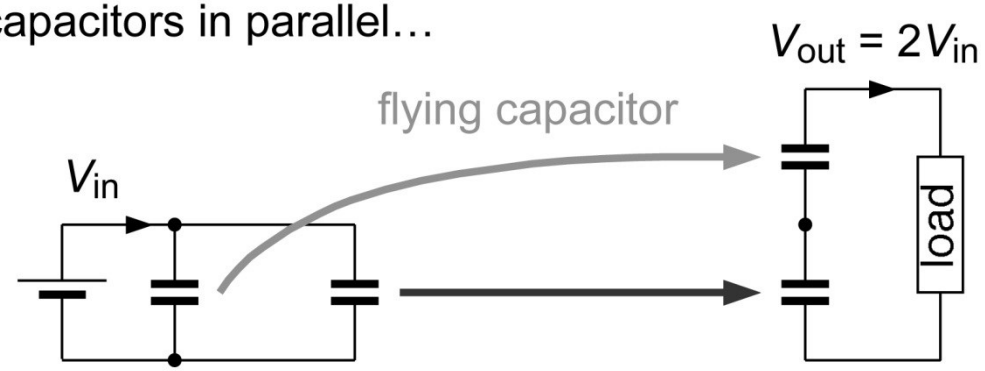
There are two general approaches, because electrical energy can be stored either in an inductor or a capacitor:

- switch-mode power supplies use inductors
- charge pumps use 'flying capacitors'

Simple charge pump:

charge two capacitors in parallel…

…then discharge them in series.



The switching is carried out by MOSFETs. A wide range of pumps is available (e.g. from Maxim) to provide *V*out = 2*V*in, –*V*in and others.
This technique is also used in RS-232 transmitters to generate ±12 V.

Brownout and power supervision

What happens if there is a transient loss of power in a system — a problem with the mains supply, for instance?

- If the break in power is so long that smoothing capacitors in the power supply discharge, the voltage *V*DD at the microcontroller will probably fall so low that the chip will restart when power is restored.

This is clearly a serious event because data in RAM will be lost, but at least it is clear that a restart has taken place.

- If the interruption is short, it is possible that data may be lost in some parts of the system but not others. The unaffected parts may attempt to use corrupted data from the parts that were affected.

This is called a **brown-out** and is very serious if it is not detected, because of corrupt data. Many microcontrollers therefore have brownout detectors (low voltage inhibitors) that force the chip to reset if the voltage falls below a specified value. Dedicated ICs are also available.

Complicated systems may need special power supervision circuitry to ensure that different parts start in the correct sequence.

Watchdog timer

Most microcontroller contain a ‘watchdog timer’ or ‘computer operating properly’ timer.
This comprises a timer and counter that runs independently of the rest of the chip. If the counter overflows (i.e. exceeds its range), the watchdog ‘times out’ and resets the whole microcontroller.

To avoid this happening, your program should clear the watchdog timer regularly — well before it times out. If, however, your program gets stuck in an unintentional infinite loop, the watchdog will not be cleared. The counter will then overflow and reset the chip.

Thus the watchdog protects the system from infinite loops, in a rather heavy-handed way.
Often you don’t want this ‘protection’, so turn the watchdog off!
The watchdog in Microchip microcontrollers can also be used to wake it up from sleep — next topic.

Sleep

Many microcontrollers spend a lot of time doing nothing!

- remote controls do nothing until the user presses a control
- a data logger may be required to take a reading only once per minute

It is clearly desirable to reduce the power consumption as far as possible when there is nothing useful to do. Most microcontrollers therefore have a **sleep**, **stop**, **idle** or **standby** mode. When this is entered,

- the main clock oscillator stops (but not always the watchdog timer!)
- execution of instructions therefore stops
- all data in memories is retained ('fully static')
- the outputs remain active (but of course will not change their values)

The current drawn when asleep is very low in modern microcontrollers — the data sheet claims 1 nA (10^{-9} A) for the 16F630 and below 1 µA for an older PIC. Ensure that all inputs are at *V*SS or *V*DD to keep the current low!

Wake up from sleep

Microcontrollers can be woken by either internal or external signals, typically:

- when the voltage on an input pin changes logical value
- an internal oscillator and counter ‘times out’ after a known delay. PICs use the watchdog timer for this; Motorola use a timer in the keyboard interface module

There is a delay after the wake-up signal has been detected, to allow time for the clock oscillator to stabilize. The microcontroller may then simply pick up from where it had gone to sleep, or may reset and start at the beginning again (in which case it is important to check whether the µC is starting after the power was first turned on or after a wakeup).

The low current drawn while asleep means that there is no need for an on–off switch: most modern electronics is partly ‘on’ at all times.

Strategies for saving power

Use a slow clock:
For example, the PIC16F630 with a 3 V supply draws

- 20 µA at 32 kHz clock
- 400 µA at 4 MHz

Some microcontrollers can be switched between fast and slow clocks to save power when fewer operations are needed.

Turn off all peripherals that are not needed — ADCs and so on can be switched off when not in use.

Ensure that all inputs are driven or connected to *V*SS or *V*DD — don't forget internal pull-ups, for instance.

Put the processor to sleep whenever possible — it is usually more efficient to perform tasks quickly and go back to sleep, rather than keep working at a slow pace. We'll look at this next.

Use modern components, designed for lower supply voltages

Power consumption — example (1)

A data logger based on a PIC16F630 running at 3 V takes a reading every second. It takes about 1,000 instructions to process and store the data.
What is the best approach to minimize the average current? Consider:

1. running continuously with 4 MHz oscillator (1 MHz instruction clock)
2. running continuously at 32 kHz oscillator (8 kHz instruction clock)
3. sleeping between readings, then processing the data at 4 MHz
4. sleeping between readings, then processing the data at 32 kHz

Take the current to be 1 µA when asleep to allow for other components.

