# Computer Science 4723

**Fall Semester, 2013**

## What will we do in this course?

This course is primarily an introduction to *embedded systems.* Embedded systems typically are "invisible" computing systems, embedded in some product or process.
The processors are typically *controllers*, sampling some set of inputs and providing some appropriate set of outputs. Essentially, they are complex state machines.

- We will look at the architecture of a typical microprocessor system that is used in many kinds of embedded systems.
- We will look at some of the common peripheral devices contained within the same package as the host processor — timers, communication devices, etc.
- We will look at ways to acquire input from external sensors, and develop outputs which depend on these inputs.
- We will design some analog devices to "condition" inputs so they will be amenable to be input to our systems.
- We will design output controllers for different kinds of devices; e.g., motors, servo mechanisms, etc.
- We will examine different ways for devices to communicate with a host, or with each other.

Commonly cited examples of systems with embedded processors are digital cameras, cell phones, PDA's, the various digital media players, etc.
Today, almost any electrical appliance is likely to have an embedded processor (coffee makers, programmable thermostats, remote controls, coffee grinders, keyboards, etc.)
In fact, there are far more embedded processors in our lives than there are laptop or desktop computers or servers.
Their interconnectivity is also increasing dramatically — short and medium range communications (e.g., Bluetooth and wifi) are found in many systems; For example, Bluetooth has become common in cell phones (providing a link between short and long range communications.) Near-field communication (NFC) is also becoming common in cell phones and tablets.

One of the things we will do in this course is look at some devices for short range wireless communication; e.g., the kinds of devices used for wireless mouse and keyboard applications, or for short-range sensor networks.

## Why are embedded processors so popular/useful?

It would almost always be more efficient to have a controller specifically designed for a product, with the algorithms implemented directly in the hardware. The device could be smaller, cheaper (in high volume), and more efficient. So why use a computer?

Generally, the startup costs to manufacture a new device (integrated circuit device, anyway) are very high. Also, the design time for circuitry is usually considerably longer than for a program.

Essentially, startup costs, volume requirements, and time-to-market considerations dictate that products use standard, reconfigurable hardware elements that can be manufactured cheaply in very high volume.

Of course, there is a spectrum of such products, offering tradeoffs in cost, design time, and performance. Two such technologies are the embedded processors, and field programmable gate array logic (FPGA).

Both are reprogrammable, and differ primarily in cost, performance, and design time. Generally FPGA's can provide higher performance at higher cost, and with more design effort.

(Actually, many current FPGA's have microprocessors embedded in them, either implemented in the FPGA logic, or as a built-in core.)

We will concentrate on the embedded processor approach in this course.

## ATMEL AVR architecture

We will use the ATMEL AVR series of processors as example input/output processors, or controllers for I/O devices.

These 8-bit processors, and others like them (PIC microcontrollers, 8051's, etc.) are perhaps the most common processors in use today. Typically, they are not used as individually packaged processors, but as part of *embedded systems*, particularly as controllers for other components in a larger integrated system (e.g., mobile phones).
There are also 16- and 32-bit processors in the embedded systems market; the MIPS processor family is commonly used in the 32-bit market, as is the ARM processor. (The ARM processor is universal in the mobile telephone market.)

We will look at the internal architecture of the ATMEL AVR series of 8-bit microprocessors.
They are available as single chip devices in package sizes from 6 pins (external connections) to more than 100 pins, and with program memory from 1 to 256 Kbytes.

## AVR architecture

Internally, the AVR microcontrollers have:

- 32 8-bit registers, `r0` to `r31`
- 16 bit instruction word
- a minimum 16-bit program counter (PC)
- separate instruction and data memory
- 64 registers dedicated to I/O and control (and an additional 160 I/O registers in some devices)
- external interrupt capability, interrupt source is programmable
- most instructions execute in one cycle

The top 6 registers can be paired as address pointers to data memory.
The X-register is the pair `(r26, r27)`,
the Y-register is the pair `(r28, r29)`, and
the Z-register is the pair `(r30, r31)`.
The z-register can also point to program memory, allowing constant data to be stored there.
Generally, only the top 16 registers (`r16` to `r31`) can be targets for the immediate instructions.

There are several classes of AVR microcontrollers, targeting different types of applications. We will discuss two classes, the **tiny**AVR, and the **mega**AVR classes.

Both classes have nearly identical instruction sets; the core instructions for all AVR processors is identical. The megaAVR devices have some additional instructions, including signed and unsigned multiply instructions (8 bit operands, 16 bit result) that complete in two cycles.

MegaAVR devices also typically have a richer set of peripherals.
It is worthwhile to look at the ATmega48/88 and the ATtiny48/88 for comparison. These devices are the same physical size, and are pin compatible (moving from tiny to mega — there are some pin functions in the mega device not available in the tiny device).
There are 8 instructions in the mega device which are not implemented in the tiny – mainly multiply.
The tiny device has a maximum clock speed of 12MHz; the mega device, 20MHz.
There are more peripherals (e.g., 3 vs. 2 counters), more communication options, and more programmability of the peripherals in the mega device.
They both have the same size program memory, but the tiny device has half the RAM, and a smaller eeprom.
The tiny device is cheaper.

## Program memory

In all the AVR processors, the program memory is *flash programmable*, and is fixed until overwritten with a programmer. It is guaranteed to survive 10,000 rewrites.

In many processors, self programming is possible. A bootloader can be stored in protected memory, and a new program downloaded by a simple serial interface.

In the smallest devices (now obsolete), there is no data memory — programs use registers only. In those processors, there is a small stack in hardware (3 entries). In the processors with data memory, the stack is located in the data memory.

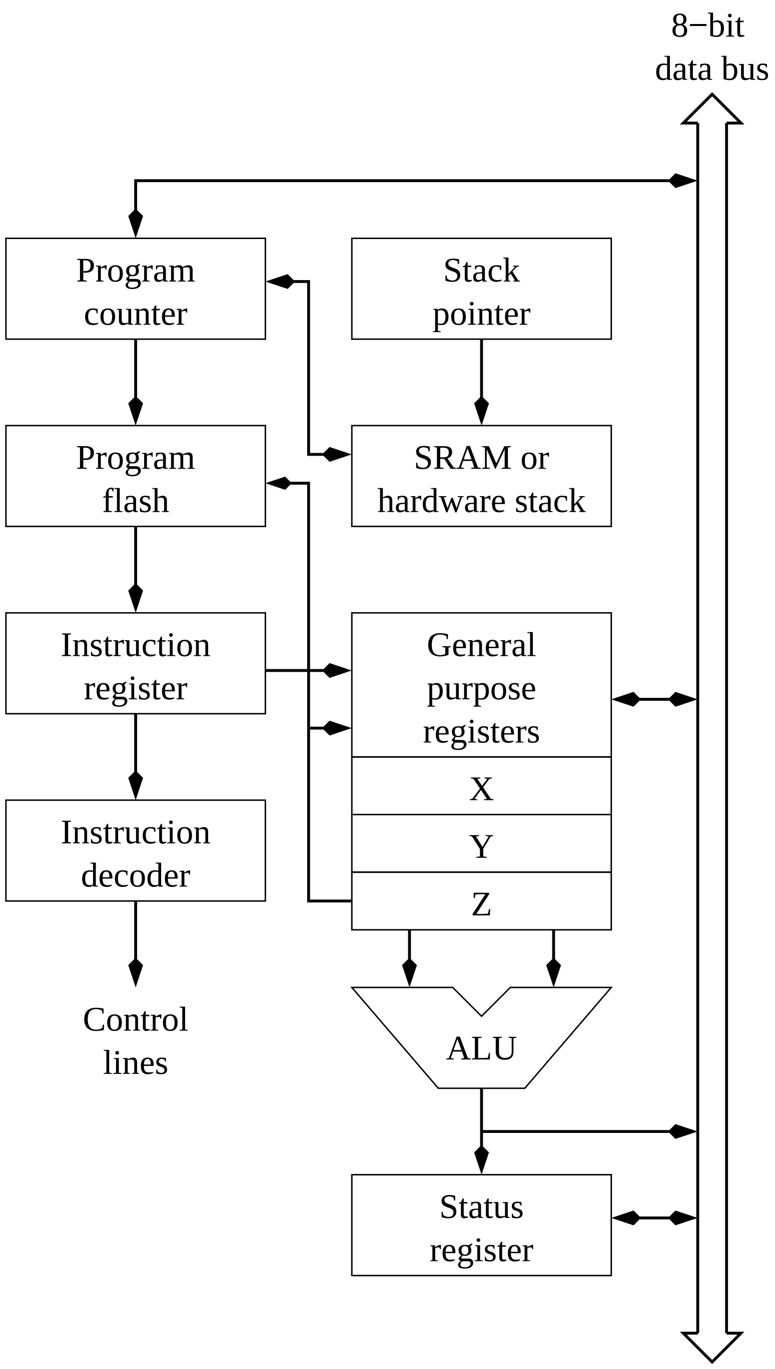
The size of on-chip data memory (SRAM — static memory) varies from 0 to 8 Kbytes.

Most processors also have EEPROM memory, from 0 to 4 Kbytes.

The C compiler can only be used for devices with SRAM data memory.