The answers to the first two are trivial, from the previous slide:

1. running continuously at 4 MHz — 400 µA
2. running continuously at 32 kHz — 20 µA

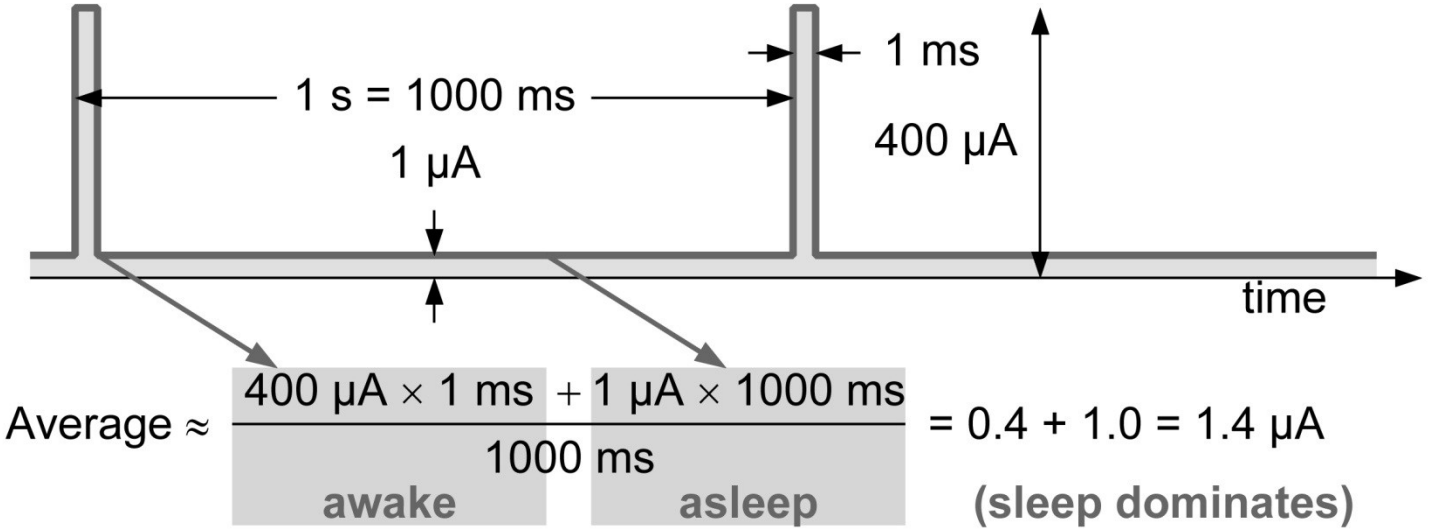
These are obviously going to be the highest figures, but are nevertheless pretty low — modern microcontrollers are amazingly efficient.

Power consumption — example (2)

Next we need to calculate the average current for:

3. sleeping between readings, then processing the data at 4 MHz

The processor draws 400 µA when running and 1 µA when asleep. For how long is it awake? It must execute 1,000 instructions and the instruction clock is 1 MHz (oscillator ÷ 4), so this takes 1 ms. The current as a function of time looks like this (not to scale).



Power consumption — example (3)

Finally:

4. sleeping between readings, then processing the data at 32 kHz

The processor draws 20 µA when running and 1 µA when asleep.
This time the processor is awake for 1,000/8,000 s = 125 ms.

The average current is therefore

$$\frac{\underbrace{20\ \mu A \times 125\ ms}_{\text{awake}} + \underbrace{1\ \mu A \times 1000\ ms}_{\text{asleep}}}{1000\ ms} = 2.5 + 1.0 = 3.5\ \mu A \quad \text{(wake dominates)}$$

Conclusion: sleep as long as possible, then get the processing done as quickly as possible.

(But in fact all the currents are impressively low!)

Power consumption — example (4)

How long would a pair of AA cells last in this equipment?
Capacity of cells = 2500 mAh.
Current in most economical case = 1.4 μA = 0.0014 mA.

$$\text{Lifetime in hours} = \frac{2500}{0.0014} = 1{,}800{,}000 \text{ hours} = 200 \text{ years!}$$

Lifetime in hours = 1,800,000 hours = 200 years!

Of course the cells would expire through self-discharge long before this! (Never use NiCd or NiMH for such an application.)

The calculation has been over-simplified because we should include the current drawn by the sensor and other components in the data logger, but the principle is sound.

Conclusions

Looked at common sources of power

- mains — cheap but needs PSU
- batteries — wide choice, expensive, annoying characteristics
- switching supplies and charge pumps to give different voltages

Watchdog timer

- remember to turn it off if you don't want it!

Sleep mode

- valuable approach to save energy
- no on–off switch needed
- can be woken by external events or an internal timer

Strategies for saving power

- calculations for a typical example
- **modern electronics can be amazingly efficient in its use of power!**

Practicals: Introduction to RTC

Real Time Clock is a device which is able to store and increment current time, even when external power is out.

Once more, we will be connecting external module with RTC to our board over TWI. We will use functions from i2c_master.c once more, you can copy i2c_master.h and i2c_master.c from lab 6.

Repetition: I2C functions

As you can see in i2c_master.c and i2c_master.h for most of I2C tasks there are functions prepared:

- **master_open_write(uint8_t adr); //open slave „adr“ for writing**
- **master_open_read (uint8_t adr); //open slave „adr“ for reading**
- **master_close(); //close current connection. Use after writing complete.**

- **master_write(uint8_t oneByte); //write oneByte to the active slave**
- **master_open_read_next(); //return next Byte, use if you will need to read more**
- **master_open_read_last();//read last Byte, use to end reading.**

DS1307 RTC

To address slave RTC we will need to open it for write and read with its address. According to documentation, address is 1101000 + RW bit.

*Fortunately, i2c_master_open functions take care of RW bit themselves, therefore we can use address 1101 0000 for both of them. In **hexadecimal**, 1101 is **D**, therefore address becomes **0xD0**.*

Prepare: look up how do functions from i2c_master take care of RW bit!

*To complete time_read function, refer to p.12 of lab instruction(**Timekeeper Registers**).*

To complete the rest, make sure you understand functions in main.c, make a list of them with a simple explanation of what does function do, in case you will feel confused.

Microcontroller

Summary:

Input / Output pins, Timer/Counter, PWM, Code examples …

(third iteration, but some errors still could be expected!)

HOCHSCHULE RHEIN-WAAL
Rhine-Waal University of Applied Sciences

Summary: Input pins (1)

We use Data Direction Registers to choose whether particular pin works as input (0 in DDR) or output (1 in DDR).

Furthermore, if we expect ACTIVE_LOW input, we would set pullup on the pin.

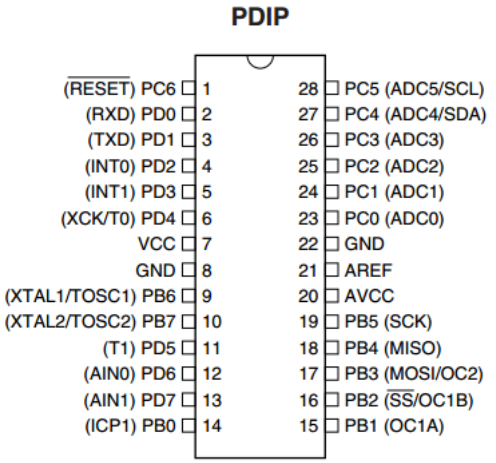
To ensure a pin is in input mode:

```
DDRX &= ~(1<< pin); //set pin as input
```

Where **X** is **D, B** or **C** for pins **PD**, **PB** and **PC**;
pin is PD0-PD7, PB0-PB5 or PC0-PC6.
*** on AVR board pins are further limited, see numbers under the pins to ensure right pin name.**

```
PORTX |= 1 << pin ; // enable pullup on pin
```

*Remark: Some more complex devices require used pins to be set to input, for example **ADC** and **SPI** communication. Do not use pullup for them.*



Summary: Input pins (2)

After the setup for Input is done, the input values will appear in PINX.

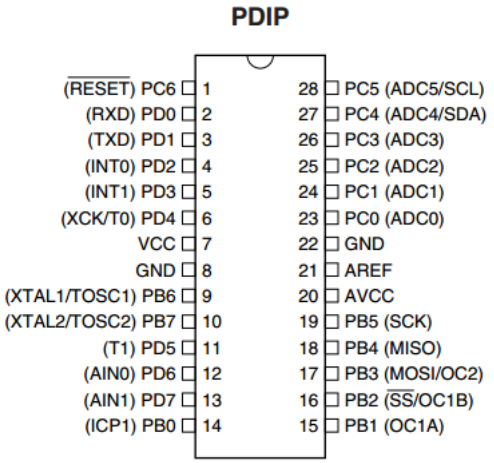
Use bitmasks to read particular bits.

To check if a pin is high or low:

```
if(PINX & (1<<pin))
{          //TODO          }
```

To check if an ACTIVE LOW pin is active:

```
if(~PINX & (1<<pin))
{          //TODO          }
```



Summary: Output pins

To output a digital value, PWM signal or use pins in modes like SPI, set corresponding DDR bits to 1:

```
DDRX |= (1<< pin); //set pin as output
```

Then use **PORTX** to choose if output should be high or low:

```
PORTX |= (1<< pin);  //set pin to high
PORTX &= ~(1<< pin); // set pin to low
```

DDR / PORT setup and usage example

Assume we want to have 4 outputs for LEDs and 2 inputs for buttons (ACTIVE_LOW):

```
void init(void)
{
DDRB |= 1<< PB0| 1<< PB1| 1<< PB2| 1<< PB3; //outputs
DDRD &= ~(1<< PD2| 1<< PD3); //inputs
PORTD |= 1<< PD2 | 1<<PD3; //pullup on PD2 and PD3
}

main(){
while(1)
{

if(~PIND & (1<< PD2| 1<< PD3)) PORTB = 255; //all LEDs ON
else PORTB = 0; //all LEDs ON

}}
```

Summary: ADC

The function of **ADC**: to represent **analog inputs** in **digital form**.

In practice, you are likely to use it for:

- **Reading potentiometer value**
- **Reading analog sensor**
- **Measuring some voltage in circuit**

How to use it in our code?

ADC operation is controlled by two registers (mostly)

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

ADC modes of operation

ADC will provide the digital representation of analog value in 8-bit registers **ADCH** and **ADCL**, which could be referenced from our code by **uint16_t ADCW**:

```
myReading = ADCW; //16 bit value (ADC is only 10 bit long)
```

There are 2 modes of operation: **normal** and **free running**.

Normal mode means that after writing **1** to **ADSC**, **exactly 1 conversion** will be made and ADSC will be automatically reset (back to **0**) by hardware (waiting for user to place **1** there again, at the time needed for conversion).

Free running mode will do conversion with some frequency determined by ADC prescaler, and keep doing it as long as **ADSC == 1** (remains unchanged all the time)

ADC input channel

ADC can receive the analog value for conversion from **6 pins**: **PC0-5**. However, at a single moment we must connect it only to **one** of the pins, and this is done using ADMUX register. There are **4 MUX** bits: **MUX3, MUX2, MUX1, MUX0.** You can change the ADC channel during runtime.