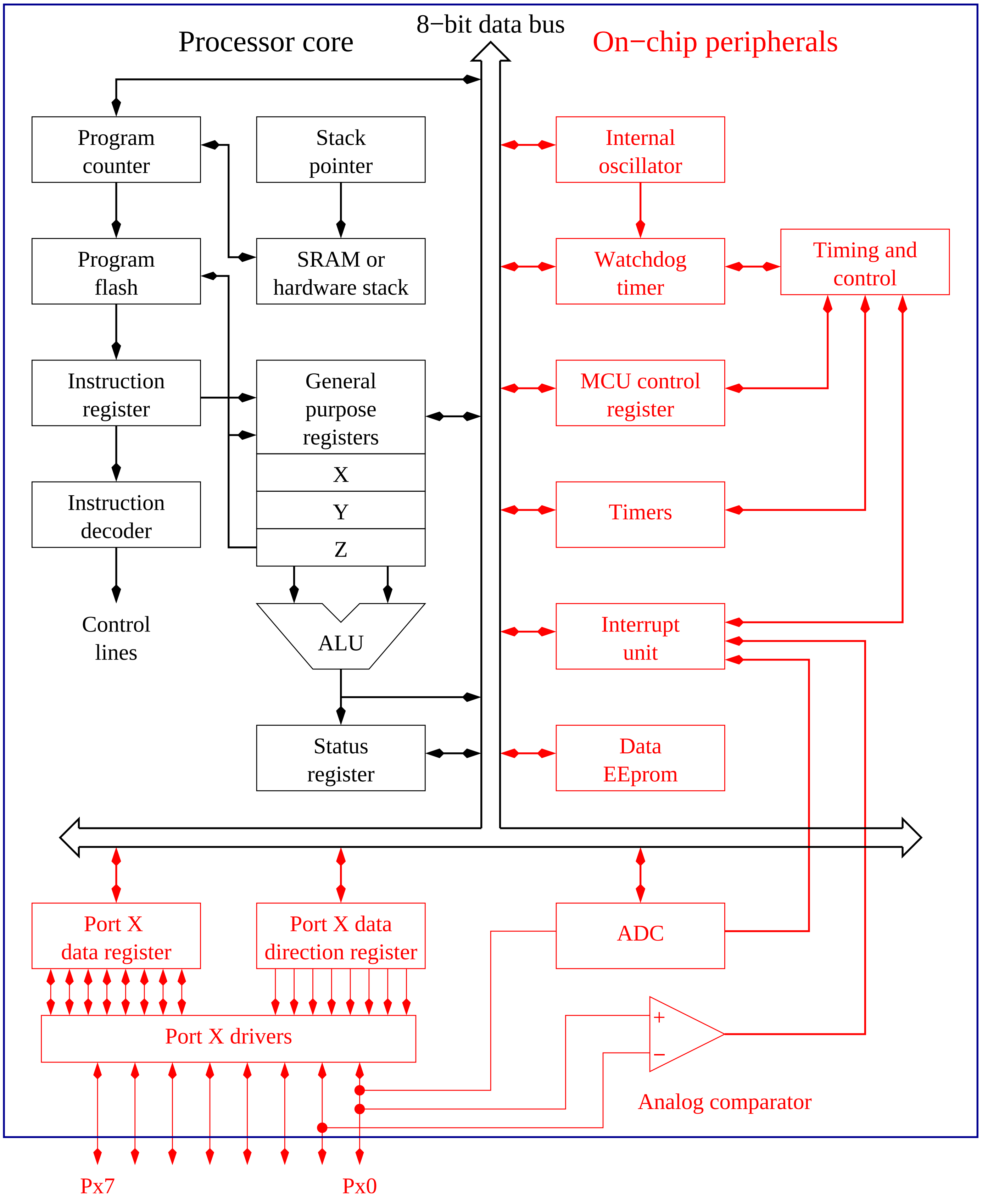
Only a few of the older ATMEL devices do not have SRAM memory.

## ATMEL AVR datapath



Note that the datapath is 8 bits, and the ALU accepts two independent operands from the register file.
Note also the status register, (`SREG`) which holds information about the state of the processor; e.g., if the result of a comparison was 0.

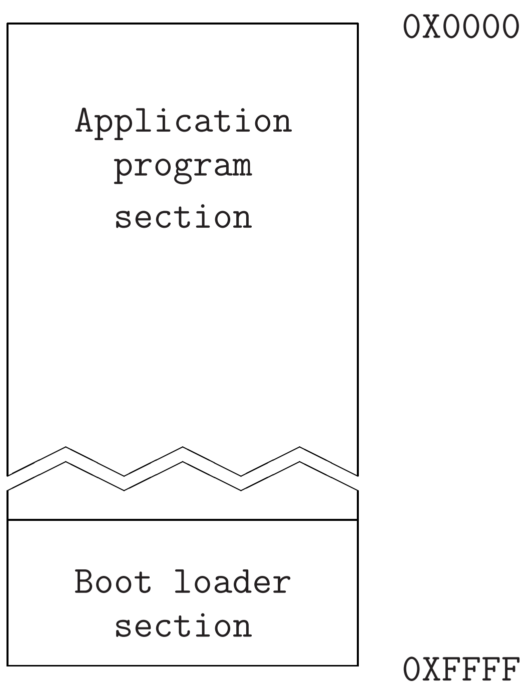
## Typical ATMEL AVR device



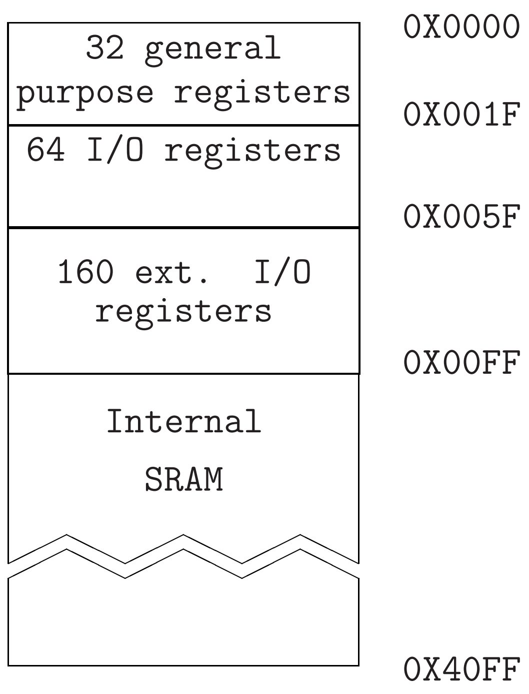
## The AVR memory address space

Instructions are 16 bits, and are in a separate memory from data.
Instruction memory starts at location 0 and runs to the maximum program memory available.
Following is the memory layout for the ATmega1284p:

**Program memory map**



**Data memory map**



Note that the general purpose registers and I/O registers are mapped into the data memory.

Although most (all but a few of the tiny) AVR processors have EEPROM memory, this memory is accessed through I/O registers.

## The AVR instruction set

Like the MIPS, the AVR is a load/store architecture, so most arithmetic and logic is done in the register file.
The basic register instruction is of the type `Rd ← Rd op Rs`
For example,

```
   add r2, r3    ; adds the contents of r3 to r2
                 ; leaving result in r2
```

Some operations are available in several forms.
For example, there is an *add with carry* (`adc`) which also sets the carry bit in the status register `SREG` if there is a carry.

There are also immediate instructions, such as *subtract with carry immediate* (`sbci`)

```
  sbci r16, 47  ; subtract 47 from the contents of r16,
                ; leaving result in r16 and set carry flag
```

Many immediate instructions, like this one, can only use registers 16 to 31, so be careful with those instructions.
There is no add immediate instruction.
There is an *add word immediate* instruction which operates on the pointer registers as 16 bit entities in 2 cycles. The maximum constant which can be added is 63.

There are also logical and logical immediate instructions.

There are many data movement operations relating to loads and stores. Memory is typically addressed through the register pairs `X` (`r26, r27`), `Y` (`r28, r29`), and `Z` (`r30, r31`).
A typical instruction accessing data memory is load indirect, `ld`.
It uses one of the index registers, and places a byte from the memory addressed by the index register in the designated register.

```
ld r19, X  ; load register 19 with the value pointed to
           ; by the index register X (r26, r27)
```

These instructions can also post-increment or pre-decrement the index register. E.g.,

```
ld r19, X+ ; load register 19 with the value pointed
           ; to by the index register X (r26, r27)
           ; and add 1 to the register pair

ld r19, -Y ; subtract 1 from the index reg. Y (r28, r29)
           ; then load register 19 with the value pointed
           ; to by the decremented value of Y
```

There is also a load immediate (`ldi`) which can only operate on registers `r16` to `r31`.

```
ldi r17, 14   ; place the constant 14 in register 17
```

There are also `push` and `pop` instructions which push a byte onto, or pop a byte off, the stack. (The stack pointer is in the I/O space, registers `0X3D, 0X3E`).

There are a number of branch instructions, depending on values in the status register. For example, *branch on carry set* (`BRCS`) branches to a target address by adding a displacement (-64 to +63) to the program counter (actually, `PC +1`) if the carry flag is set.

```
  brcs  -14   ; jumps back -14 + 1 = 13 instructions
```

Perhaps the most commonly used is the **`relative jump`** (`rjmp`) instruction which jumps forward or backwards by 2K words.

```
  rjmp  191   ; jumps forward 191 + 1 = 192 instructions
```

The *relative call* (`rcall`) instruction is similar, but places the return address (`PC + 1`) on the stack.
The *return* instruction (`ret`) returns from a function call by replacing the `PC` with the value on the stack.

There are also instructions which skip over the next instruction on some condition. For example, the instruction *skip on register bit set* (`SRBS`) skips the next instruction (increments PC by 2 or 3) if a particular bit in the designated register is set.

```
  sbrs  r1, 3   ; skips next instruction if
                ; bit 3 in r1 is 1
```

Similarly, for I/O registers:

```
  sbic  PINB, 4 ; skips next instruction if
                ; bit 4 in I/O register PINB is 1
```

There are many instructions for bit manipulation; bit rotation in a byte, bit shifting, and setting and clearing individual bits.
There are also instructions to set and clear individual bits in the status register, and to enable and disable global interrupts.
The instructions `SEI` (set global interrupt flag) and `CLI` (clear global interrupt flag) enable and disable interrupts under program control.
When an interrupt occurs, the global interrupt flag is cleared, and reset when the return from interrupt (**`reti`**) is executed.
Individual devices (e.g., timers) can also be set as interrupting devices, and also have their interrupt capability turned off. We will look at this capability later.

There are also instructions to input values from and output values to specific I/O pins, and sets of I/O pins called *ports*.
We will look in more detail at these instructions later.

## The status register (`SREG`)

One of the differences between the MIPS processor and the AVR is that the AVR uses a status register — the MIPS uses the `set` instructions for conditional branches.
The SREG has the following format:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| I | T | H | S | V | N | Z | C |

I is the interrupt flag — when it is cleared, the processor cannot be interrupted.
T is used as a source or target by the bit copy instructions, `BLD` (bit load) and `BST` (bit store).
H is used as a "half carry" flag for the BCD (binary coded decimal) instructions.
S is the sign bit. $S = N \oplus V$.
V is the 2's complement overflow bit.
N is the negative flag, set when a result from the ALU is negative.
Z is the zero flag, set when the result from the ALU is zero.
C is the carry flag, set when there is a carry from an arithmetic or logical operation.

The description of each instruction describes which flags are set as a result of executing that instruction.

## An example of program-controlled I/O for the AVR

Programming the input and output ports (ports are basically registers connected to sets of pins on the chip) is interesting in the AVR, because each pin in a port can be set to be an input or an output pin, independent of other pins in the port.

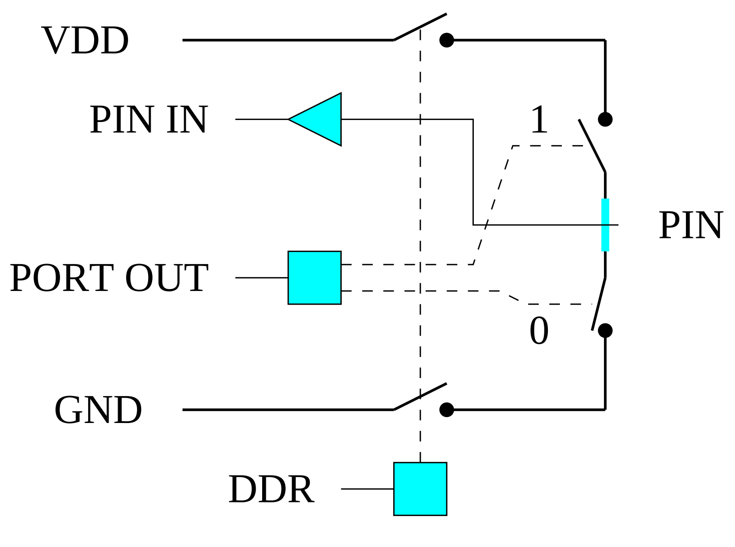
Ports have *three* registers associated with them.