The setting for **MUX** bits is just **number of pin written in binary form**, as you can verify with the table below.

Table 75. Input Channel Selections

MUX3..0	Single Ended Input
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5

Atmel 199

ADC Example incl. ADC interrupt

There is an option to call an interrupt every time conversion is completed. Setting for this would be:

```
ADCSRA |= 1<<ADEN; // Enable ADC
ADCSRA |= 1<<ADIE;  //Enable interrupt
ADMUX |= pin;// Choose your input pin

DDRC &= ~( 1<< pin); //Make sure input pin
                     // also works as input!

volatile uint16_t myVariable;

ISR(ADC_vect)
{
myVariable = ADCW;
//TODO: use the reading!
}
```

ADC – further settings (1)

As already mentioned, ADC has few more settings:

Prescaler frequency (different from timer prescalers!):

Table 76. ADC Prescaler Selections

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

For example, use the highest prescaler to maximise conversion precision:

```
ADCSRA |= 1<<ADPS2  | 1<<ADPS1 | 1<<ADPS0;
```

ADC – further settings (2)

Free running mode:

```
ADCSRA |= 1 << ADFR;
```

Choose reference voltage:

Table 74. Voltage Reference Selections for ADC

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal V_{ref} turned off
0	1	AV_{CC} with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

```
ADSCRA |= 1 << REFS0; //AVcc as reference
```

ADC – full setup example

Assume we want to convert a value of **potentiometer connected to PC3 continuously**, and use it to **control output of some device (e.g. DC motor or LED):**

```
void init(void)
{
ADCSRA |= 1<< ADEN | 1<< ADFR | 1<< ADSC;//ADC enabled, free run
ADMUX |= 1<< MUX0 | 1<< MUX1;//Selected ADC channel: 3
DDRC &= ~(1<< PC3);//Required: ADC3 configured as an input
}

main(){
while(1)
{

output = ADCW; // output can be OCR register for PWM. If it is
// a simple variable, you must actually use it
// with more comments / statements.

}}
```

Summary: Timer/Counters (1)

As you might have already learned, AtMega8 has 3 Timer/Counters.

They are:

Timer/Counter0 – 8 bit (**TCNT0** overflows after **255**)
Timer/Counter1 – 16 bit (**TCNT1** overflows after **65535**)
Timer/Counter2 – 8 bit (**TCNT2** overflows after **255**)

How Timer/Counter work:

if prescaler is chosen to be value **P** (e.g. **P** = **1, 8, 64, 1024**),
then corresponding **TCNT** register will be **incremented (increased by 1)** every time **clock generates P signals** (in our case, clock normally generates 8M signals per second).

If prescaler is not set, **TCNT** is never changed in hardware (timer is turned off).

Summary: Timer/Counters (2)

Therefore, to calculate the **frequency of overflows**:

$f_{overflows} = f_{clock} / (P * 2^{(bits\_in\_timer\_counter)})$ where **P** is prescaler value, **bits_in_timer_coutner** is **8** for **TC0** and **TC2**, or **16** for **TC1**.

If you measure some time in **T** = **TCNT**[Timer/Counter units] (assume you measured time to be **N** [TC units], like 123 or some other number) and want to know how much it is in seconds [s]:

$T\ [s] = T\ [TC\ units] * P / f_{clock} = TCNT * P / f_{clock}\ [s]$

From this follows, that maximum time Tmax you can directly measure in TC is:
(for maximal prescaler of 1024)
Since N is 255 at most for **TC0** and **TC2** → Tmax = 255*1024 / 8M = **32,64** ms
Since N is 65535 at most for **TC1** → Tmax = 65535 *1024/8M = **8,355** s

Summary: Timer/Counters (3)

How to measure **any** time using Timer/Counter **(time greater than Tmax)** ?

Simply count the number of overflows!

```
uint16_t overflows_TC1 = 0;

ISR(TIMER1_OVF_vect)
{
overflows_TC1++;
}
```

Then actual time would be:
t = `overflows_TC1` * *T1* + *T[s]*,
where T1 is period of overflows of TC1 (inverse of frequency, see last slide for values / formula) and T[s] is equal ***TCNT**** ***P*** / f_{clock} *[s]*

Summary: Prescaler init TC0 (1)

For TC0:

```
TCCR0 |= (1 << bit_name);    //bit_name could be:
                             //CS02, CS01, CS00.
```

If you want a bit to be 0, just leave it not initialized (0 is default value).

Table 34. Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge
1	1	1	External clock source on T0 pin. Clock on rising edge

Summary: Prescaler init TC0 (2)

Complete statements would be **(choose only ONE line of the statements!)** :

```
TCCR0 |= (1 << CS00) ;    // P = 1

TCCR0 |= (1 << CS01) ;    // P = 8

TCCR0 |= (1 << CS01) | (1 << CS00) ;    // P = 64

TCCR0 |= (1 << CS02) ;    // P = 256

TCCR0 |= (1 << CS02) | (1 << CS00) ;    // P = 1024
```

Table 34. Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge
1	1	1	External clock source on T0 pin. Clock on rising edge

Summary: Prescaler init TC1

Complete statements for TC1 **(choose only ONE line of the statements!)** :

```
TCCR1B |= (1 << CS10) ;    // P = 1

TCCR1B |= (1 << CS11) ;    // P = 8

TCCR1B |= (1 << CS11) | (1 << CS10) ;    // P = 64

TCCR1B |= (1 << CS12) ;    // P = 256

TCCR1B|= (1 << CS12) | (1 << CS10) ;    // P = 1024
```

Table 40. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge
1	1	1	External clock source on T1 pin. Clock on rising edge

Summary: Prescaler init TC2

Complete statements for TC2 **(choose only ONE line of the statements!)** :

```
TCCR2 |= (1 << CS20) ;    // P = 1

TCCR2 |= (1 << CS21) ;    // P = 8

TCCR2 |= (1 << CS21) | (1 << CS20) ;    // P = 32

TCCR2 |= (1 << CS22) ;    // P = 64

TCCR2|= (1 << CS22) | (1 << CS20) ;    // P = 128

TCCR2|= (1 << CS22) | (1 << CS21) ;    // P = 256

TCCR2|= (1 << CS22) | (1 << CS21) | (1 << CS20) ; ;    // P = 1024
```

Table 46. Clock Select Bit Description

CS22	CS21	CS20	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	clk_{T2S}/(No prescaling)
0	1	0	clk_{T2S}/8 (From prescaler)
0	1	1	clk_{T2S}/32 (From prescaler)
1	0	0	clk_{T2S}/64 (From prescaler)
1	0	1	clk_{T2S}/128 (From prescaler)
1	1	0	clk_{T2S}/256 (From prescaler)
1	1	1	clk_{T2S}/1024 (From prescaler)

Example: choosing prescaler for 1 Hz (1)

Assume your goal is to get some event repeated every second.

There will be actually few ways to have it done:

- Check TCNT in the main loop, as soon as >= than 1 second expires, you execute your event and set TCNT to 0 manually;

```
if(TCNT >= oneSecond_in_TCNT_Units )
    {myEvent();
    TCNT=0;}
```

- Compare interrupt: instead of checking in the main loop, you let the hardware do it for you. You clear TCNT manually;

- CTC (Time Compare mode) on TC1 + overflow interrupt

```
//MODE 4 : CTC
TCCR1B |= 1<< WGM12;
```

Example: choosing prescaler for 1 Hz (2)

In all of the scenarios, you need to choose a prescaler which will let time to count upward of 1 s.

This means that frequency coming from formula must be less than 1 Hz:

$f_{overflows} < 1\ Hz$

$f_{clock} / (P * 2^{(bits\_in\_timer\_counter)}) < 1\ Hz$

$f_{clock} / (2^{(bits\_in\_timer\_counter)}) < P$

For 8 bit:

$P > 31250$

Can this condition be satisfied?

We could use overflows counter variable to count to 1 second with TC0 and TC2. Example will continue with TC1 where this trick is not needed.

Example: choosing prescaler for 1 Hz (3)

For 16 bit:

P > ***122.07***

Therefore following prescaler settings are appropriate:

```
TCCR1B |= (1 << CS12) ;    // P = 256
TCCR1B|= (1 << CS12) | (1 << CS10) ;    // P = 1024
```

```
What value should be check for to see 1 second has expired?
```

TCNT = ***t*** * f_{clock} / ***P*** , *where t is = 1 s (or just 1)*
31250 *for* ***P = 256***
7813 *for* ***P = 1024*** *(7812.5 rounded up)*

Summary: TC interrupts (1)

The simplest interrupt you can have for a TC is overflow interrupt. As long as TC is enabled (prescaler is given a value, = not zero), overflow interrupt will be called with frequency $f_{overflows} = f_{clock} / (P * 2^{(bits\_in\_timer\_counter)})$

It lets you do some tasks with a known period.

To enable this interrupt:

```
sei(); //global interrupts enabled
//choose only interrupts that you need
TIMSK |= 1<<TOIE0 ; // overflow interrupt enable for TC0
TIMSK |= 1<<TOIE1 ; // overflow interrupt enable for TC1
TIMSK |= 1<<TOIE2 ; // overflow interrupt enable for TC2
```

7	6	5	4	3	2	1	0	
OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	–	TOIE0	TIMSK
R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
0	0	0	0	0	0	0	0	

Summary: TC interrupts (2)

Do not forget to provide the Interrupt Service Routines outside of main function:

```
ISR(TIMER0_OVF_vect) //can be TIMER1, TIMER2 as well
{
  //TODO
}
```

otherwise your program may crash!!!

Another very important interrupt for TCs is compare interrupt. You can choose a value for **OCR1A**, **OCR1B** for **TC1** and **OCR2** for **TC2**. (OCR stands for Output Compare Register, and there are 3 of them: **1A**, **1B** and **2**)

Whenever TC will reach the value currently in Output Compare Register, the interrupt will be called.

Summary: TC interrupts (3)

To enable Output Compare Interrupts:

```
sei();
TIMSK |= ( 1 << OCIE1A ); //for OCR1A
TIMSK |= ( 1 << OCIE1B ); //for OCR1B
TIMSK |= ( 1 << OCIE2 ); //for OCR2
```

Remember, for this to work there must be a prescaler chosen for corresponding TC, as well as OCR value set correspondingly.