The data direction register (`DDR`) determines which pins are inputs (by writing a 0 to the DDR at the bit position corresponding to that pin) and which are output pins (similarly, by writing a 1 in the DDR).

The `PORT` is a register which contains the value written to an output pin, or the value presented to an input pin.

Ports can be written to or read from.

Following is a simplified diagram of one bit of an input port:



The `PIN` register can only be read, and the value read is the value presently at the pins in the register. Input is read from a pin.

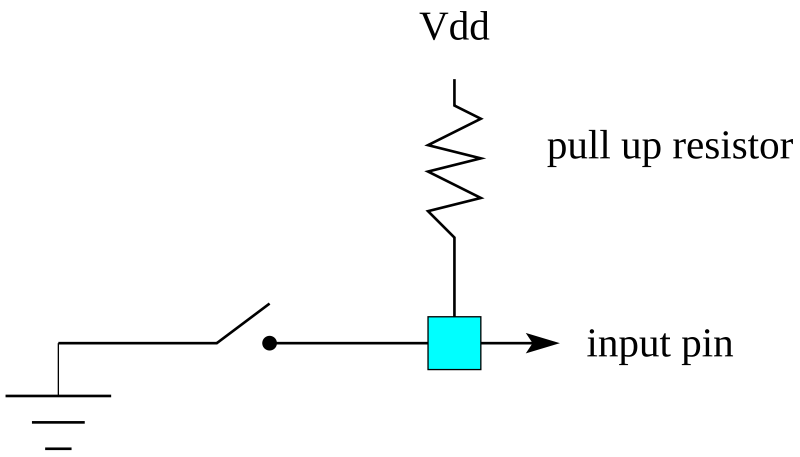
The bits in the `PORT`, `PIN`, and `DDR` registers control the direction and value of what is read or written at a particular port. The following table shows the settings to read, write, and disable a single pin in a port:

| `DDRxn` | `PORTxn` | I/O | Comment |
|---|---|---|---|
| 0 | 0 | input | disabled (tri-state) |
| 0 | 1 | input | pull-up resistor enabled |
| 1 | 0 | output | Zero (low) output |
| 1 | 1 | output | One (high) output |

In general, if `DDRxn` is 0, pin `n` of port `x` is programmed to input. `PORTxn` should be set to 1 if a value is to be read; if it is set to 0 the port is actually disconnected.
If `DDRxn` is 1, the port is programmed to output, and the value of `PORTxn` (0 or 1) is the output value.

An external value is read from a `PIN` register, a value is written to a `PORT` register.
Prior to reading a value, the corresponding PORT pin should be set to 1. Typically, `DDR` is first set to 0. Why?

The short program following shows the use of these registers to control, read, and write values to two ports.
It reads 8 bits (switch inputs) from PORT D and writes the same values to PORT C (which we will connect to a set of light emitting diodes, or LEDs.)
(Ports are designated by letters in the AVR processors.)
Normally, a *pull-up resistor* is used to keep the input pins high when the switch is open — basically, when the switch is open, nothing is connected to the pin. When the switch is closed, the pin is grounded (set to 0).
The pull-up resistors are provided in the processor. (This is what happens when the PORT is set to 1 and DDR is set to 0.)



## A simple program-controlled I/O example

The following program reads port D and writes the value to port C. It is an infinite loop, as are many examples of program controlled I/O.

```
; read port D, write contents to port C - infinite loop

#include <m1284pdef.inc>
        .org 0
; define interrupt vects
vects:
    rjmp    RESET

RESET:
        ldi R16, 0X00   ; load register 16 with zero
        out DDRD, r16   ; set port D to input (all 0's)
        ser R16         ; load register 16 to all 1's
        out PORTD, r16  ; set pullups (1's) on port D input
        out DDRC, R16   ; set port C to output

LOOP:                   ; infinite loop
        in r16, PIND    ; read port D
        out PORTC, r16  ; output to port C
        rjmp LOOP       ; repeat loop
```

Note that the definition file allows ports and registers to have the same symbolic names as in the documentation.
There is a similar file for each of the AVR devices.

This program will run as it is, but it would normally contain a `jmp` instruction for all the possible interrupt sources.
In this program, no interrupts are enabled (except `RESET`, which is always enabled) so it doesn't matter.
The same program could be written in C as follows:

```
#include <avr/io.h>

int main(void)
{
        DDRD=0x00;      // set port D to input
        PORTD=0xFF;     // set pullups
        DDRC=0xFF;      // set port C to output

        while (1)
        {
                PORTC=PIND;     // read port D and write
                                // contents to port C
        }
}
```

Again, the `include` file contains definitions which allow the registers to have the same names as in the documentation.

## Another example

The following program causes one pin of port B (PB5) to toggle when PB4 is set to 0 (say, with a switch.)
If PB5 is connected to a speaker, it will buzz.
The program reads pin 4 of port B until it finds it set to zero (the button is pressed). Then it jumps to code that sets bit 5 of port B (the speaker input) to 0 for a fixed time, and then resets it to 1. (Recall that pins are read, ports are written.)

```
#include <m1284pdef.inc>
        .org 0

; define interrupt vectors
vects:
    rjmp    reset

reset:
    ldi R16, 0b00100000  ; load register 16 to set PORTB
                         ; registers as input or output
    out DDRB, r16        ; set PORTB 5 to output,
                         ; others to input
    ser R16              ; load register 16 to all 1's
    out PORTB, r16       ; set pullups (1's) on inputs
```

```
LOOP:                    ; infinite wait loop
    sbic PINB, 4         ; skip next line if button pressed
    rjmp LOOP            ; repeat test

    cbi PORTB, 5         ; set speaker input to 0
    ldi R16, 128         ; set loop counter to 128

SPIN1:                   ; wait a few cycles
    subi R16, 1
    brne SPIN1

    sbi PORTB, 5         ; set speaker input to 1
    ldi R16, 128         ; set loop counter to 128

SPIN2:
    subi R16, 1
    brne SPIN2

    rjmp LOOP            ; speaker buzzed 1 cycle,
                         ; see if button still pressed
```

Following is a (roughly) equivalent C program:

```
#include <avr/io.h>
#include <util/delay_basic.h>

int main(void)
{
DDRB  = 0B00100000;
PORTB = 0B11111111;
while (1) {
   while(!(PINB&0B00010000))  {
      PORTB = 0B00100000;
      _delay_loop_1(128);   // from AVR libc
      PORTB = 0;
      _delay_loop_1(128);
      }
   }
return(1);
}
```

Note the use of the function `_delay_loop_1()`
This function is just a delay loop, similar to the assembler code shown previously. It is described in the header file `<util/delay_basic.h>` in the AVR LIBC library.

The main difference in these two programs is that the assembler program toggles a single bit in `PORTB` to create the sound, using the `sbi` and `cbi` assembler instructions, while the C code writes all the bits in `PORTB`.

In C, a single bit can be changed by reading the whole byte and using logic operations to change only the bit required. This is similar to what is done in the `while` statement, which "masks" the input bit being tested. See the following:

```
#include <avr/io.h>
#include <util/delay_basic.h>
int main(void)
{
DDRB  = 0B00100000;
PORTB = 0B11111111;
while (1) {
   while(!(PINB&0B00010000))  {
      PORTB |= 0B00100000;  // set pin 5 to 1
      _delay_loop_1(128);   // from AVR libc
      PORTB &= 0B11011111;  // set pin 5 to 0
      _delay_loop_1(128);
      }
   }
return(1);
}
```

Although it may seem that the assembler code should be more efficient, since it directly changes a single bit, and the C code as written requires that the port value be read and then rewritten, the C compiler is "intelligent" enough to generate a single instruction for this code.
We will see another way to generate a "bit mask" in the next C example.

Two words about mechanical switches — they bounce! That is, they make and break contact several times in the few microseconds before full contact is made or broken. This means that a single switch operation may be seen as several switch actions.
The way this is normally handled is to read the value at a switch (in a loop) several times over a short period, and report a stable value.

## The AVR libc library

The C program on the previous page contained two identical function calls, (`_delay_loop_1(128);`).
This is one of the many useful functions defined in the AVR libc library, used in avr-gcc. Actually, it is a function in the `delay_basic.h` set of functions. They are delay loops, virtually identical to the delay loop code in the assembly program example.

The function prototypes are contained in the `.h` files.
The header file `#include <util/delay_basic.h>`
defines two functions,
`_delay_loop_1((uint8_t __count);`, and
`_delay_loop_2(uint16_t __count);`.
They are counted loops, and use a fixed number of cycles per iteration (3 and 4 cycles, respectively).
Note that these are "busy wait" loops — the processor can do nothing else while these loops are running. This type of delay is often used for *small* delays.

The library also contains the files which define the register values, etc. associated with a particular processor. The definitions are found in
`#include <avr/io.h>`
Actually, this file includes other `include` files like
`#include <avr/portpins.h>`

This library contains many useful low-level functions, and we will use it frequently in the future.

**Interrupts in the AVR processor**

The AVR uses vectored interrupts, with fixed addresses in program memory for the interrupt handling routines.

Interrupt vectors point to low memory; the following are the locations of the memory vectors for some of the 31 possible interrupt events in the ATmega1284p:

| Address | Source | Event |
|---|---|---|
| `0X000` | RESET | power on or reset |
| `0X002` | INT0 | External interrupt request 0 |
| `0X004` | INT1 | External interrupt request 1 |
| `0X006` | INT2 | External interrupt request 2 |
| `0X008` | PCINT0 | pin change interrupt request 0 |
| `0X00A` | PCINT1 | pin change interrupt request 1 |
| `0X00C` | PCINT2 | pin change interrupt request 2 |
| `0X00E` | PCINT3 | pin change interrupt request 3 |
| `0X010` | WDT | Watchdog Timer |
|  | . | . |
|  | . | . |

Interrupts are prioritized as listed; `RESET` has the highest priority.

Normally, the instruction at the memory location of the vector is a `jmp` to the interrupt handler. (In processors with 2K or fewer program memory words, `rjmp` is sufficient.)

In fact, our earlier assembly language program used an interrupt to begin execution — the `RESET` interrupt.

## Reducing power consumption with interrupts

The previous program only actually did something interesting when the button was pressed. The AVR processors have a "sleep" mode in which the processor can enter a low power mode until some external (or internal) event occurs, and then "wake up" an continue processing.

This kind of feature is particularly important for battery operated devices.

Pages 42 to 44 of the ATmega1284p datasheet describe the six sleep modes in detail.

(Tiny processors have fewer sleep modes.)

Briefly, the particular sleep mode is set by placing a value in the Sleep Mode Control Register (`SMCR`). For our purposes, the *Power-down* mode would be appropriate (bit pattern `0b0000010`) but the simulator only understands the *idle mode* (`0b0000000`).