Use

```
ISR(TIMER1_COMPA_vect) //for OCR1A
ISR(TIMER1_COMPB_vect) //for OCR1B
ISR(TIMER2_COMP_vect) //for OCR2
```

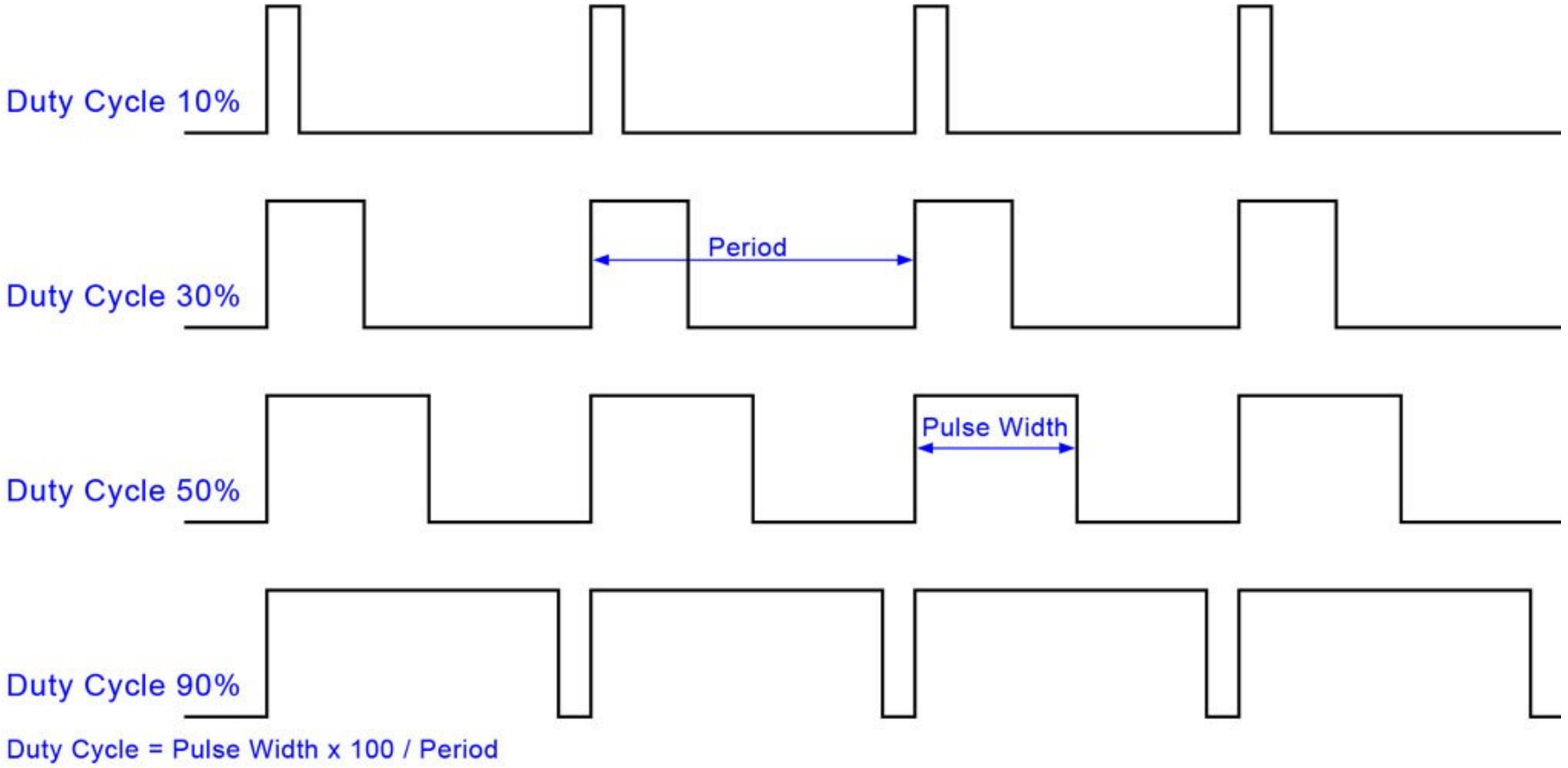
service routines!

7	6	5	4	3	2	1	0	
OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	–	TOIE0	TIMSK
R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
0	0	0	0	0	0	0	0	

Hardware PWM using TCs (1)

Background / terms used:

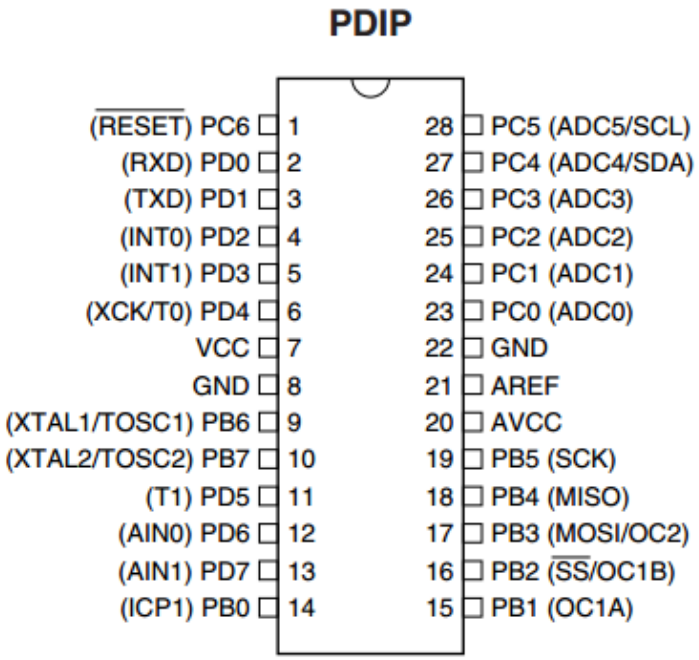
PWM – pulse width modulation is a kind of signal which looks like in the graph:



Hardware PWM using TCs (2)

TC1 and TC2 can generate PWM in hardware. It is more precise and is independent of any other code running on your device, therefore it is recommended to always use Hardware PWM.

Important: Hardware PWM can be provided only on pins PB1, PB2, PB3.



Hardware PWM using TC1 (1)

Setup of TC1 is slightly complicated by the fact that it‘s control register is split into two parts, A and B.