The low order bit in `SMCR` is set to 1 or 0 to enable or disable sleep mode, respectively.

The `sleep` instruction causes the processor to enter sleep mode, if bit 0 of `SMCR` is set.

The following code can be used to set sleep mode to *idle* and enable sleep mode:

```
ldi R16, 0b00000000 ; set sleep mode idle
out SMCR, R16       ; for power-down mode, write 00000010
```

## Enabling external interrupts in the AVR

Pages 67–68 of the ATmega1284p datasheet describe in detail the three I/O registers which control external interrupts. General discussion of interrupts begins on page 61.

We will only consider one type of external interrupt, the *pin change* interrupt. There are 32 possible `pcint` interrupts, labeled `pcint0` to `pcint31`, associated with `PORTE[0-7]` and `PORTB[0-7]`, respectively.

There are four `PCINT` interrupts, `PCINT0` for `pcint` inputs 0–7, and `PCINT1` for `pcint` inputs 8–15, etc.

We are interested in using the interrupt associated with a push-button switch connected to `PINB[4]`, which is `pcint12`, and therefore is interrupt type `PCINT1`.

Two registers control PCI interrupts, the *Pin Change Interrupt Control Register* (`PCICR`), and the *Pin Change Interrupt Flag Register* (`PCIFR`).

Only bits 0, 1, 2, and 3 are defined for these registers.

Setting the appropriate bit of register `PCICR` (in our case, bit 1 for `PCINT1`) enables the particular pin change interrupt.

The corresponding bit in register `PCIFR` is set when the appropriate interrupt external condition occurs.

A pending interrupt can be cleared by writing a 1 to this bit.

Normally, we would expect to be able to enable `PCICR` with

```
sbi PCICR, PCIE1     ; enable pin change interrupt 1
```

but this will not work because the `PCICR` register is one of the extended I/O registers, and must be written as a memory location.
The following code enables *pin change interrupt 1* (`PCINT1`) and clears any pending interrupts by writing 1 to bit 7 of the respective registers:

```
sbi PCIFR, PCIF1     ; clear pin change interrupt flag 1
ldi r28, PCICR       ; ; load address of PCIFR in Y low
clr r29              ; load high byte with 0
sbr r16, 0b00000010
st  Y, r16           ; enable pin change interrupt 1
```

There is also a register associated with the particular pins for the `PCINT` interrupts. They are the *Pin Change Mask Registers* (`PCMSK1` and `PCMSK0`).
We want to enable the input connected to the switch, at `PINB[4]`, which is `pcint12`, and therefore set bit 4 of `PCMSK1`, leaving the other bits unchanged.
Again, this register is one of the extended I/O registers, and must be written to as a memory location.

The following code sets up its address in register pair `Y`, reads the current value in `PCMSK1`, sets bit 4 to 1, and rewrites the value in memory:

```
ldi r28, PCMSK1      ; load address of PCMSK1 in Y low
clr r29              ; load high byte of Y with 0
ld r16, Y            ; read value in PCMSK1
sbr r16,0b00010000   ; allow pin change interrupt on
                     ; PORTB pin 4
st Y, r16            ; store new PCMSK1
```

Now, the appropriate interrupt vectors must be set, as in the table shown earlier, *and* interrupts enabled globally by setting the `interrupt (I)` flag in the status register (`SREG`).
This later operation is performed with the instruction `sei`.
The interrupt vector table should look as follows:

```
       .org 0
vects:
     jmp    RESET         ; vector for reset
     jmp    EXT_INT0      ; vector for int0
     jmp    EXT_INT1      ; vector for int1
     jmp    EXT_INT2      ; vector for int2
     jmp    PCINT0        ; vector for pcint0
     jmp    PCINT1        ; vector for pcint1
      ...
```

The next thing necessary is to set the stack pointer to a high memory address, since interrupts push values on the stack:

```
ldi r16, 0xff        ; set stack pointer
out SPL, r16
ldi r16, 0x04
out SPH, r16
```

After this, interrupts can be enabled after the I/O ports are set up, as in the program-controlled I/O example.

Following is the full code:

```
#include <m1284pdef.inc>
       .org 0
VECTS:
     jmp    RESET        ; vector for reset
     jmp    EXT_INT0     ; vector for int0
     jmp    EXT_INT1     ; vector for int1
     jmp    EXT_INT2     ; vector for int2
     jmp    PC_INT0       ; vector for pcint0
     jmp    PC_INT1       ; vector for pcint1

EXT_INT0:
EXT_INT1:
EXT_INT2:
PCINT0:
       reti

RESET:
                           ; set up pin change interrupt 1
       ldi r28, PCMSK1      ; load address of PCMSK1 in Y low
       clr r29              ; load high byte with 0
       ld r16, Y            ; read value in PCMSK1
       sbr r16,0b00010000   ; allow pin change interrupt on portB pin 4
       st Y, r16            ; store new PCMSK1

       ldi r28, PCICR
       sbr r16, 0b00000010
       st  Y, r16
       sbi PCIFR, PCIF1     ; clear pin change interrupt flag 1

       ldi r16, 0xff        ; set stack pointer
       out SPL, r16
       ldi r16, 0x04
       out SPH, r16

       ldi R16, 0b00100000  ; load register 16 to set portB registers
       out DDRB, r16        ; set portB 5 to output, others to input
       ser R16              ;
```

```
        out PORTB, r16       ; set pullups (1's) on inputs

        sei                  ; enable interrupts

        ldi R16, 0b00000010  ; set sleep mode
        out SMCR, R16
        rjmp LOOP

PC_INT1:
       reti
       rjmp LOOP

SNOOZE:
       sleep
LOOP:
       sbic PINB, 4         ; skip next line if button pressed
       rjmp SNOOZE          ; go back to sleep if button not pressed

       cbi PORTB, 5         ; set speaker input to 0
       ldi R16, 128         ;

SPIN1:                      ; wait a few cycles
       subi R16, 1
       brne SPIN1

       sbi PORTB, 5         ; set speaker input to 1
       ldi R16, 128

SPIN2:
       subi R16, 1
       brne SPIN2

       rjmp LOOP            ; speaker buzzed 1 cycle,
                            ; see if button still pressed
```

Following is a (roughly) equivalent C program:

Note the use of the function `_BV(arg)`. This function sets a single bit, the bit in position `arg` to 1. E.g. `_BV(3)` sets bit 3 to 1.
It is exactly equivalent to `1<<arg`.
The labels for the bits in the AVR documentation are defined in the include files for the particular target processor, so the code can match the labels in the documentation.
Note also that code uses several more functions from the AVR libc library.
The interrupt functions are defined in the files `<avr/interrupt.h>`
There are many different interrupt types, and the documentation provides the arguments for all of them.
The AVR hardware disables further interrupts (by clearing the global interrupt flag in SREG) when an interrupt occurs.
The compiler automatically enables interrupts when the interrupt handler function returns.
This means that code inside the interrupt handler function cannot be interrupted — normally interrupts do not "nest."
It is possible (but seldom advisable) to "nest" interrupts, by explicitly enabling interrupts in the interrupt handler, using the function `sei()`.
Interrupt routines are usually made to be as fast (small) as possible, to minnimize the time interrupts are disabled.
The interrupt handler we use in this example contains no code at all!

The sleep functions are defined in the files `<avr/sleep.h>`

```
#include <avr/io.h>
#include <util/delay.h>
#include<avr/interrupt.h>
#include<avr/sleep.h>

void buzz( void );

int main(void)
{
// set up pin change interrupt 1

   PCMSK1 = 0b00010000;

   PCIFR  |= _BV(PCIF1); // clear pin change interrupt flag
   PCICR  |= _BV(PCIE1); // enable pin change interrupt 1
   sei();                // enable global interrupts

   DDRB  = 0B00100000;   // set PORTB 5 to output
   PORTB = 0B11111111;   // set pullups (1's) on inputs

   set_sleep_mode(SLEEP_MODE_PWR_DOWN);
   while (1) {
      sleep_mode();
      buzz();   // buzzer code from earlier program
   }
   return(1);
}

void buzz( void )
{
   while(!(PINB & 0B00010000))  {
      PORTB |= _BV(PB5);
      _delay_loop_1(128);
      PORTB &= ~(_BV(PB5));
      _delay_loop_1(128);
   }
}

ISR(PCINT1_vect)
{
}
```

## A digression — preparation for the lab

In previous lectures, we have looked at programming the ports, and individual bits in the ports, as input and output devices.
Essentially, a port consists of three registers, the `DDR`, which determines whether each bit is input or output, the `PORT` register, which is written to for output, and the `PIN` register, which is read for input.
The `PORT` register also enables or disables pullup resistors in input mode.
We now want to look at some simple devices we can connect to these ports. In the first lab, we used the switches and LEDs on the STK-500 board, and we would like now to interface to some simple devices off this board.

We will use two devices, (as well as the push button switches on the board) — a "7 segment LED" display, and a network of resistors.

First, we will look at some aspects of the C language that will be useful.

In C, it is typical to use a *header file* to contain C declarations, function prototypes, macros definitions, and other things.
The usual convention is to give header files names that end with `.h`
User defined header files are often used to also collect the system header files.
For lab 1, we can use it to define some labels that relate the hardware connections to the software we write. Assume that we connect pin 0 of a port to LED segment A of a 7 segment display, pin 1 to segment B, etc.
We could use the following to declare this; usually in a header file, say file `test.h`:

```
#include <avr/io.h>


#define G       0b00000001
#define A       0b00000010
#define F       0b00000100
#define B       0b00001000
#define E       0b00010000
#define C       0b00100000
#define D       0b01000000
#define DP      0b10000000
```

Note that if we use these definitions to construct our outputs, and change the wiring for some reason, all we need to do is change the definition appropriately in the header file.

Now, we can define the 7 segment display output for the numbers 0 (ZERO) and 1 (ONE) (say, in the main program, `test.c`) as

```
#include "test.h"

int main (void)
{
    ZERO = A|B|C|D|E|F;
    ONE = B|C;
        .
        .
        .
}
```

In general, it is a good idea to logically separate the hardware and software aspects of a design. Typically, we do this in a header file.

Header files are copied into the file into which they are included. `define`s are treated as straight text substitution, and consequently do not become part of the executable program.

Although we will not look in detail at its construction (yet), the lab contains a link to a `Makefile` that will compile a C program called `test.c`. It will also write the program to flash memory.

## Types, in C

Unlike Java, the size (bit representation) of numbers and other types in C is not explicitly defined. Minimum sizes *are* defined, and type `int` has a minimum of 16 bits.

There are integer types with exact sizes in C, defined in the file `<stdint.h>`
We will often use the 8-bit exact size type, `int8_t`
Like other inter types, it there is an unsigned variant, `uint8_t`

We use the explicit 8 bit integers whenever possible, because the processor has an 8 bit word length, so larger bit representations take more instructions to do an arithmetic or store operation.

Sometimes we also use the explicit 16 bit integers, `int16_t` and `uint16_t` as well, and they are typically implemented with the register pair instructions, if possible.

The next slides (and much of this lecture) will review simple circuit theory, and show several uses for these simple circuit elements.

## Basic circuit elements — resistors

Electrical circuits can be modeled by a small number of “ideal” components. One of the simplest and most useful of these is the *resistor*. In some ways, electrical circuits can be modeled by fluid (hydraulic) systems, and this may provide a useful visual model for simple circuits.

The basic parameters for an electrical circuit are *current (I)* and *voltage (V)*.

Current is the rate of flow of *charge (Q)* — in good conductors (metals) the charge carriers are electrons.

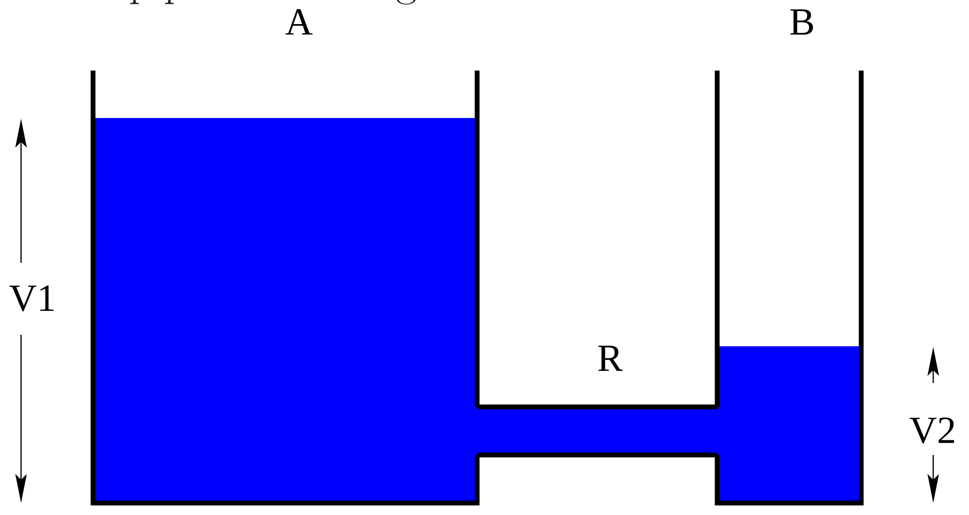
$$I = dQ/dt$$

Voltage is a measure of potential energy.

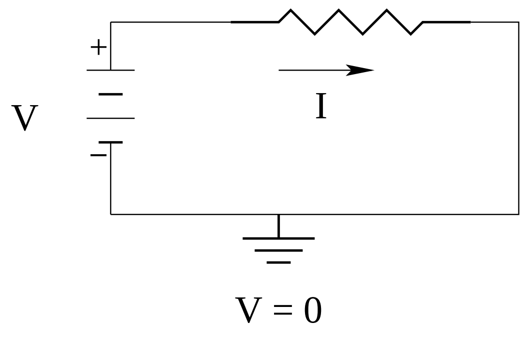
A potential energy (voltage) *difference* causes current to flow from one point to another.

The rate at which the charge flows (current) is determined by the potential energy difference, and by the resistance in the circuit.

In a hydraulic model, consider two containers partly full of water. Water will flow from the reservoir with the higher potential (V1) to the lower potential (V2).
The rate at which the water will flow depends on the size (resistance) of the pipe connecting the two reservoirs.



Following is an equivalent electrical circuit:



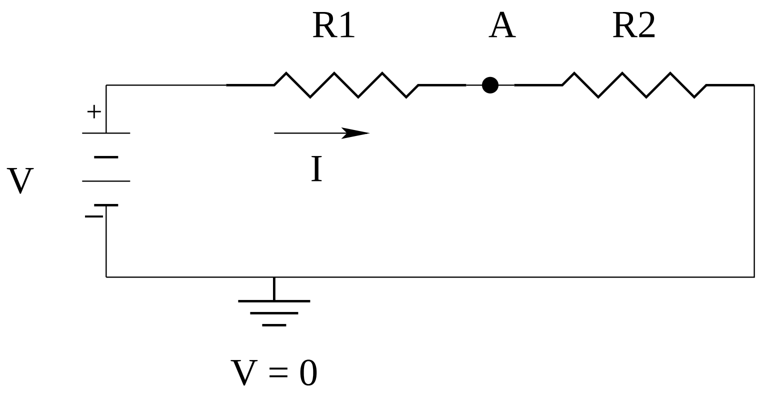
The resistance, $R$, is defined as

$$R = \frac{V}{I}$$

This relationship (written as $V = I \times R$) is called **Ohm's law**.
Note that $v$ is a *potential difference*, and one terminal of a circuit is usually taken as having a potential of 0 volts.

## Fun with resistors

Suppose we have a circuit with two resistors in series:



What is the total resistance here?

$$R_{total} = R1 + R2$$

In fact, for any resistors in series, the total resistance is the sum of the resistances in series.
The current flowing in each of the resistors in series is the same, and is found from Ohm's law as

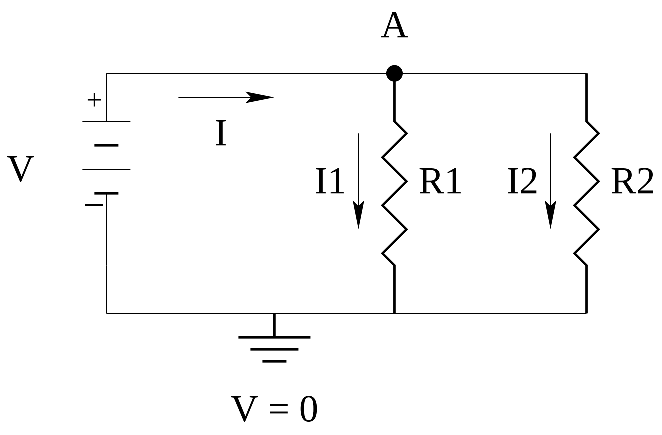
$$I = \frac{V}{R_{series}}$$

If R1 and R2 are identical, what is the voltage at node A?

How about if R2 = 2 $\times$ R1?
In general, how could the voltage at node A be determined?

$$V_A = I \times R2$$

Suppose the resistors are arranged in a parallel configuration:



What is the total resistance in the circuit now?

Suppose $R1 = R2$. Then the current in both would be the same, and $I1$ and $I2$ would each be $I/2$.

In general, $I1 = V/R1$, and $I2 = V/R2$.
Also, $I = I1 + I2$. (At node A, the current flowing into the node must equal the current flowing out of the node.)

If there is a single resistance (say, $R_{parallel}$) equivalent to the this network, then

$$V/R_{parallel} = V/R1 + V/R2$$

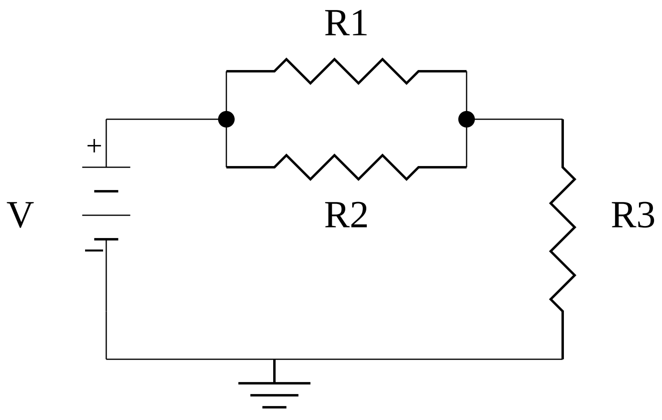
We therefore have the relationship

$$1/R_{parallel} = 1/R1 + 1/R2$$

Clearly, this generalizes for any parallel combination of resistors.

## Series and Parallel Equivalent circuits

Clearly, in analyzing a circuit, we can replace parallel and series resistance combinations with equivalent resistances.
We can use this idea to simplify more complex circuits:



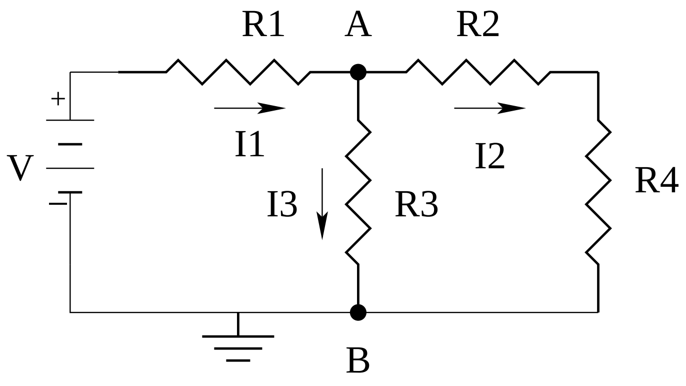
Here, the parallel components, R1 and R2, can be replaced by their equivalent resistance, $(R1 \times R2/(R1 + R2))$
This can then be added to the series resistance, R3.
The net result is a single equivalent resistance.

Can this be done for all circuits containing only resistors and a voltage source?

No, so how can more complex circuits be analyzed?

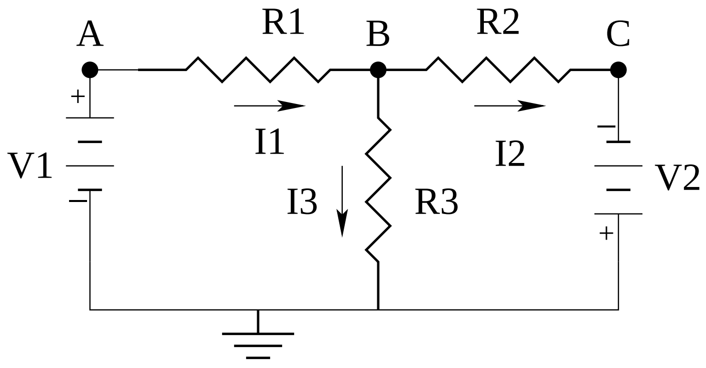
## Analyzing complex circuits — Kirchoff's laws

There are two useful laws for analyzing an electrical circuit, both based on conservation laws in physics.
The first states that at any node in a circuit (a node is where two or more components are connected) the current flowing into the node is exactly equal to the current flowing out of the node. This follows from the *conservation of charge.*
In the following, at node A, $I1 = I2 + I3$

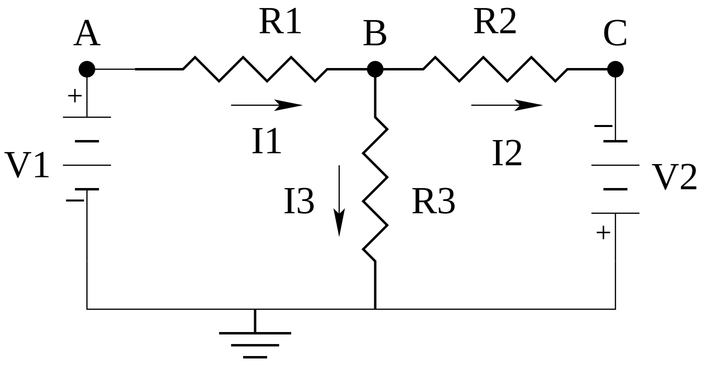


where I1 is the current through R1, I2 is the current through R2 and R4, and I3 is the current through R3.
I1 flows into node A, and I2 and I3 flow out of node A.