Use quoted tables to determine bits you need to set for your operation mode -> find in which register part those bits are -> use known syntax:

```
REGISTER   |=  ( 1<< bit_to_set )   ;
```

Bit	7	6	5	4	3	2	1	0	
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	W	W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

7	6	5	4	3	2	1	0	
ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
0	0	0	0	0	0	0	0	

Hardware PWM using TC1 (2)

Table 39. Waveform Generation Mode Bit Description

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation[1]	TOP	Update of OCR1x	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP

Table 39. Waveform Generation Mode Bit Description (Continued)

Mode	WGM13	WGM12 (CTC1)	WGM11 (PWM11)	WGM10 (PWM10)	Timer/Counter Mode of Operation[1]	TOP	Update of OCR1x	TOV1 Flag Set on
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Note: 1. The CTC1 and PWM11:0 bit definition names are obsolete. Use the WGM12:0 definitions. However, the functionality and location of these bits are compatible with previous versions of the timer

Hardware PWM using TC1 (3)

From all the PWM modes, let us pay attention to:

Modes 5,6,7 – 8, 9, 10 bit fast PWM
(convenient to control LEDs, heaters, fans, magnets, etc.)
Frequency of PWM in this mode will be: $f_{overflows} = f_{clock} / (P * 2^{(PWM\_bits)})$

When choosing PWM mode consider: The more bits you have in your PWM, the more precise you can choose the output level (just like bits in ADC). On the other hand, your PWM frequency decreases as
f ~ 1 / (2^bits_in_PWM)

Low frequency means: slower response time; less smooth signal/reaction.

```
//MODE 5 : 8 bit fast PWM
TCCR1B |= 1<< WGM12;
TCCR1A |= 1<< WGM10;
```

Hardware PWM using TC1 (4)

```
//MODE 6 : 9 bit fast PWM
TCCR1B |= 1<< WGM12;
TCCR1A |= 1<< WGM11;

//MODE 7 : 10 bit fast PWM
TCCR1B |= 1<< WGM12;
TCCR1A |= 1<< WGM10 | 1<< WGM11;
```

Mode 14 – Fast PWM with top at ICR1
(used to control servos)
Frequency of PWM in this mode will be: $f_{overflows} = f_{clock} / (P * ICR1)$

```
//MODE 14 : fast PWM with top at ICR1
TCCR1B |= 1<< WGM13  | 1<< WGM12;
TCCR1A |= 1<< WGM10;
```

Hardware PWM using TC1: duty cycle

Duty cycle means what percentage of time the output is HIGH for your PWM signal.
I.e., it is the value of the analog signal you are approximating :

Vout = % duty cycle * Vcc

There are two options, duty cycle can be:
1) **(TOP – OCR)/TOP**
2) **OCR/TOP.**

TOP is 2^bits for modes 5-7 and ICR1 for mode 14. Make sure you do not choose OCR greater than TOP.

COM (Compare Output Mode) bits will determine if option 1 or 2 determines your duty cycle.

Hardware PWM using TC1: COM bits

Table 37. Compare Output Mode, Fast PWM[1]

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
0	0	Normal port operation, OC1A/OC1B disconnected.
0	1	WGM13:0 = 15: Toggle OC1A on Compare Match, OC1B disconnected (normal port operation). For all other WGM1 settings, normal port operation, OC1A/OC1B disconnected.
1	0	Clear OC1A/OC1B on Compare Match, set OC1A/OC1B at BOTTOM, (non-inverting mode)
1	1	Set OC1A/OC1B on Compare Match, clear OC1A/OC1B at BOTTOM, (inverting mode)

Depending on which channel you want to use (OCR1A on PB1 or OCR1B on PB2), you should set COM1A1 and COM1A0 or COM1B1 and COM1B0.

It is easiest to use option on 3rd row of this table.
Your duty cycle will be **OCR/TOP.**

```
TCCR1A |= 1<< COM1A1 ; // for OCR1A on PB1
TCCR1A |= 1 << COM1B1 ; // for OCR1B on PB2
```

Hardware PWM using TC1: conclusion

Therefore, to use Hardware PWM on TC1, you have to:

1. **Choose mode of operation according to the device you work with** and precision/speed requirement. **Initialize TC1 for this mode**.

2. Set COM bits for channels you want to use (you can have 2 PWM signals in the same mode but different duty cycle coming from TC1)

3. **Set OCR1A or OCR1B and enjoy your PWM on PB1 or PB2!**

Hardware PWM using TC2

Timer/Counter 2 has less modes to choose from:

Table 42. Waveform Generation Mode Bit Description

Mode	WGM21 (CTC2)	WGM20 (PWM2)	Timer/Counter Mode of Operation[1]	TOP	Update of OCR2	TOV2 Flag Set
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR2	Immediate	MAX
3	1	1	Fast PWM	0xFF	BOTTOM	MAX

To use mode 3: Fast PWM (8 bit), set:

```
TCCR2 |= 1 << WGM21 | 1 << WGM20;
```

Hardware PWM using TC2: COM bits

COM21	COM20	Description
0	0	Normal port operation, OC2 disconnected
0	1	Reserved
1	0	Clear OC2 on Compare Match, set OC2 at BOTTOM, (non-inverting mode)
1	1	Set OC2 on Compare Match, clear OC2 at BOTTOM, (inverting mode)

Again we will use non-inverting mode.
Your duty cycle will be **OCR2 / TOP.**

```
TCCR2 |= 1<< COM21 ; // for PWM on PB3
```

Hardware PWM using TCs: conclusion

Therefore, to use Hardware PWM on TC2:

```
TCCR2 |= 1 << WGM21 | 1 << WGM20;
TCCR2 |= 1<< COM21 ;

OCR2 = 200 ; // 200 is an example. Use value from 0 to 255
to choose duty cycle.
```

Remarks:

- **you can use PWM AND Compare interrupts at the same time.**
- **BUT: Manipulations to TCNTs can alter the functionality of both PWM and interrupts.**

Controlling servos

A typical servo will rotate between **NEGATIVE_LIMIT**, **0** and **POSITIVE_LIMIT** according to pulse length in **PWM** signal it receives.

The servos you work with in the labs need:
1 ms pulse for **NEGATIVE_LIMIT**;
1,5 ms pulse for **0** ;
2 ms pulse for **POSITIVE_LIMIT**.

According to specification, we should keep **pulse width** at **20 ms**, but in practice we can go as low as **7 ms**, updating servo set point even more frequently.

Controlling servos (1)

How many independent servos can a single ATMega8 control?

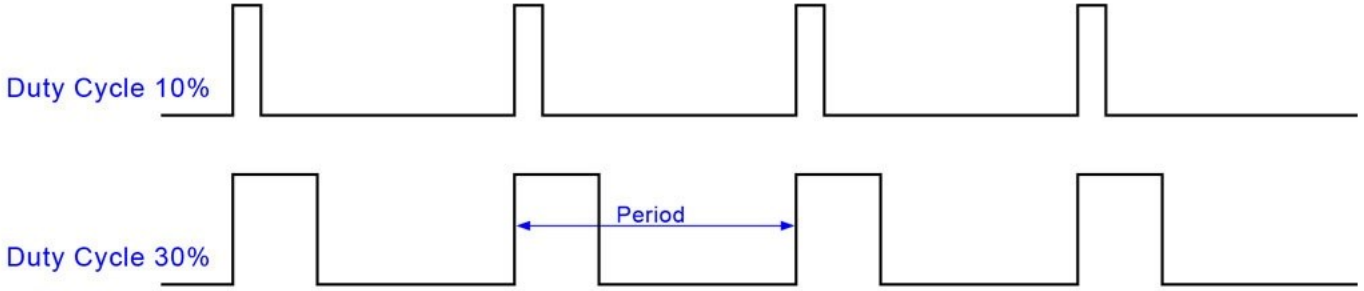
Actually all 3 with hardware PWM:

```
void servo_init(void){

    // Port einstellen
    DDRB |= (1 << DDB1) |(1 << DDB2) |(1 << DDB3);      // PB1,2,3 as Output
    PORTB &= ~( 1 << PB1);      // Low value for init


    // Timer 1
    // prescaler 8 => 8Mhz => 1us per step
    // Fast PWM mode 14 (top = ICR1)
    TCCR1B = (1 << CS11) | (1 << WGM12) | (1 << WGM13);
    TCCR1A = (1 << COM1A1) |(1 << COM1B1) | (1 << WGM11);
    ICR1 = 20000UL;


    TCCR2 |= 1<< COM21 | 1 << WGM21 | 1 << WGM20 | 1 << CS21| 1 << CS22;

    OCR1A = 1500;
    OCR1B = 1500;
    OCR2 = 47; //value for 1.5 second



}
```

And more with software PWM! Limited by total pin number and PWM current consumption, which could be made low.

Hardware PWM using TCs: conclusion (1)

On connection between the programming and real electrical signal:



First, selection of PWM mode will determine the Period of PWM (together with prescaler setting). Imagine, you use

//MODE 7 : 10 bit fast PWM → T = 2^10 * P / f_{clock}

```
TCCR1B |= 1<< WGM12;
TCCR1A |= 1<< WGM10 | 1<< WGM11;
```

And now you want to have duty cycle of 30%. What OCR value corresponds to this?

30%* 2^10 = 307. This is the value you should set in OCR1A or OCR1B.

8-bit PWM on PB1 and PB2, using TC1

```
void init(void)
{
DDRB |= 1<< PB1 | 1<< PB2;
TCCR1B |= (1 << CS10) ;    // P = 1
//MODE 5 : 8 bit fast PWM, frequency = 64 kHz
TCCR1B |= 1<< WGM12;
TCCR1A |= 1<< WGM10;
TCCR1A |= 1<< COM1A1 ; // for OCR1A on PB1
TCCR1A |= 1 << COM1B1 ; // for OCR1B on PB2
}

main(){
while(1)
{
OCR1A=200; //PB1 duty cycle ~80%, change this to your value
OCR1B=100; //PB2 duty cycle ~40%
}}
```

Servo PWM on PB1 and PB2, using TC1

```
void init(void)
{
DDRB |= 1<< PB1 | 1<< PB2;
TCCR1B |= (1 << CS11) ;   // P = 8
//MODE 14 : fast PWM with top at ICR1
TCCR1B |= 1<< WGM13  | 1<< WGM12;
TCCR1A |= 1<< WGM10;
TCCR1A |= 1<< COM1A1 ; // for OCR1A on PB1
TCCR1A |= 1 << COM1B1 ; // for OCR1B on PB2
ICR1=7000UL;
}
main(){
while(1)
{
OCR1A=1500; // PB1 sets servo close to 0 deg, 1.5 ms
OCR1B=1000; //PB2 sets servo close to minimal angle, 1.0 ms
}}
```

8-bit fast PWM on PB3, TC2

```
8-bit fast PWM on PB3, frequency = 64 kHz, TC2

void init(void)
{
DDRB |= 1<< PB3;
TCCR2 |= (1 << CS20) ;    // P = 1
TCCR2 |= 1<< COM21 ;
TCCR2 |= 1 << WGM21 | 1 << WGM20;
TCCR2 |= 1<< COM21 ; // for PWM on PB3
}
main(){
while(1)
{
OCR2=150; // set to the needed value
}}
```