This law applies even if there are several voltage sources:



The second law states that, in any closed loop in a circuit, the net voltage difference is always zero. (Since voltage is potential energy, this is simply conservation of energy.)
It is often stated as "the sum of the voltages in a closed loop is zero."



Here there are three possible loops; considering only the outer loop, and traversing the loop in a clockwise direction,

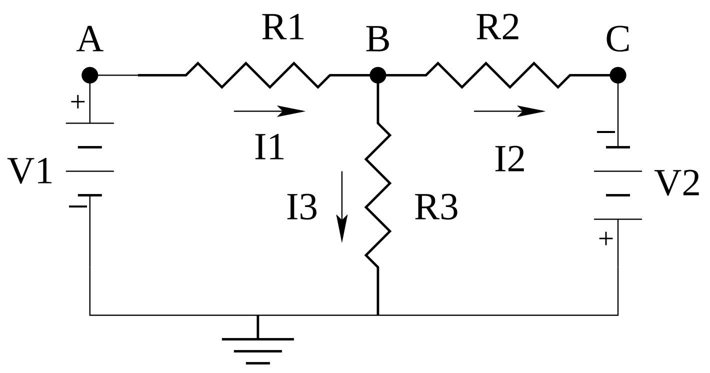
$$-V1 + I1 \times R1 + I2 \times R_2 - V2 = 0$$

Note that the sign here is the sign from the sources in the order in which they are encountered in the traversal of the loop.
If we use either of the inner loops, we can use the fact that $I3 = I1 - I2$.

Either one of these laws can be used to derive a general method for solving for voltages and/or currents in a circuit.

## Node Analysis

Consider the following circuit:



Let R1 be 2K ohms, R2 be 3K ohms, and R3 be 1K ohms, V1 be 3 Volts, and V2 be 10 volts.

Node A is at 3V, node C is at -10V, and only node B is unknown.

We wish to solve for the currents, $I1$, $I2$, and $I3$.

Of course, we need only find any two of those, since $I1 = I2 + I3$.

From Ohm's law:

$I1 = (V1 - VB)/R1 = (3 - VB)/2000$

$I2 = (VB - V2)/R2 = (VB - (10))/3000$

$I3 = VB/R3 = VB/1000$

Since, at node B, $I1 = I2 + I3$, we can write a single equation for the unknown voltage at VB as

$(3 - VB)/2000 = (VB - (10))/3000 + VB/1000$

or VB = -1.0V

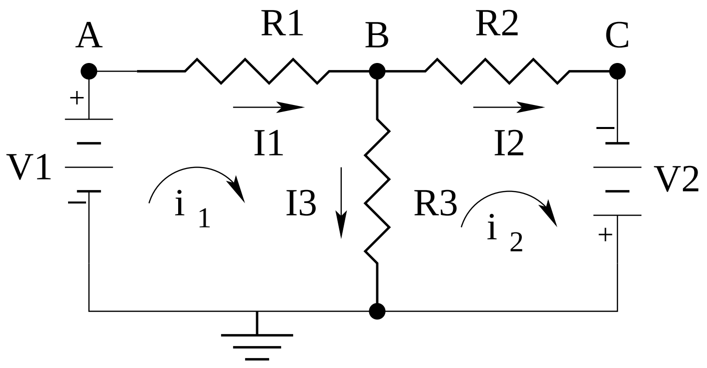
The currents can be found by substituting for VB, giving

$I3 = VB/1000 = -0.001$ amp $= 1$ ma

similarly, $I2 = 3$ ma and $I1 = 2$ ma

## Mesh Analysis

Consider the same circuit:



Again, R1 is 2K ohms, R2 is 3K ohms, and R3 is 1K ohms, V1 is 3 Volts, and V2 is 10 volts.

Here we will use the second law, and consider closed loops (meshes). We will define two mesh currents, $i_1$ and $i_2$, and write equations for each mesh:

$-V1 + i_1 \times R1 + (i_1 - i_2) \times R_3 = 0 = -3 + 2000i_1 + 1000(i_1 - i_2)$

$-(i_1 - i_2) \times R_3 + i_2 \times R2 = 0 = -1000(i_1 - i_2) + 3000i_2$

This gives us two equations in two unknowns, but they are simple linear equations and can be solved easily.

Both these techniques can be applied to circuits of any complexity, and result in simple sets of linear equations.

Even more importantly, both creating the equations to be solved and the actual solving of the equations can be done numerically with relatively simple code.

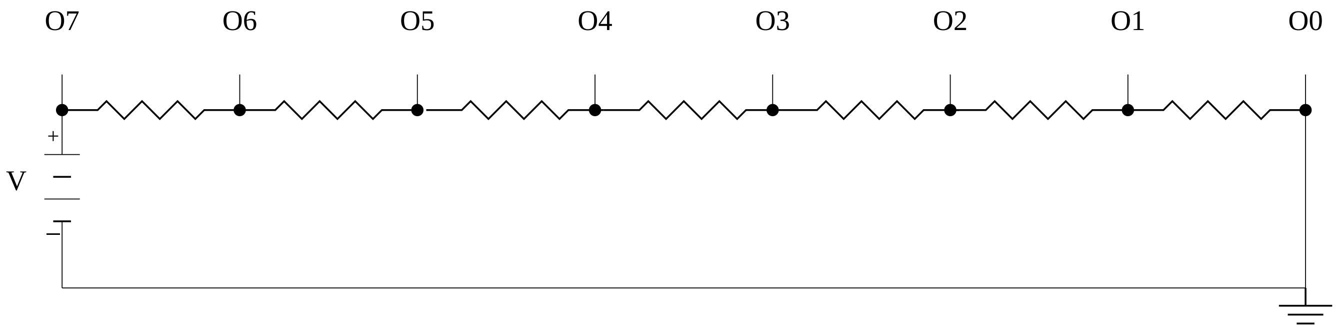
(Insert commercial for CS3731 here)

## Using resistors in circuits

Resistors are used in several ways in simple circuits.
We will look at two applications.

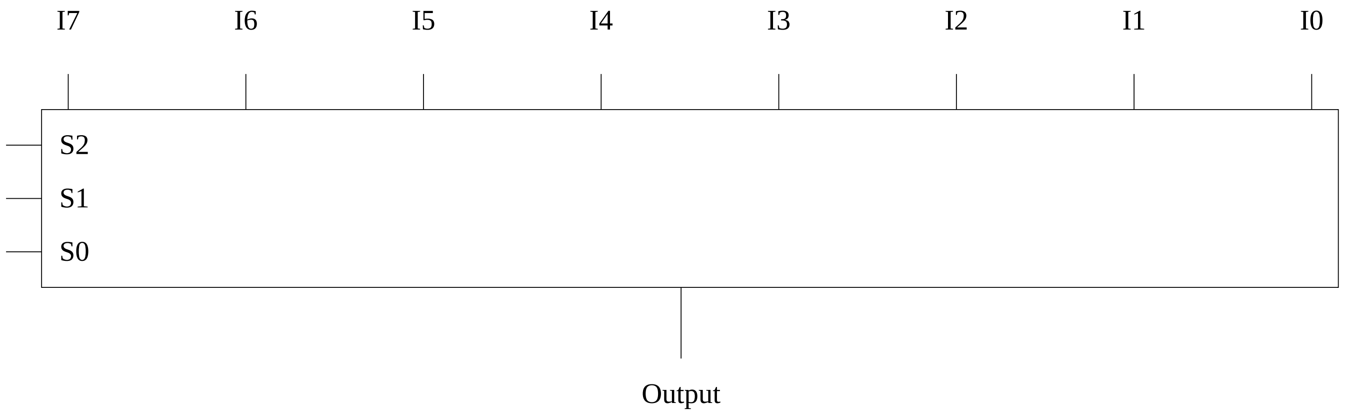
### 1. A voltage divider:

Suppose we want to divide a voltage into $n$ steps, for example, 8 steps.
Consider the following possibility:

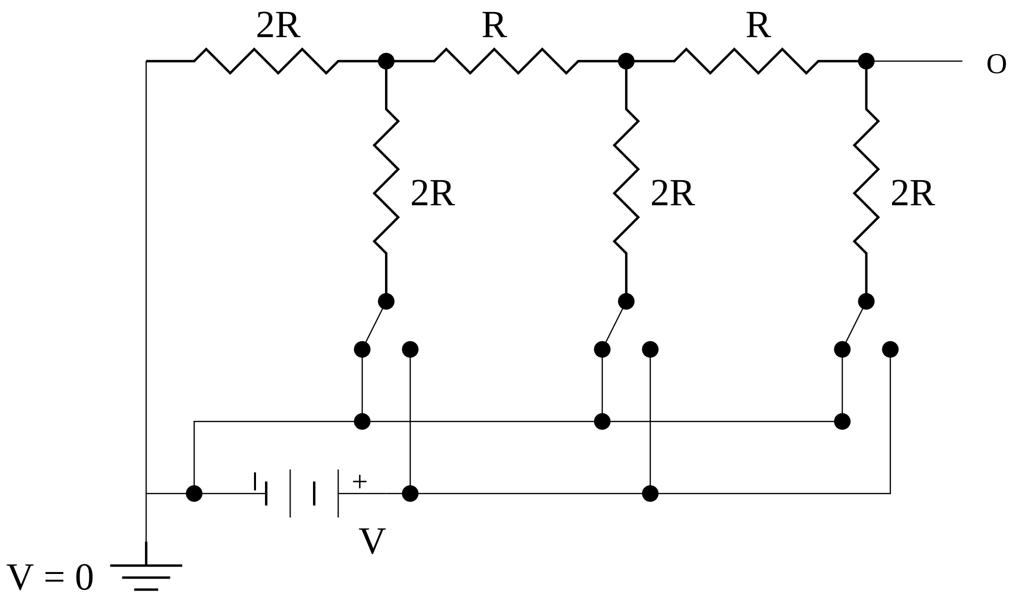


What is the voltage at each of the outputs O0 to O7?

Suppose we had an analog MUX (multiplexor) connected to the outputs O0 to O7. What would the output be then?



Consider the following:



What is the output when the switches are as shown?

When all are switched the other way?

When any one is set “on” (to the right)?

Consider what happens when the center leg is connected to V, and the other legs are connected to ground.

What happens if the resistances are not exactly equal? Suppose there is a 10% variation in the resistors.

## 2. The second common application is a current limiter:

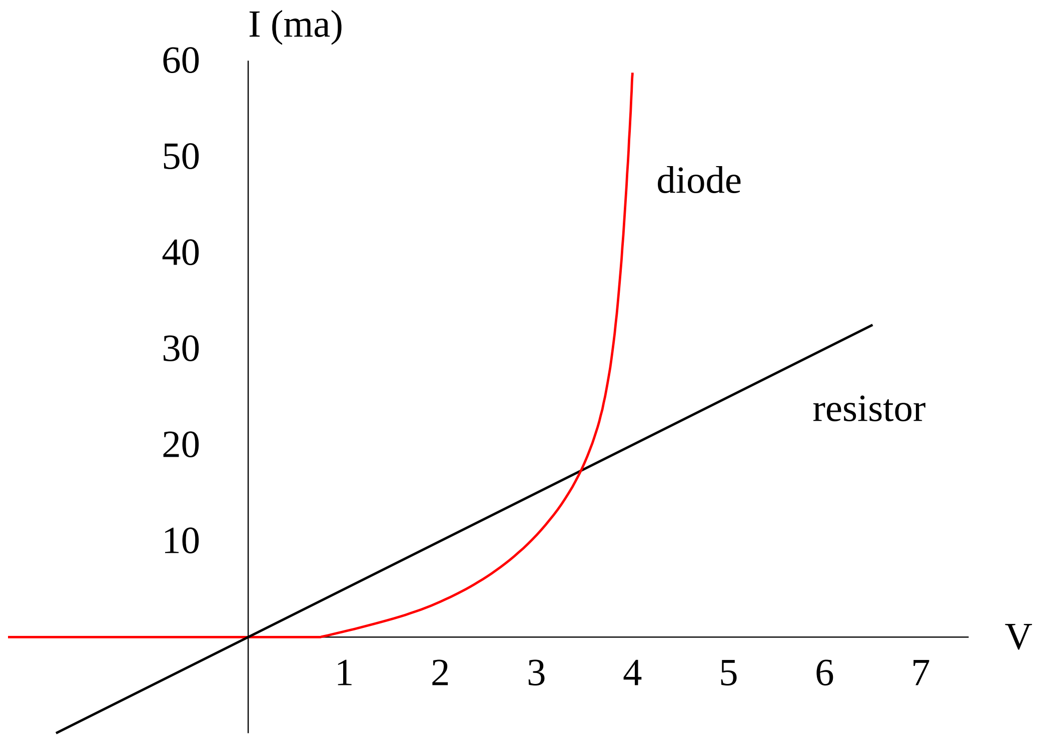
Many devices (e.g., a light emitting diode, or LED) have a very non-linear IV characteristic, and can burn out if too much current is applied. A resistor is commonly used to limit the current in such situations.

E.g., consider a LED which we want to supply with 20 ma. current to get an appropriate amount of brightness, and a 4.5 V battery circuit is available.

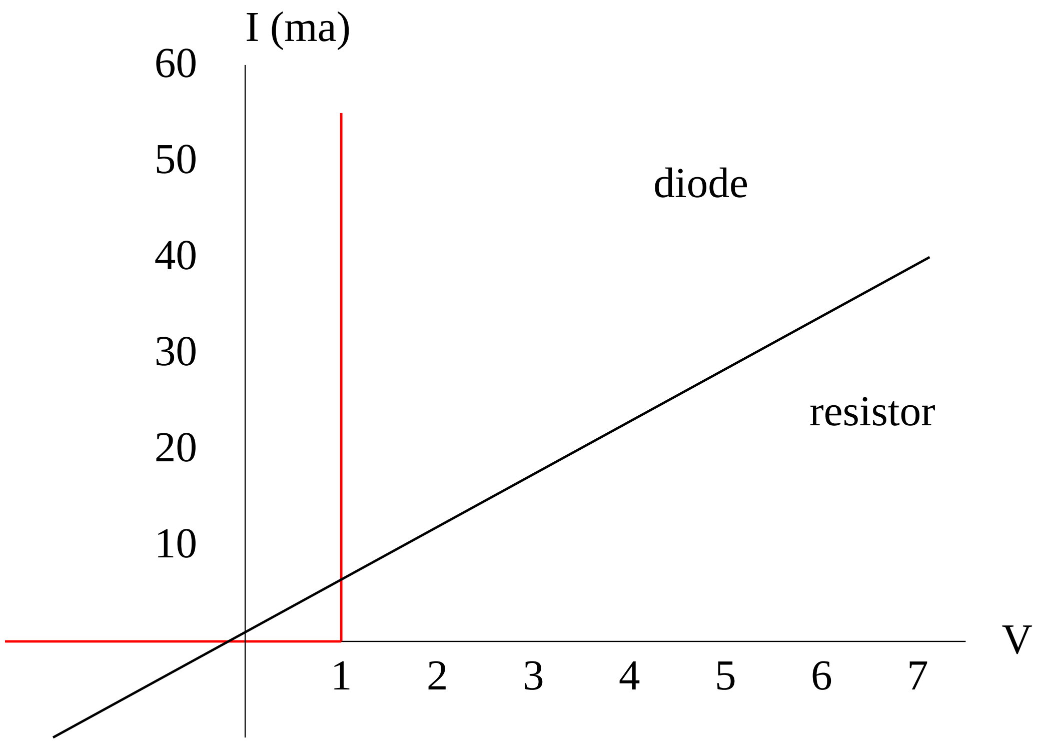
What resistance is required to limit the current to 20 ma?

Using Ohm's law, $R = V/I = 4.5\text{V}/(20 \times 10^{-3}\text{A}) = 225$ ohms

Following is a plot of a typical diode and resistor IV characteristic (the resistance is the reciprocal of the slope of the graph):



Looking at the diode characteristic, it can crudely be approximated as allowing no current to flow below a conduction threshold (about 1 volt, in the previous diagram), and then allows current to flow freely above the conduction threshold.
Essentially, the diode is modeled as having infinite resistance below the conduction threshold, and zero resistance above.



To better calculate the resistance required to limit the current, we subtract the conduction threshold voltage from the applied voltage.

In the previous example, assuming a conduction voltage threshold of 1 volt, the required resistance to limit the current to 20 ma. is

$$R = V/I = 3.5\text{V}/(20 \times 10^{-3}\text{A}) = 175 \text{ ohms}$$

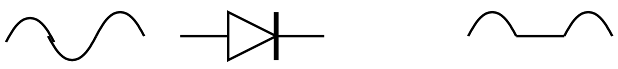
For silicon diodes, the conduction threshold is typically 0.5 to 0.8V.

## The diode

The characteristic curve for a diode on the previous page shows that for a negative voltage, the diode conducts no current, and for a positive voltage above a certain threshold, the diode is a very good conductor.

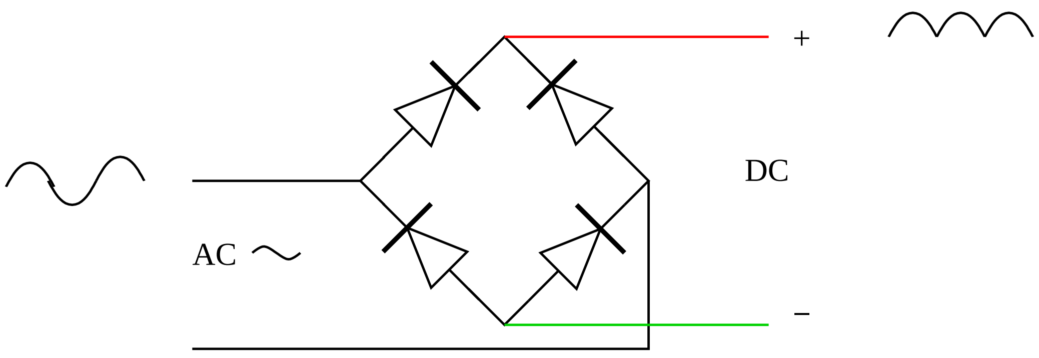
A diode is normally part of a circuit, and the remainder of the circuit (the "load") limits the amount of current flowing through the diode.

One common use of a diode is "rectification" — converting AC power to DC. A single diode will allow the positive half cycle to pass.



Rectification is commonly done with four diodes, in a configuration known as a "bridge rectifier" or "full wave rectifier." Basically, the diodes are arranged so that current can only flow in one direction.

For a sinusoidal input, the output "flips" the negative half of each cycle.

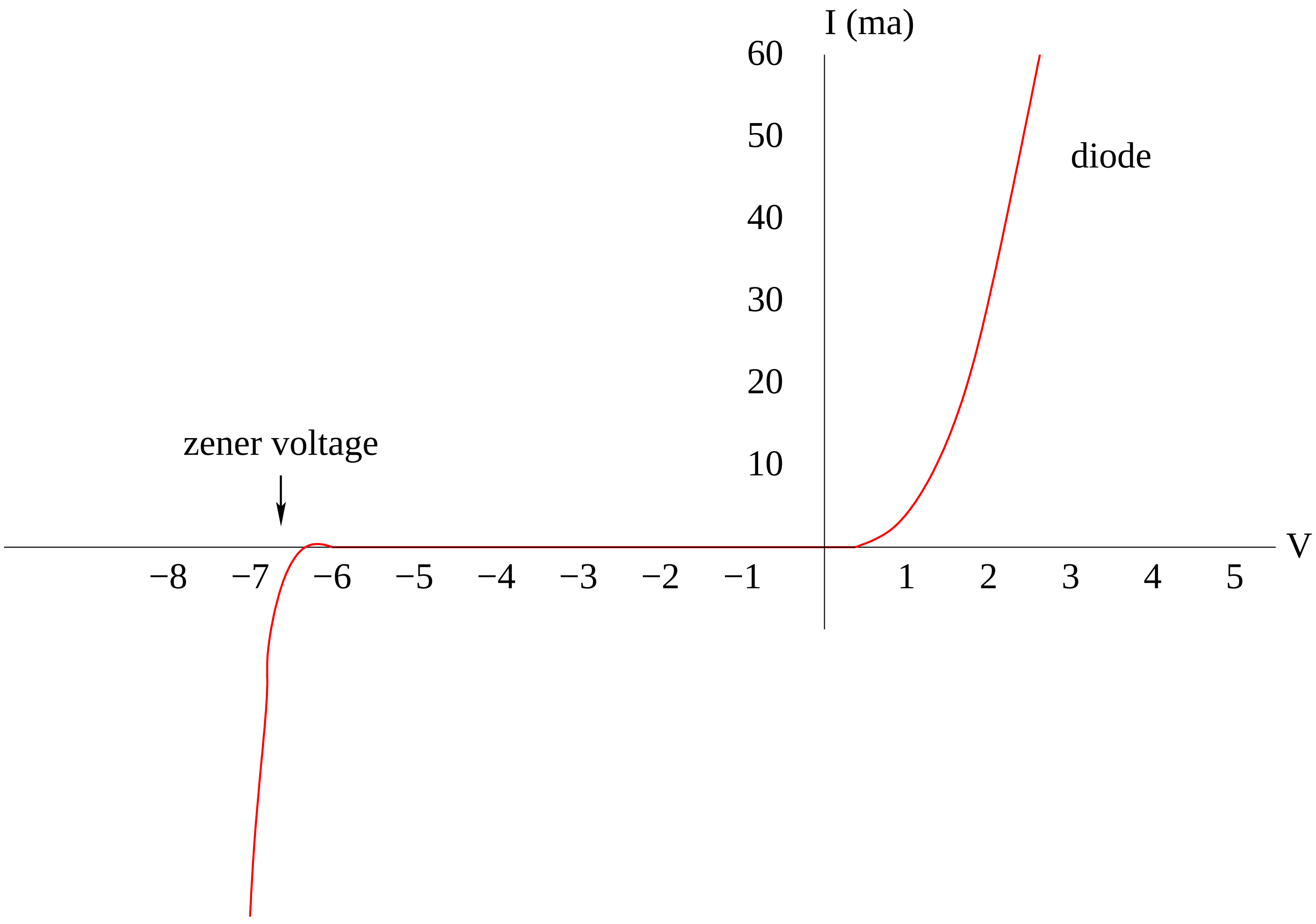


Occasionally, bridge rectifiers are used to protect against connecting a DC supply to a circuit "backwards."

The STK-500 uses a bridge rectifier for this purpose.

## Diodes characteristics

Diodes used for rectification are designed to handle large currents, and high reverse voltages. When the reverse voltage exceeds some threshold, however, a diode will "break down" and become highly conductive. The voltage at which this happens is called the *zener voltage* for the diode.



For rectifiers, the zener voltage is often several hundred volts.

## Light emitting diodes (LEDs)

LEDs are designed, not for their capability as a rectifier, but rather for the efficiency with which they convert electrical energy to light. LEDs are generally not very good rectifiers; they have low zener voltages, and a high conduction threshold.
Typically, LEDs can break down with reverse voltage as low as 5 V.

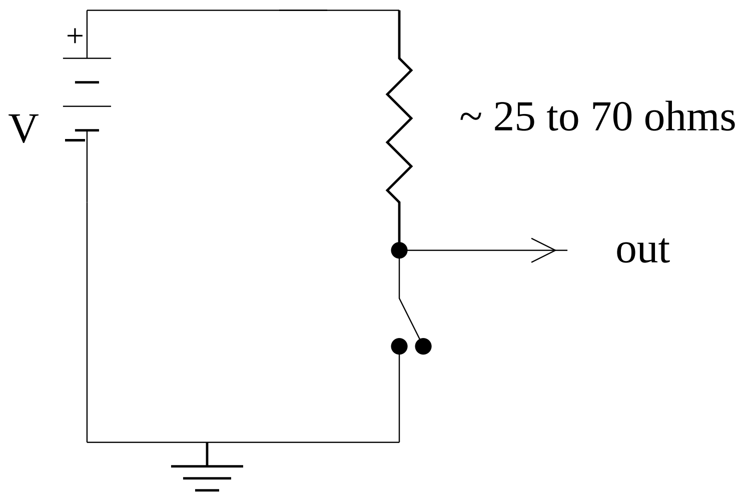
For red and yellow LEDs, the conduction threshold is typically around 2V. For green and blue (and white) LEDs, it is higher, typically around 3 to 4 volts.

In recent years, LEDs have become much more efficient, and are becoming increasingly important for illumination as well as their traditional role as displays.

The fact that diodes can be fabricated to have different (fixed) zener voltages is useful. In fact, diodes with different, known, zener voltages are fabricated as “zener diodes.” The voltage drop across such diodes (suitably current limited) can be used as a reference voltage.

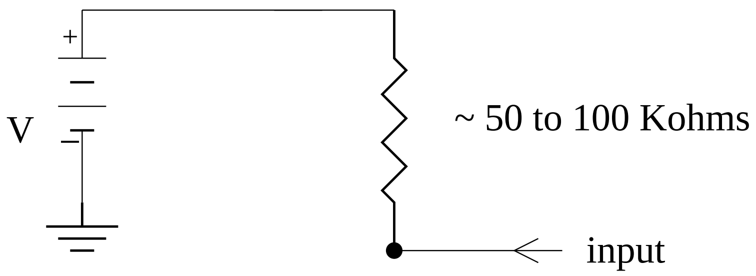
## Modeling input pins and output ports

The output ports in the AVR processors have “pull up resistors” which limit the current a port can supply.
Following is (an inexact) model of an output port on an AVR processor:



Actually, it models the output of logic 1, but incorrectly models the output of logic 0.

The model of an input pin is simpler:



## Real resistors and voltage sources

In reality, there are no "ideal" voltage sources; all have an internal resistance (although it may be quite small). For our purposes, a conventional battery will be "ideal enough."

Resistors are available as fixed resistors, sold with a particular accuracy, or tolerance, (usually 5% or 10%).

Variable resistors are also available, usually as 3 terminal devices, called potentiometers, or "pots."
These are usually adjusted by turning a dial or screw, and can be either single turn or multi-turn.
Common axial resistors are color coded; see
`http://en.wikipedia.org/wiki/`
`Resistor#Four-band_axial_resistors`
(all on one line) for a description of the color coding.

## How are voltage and current measured?

Factoid: An electrical current induces a magnetic field about a coiled wire.
This property can be used to construct a “galvanometer” — a device which measures small electrical current.
Basically, a current passes through a coil of wire, creating a magnetic field, which displaces a small magnet (like a compass needle).
In a more sophisticated variation, the coil is placed in a magnetic field (e.g., between two permanent magnets) and the magnetic force on the coil is measured. Typically, the coil is held by a spring, and the force is measured by the compression of the spring.

To measure voltage, a high resistance is placed in series with the galvanometer, in order to ensure that there is a low current in the coil.

To measure current, a low resistance shunt is placed across the terminals of the coil, so that only a small part of the current flows through the galvanometer.

We will look at how modern digital meters operate later in the course, but the lab illustrates one of the fundamental ideas behind digital meters.

## How much power does a circuit consume?

In general, the power consumed by a circuit is

$$Power = V \times I$$

In a DC circuit, this is equivalent to

$$Power = V^2/R = I^2R$$

Note that for a fixed voltage, increasing resistance reduces power consumption.

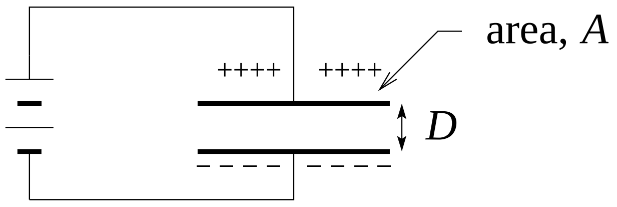
## The capacitor — capacitance, C

$$C = \frac{Q}{V}\,, \qquad Q = CV$$

where $Q$ = quantity of charge, and
$V$ = "voltage", or potential.
The capacitance C is a geometrical, or configurational, property of the capacitor. For a parallel plate capacitor,

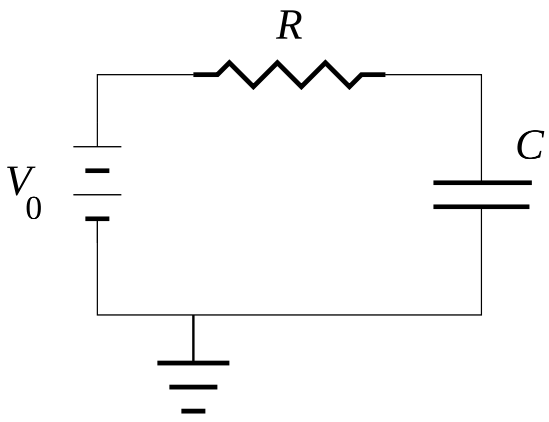
$$C = \epsilon \frac{A}{D}$$

where $A$ = area of the plate, and $D$ = distance between plates.
$\epsilon$ is a constant which depends on the material between the plates.



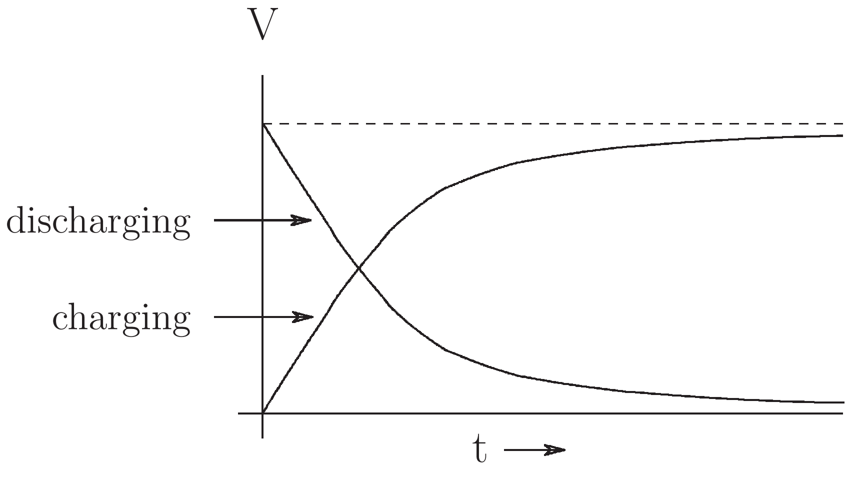
A capacitor can *store* charge on its plates.

## Capacitor in a circuit — charging through a resistor



Voltage across capacitor is

$$\begin{aligned}
V &= \frac{Q}{C}, \qquad \text{also} \quad V = IR \\
\frac{dV}{dt} &= \frac{1}{C}\frac{dQ}{dt} \qquad = \frac{1}{C}I \\
&= \frac{1}{C}\frac{V}{R} \\
\int \frac{dV}{V} &= \int \frac{1}{RC}dt \qquad \text{so} \quad V = V_0[1 - e^{-t/RC}]
\end{aligned}$$



## Use of capacitors

Capacitors have a number of uses in microelectronic circuits.
Perhaps the simplest is as a local storage of charge (current). Typically, a processor is *decoupled* from fluctuations in the power supply by a capacitor near the voltage supply pin for the device.
Generally, a 0.1–1 $\mu$f capacitor is connected to the power supply pin of a processor chip in order to supply current to the device when the internal transistors switch (on clock edges).

It is not difficult to calculate the minimum capacitance required for this decoupling if you know the switching current required, the allowable voltage drop in the device supply, and the switching time for the device.

Capacitors are also used as timing elements — the RC time constant for charging and discharging is used with a comparator to cause a circuit element to switch state at preset high and low voltage points.

Capacitors are also used as sensors.
Typically, either the physical material between the capacitor plates is modified in some way (e.g., compressed), changing its physical properties, or the separation or configuration of the plates is modified.

## More lab preparation — compiling C programs

From the next lab, you will write code in the C language.
Although the statement syntax is similar, the C language has a number of conventions and “traditions” that are not found in Java.
C compilers like `gcc` typically have a large number of options, and `gcc` in particular can target many different underlying processor architectures. The command lines for a simple compile operation can consequently become rather complex.
For example, to compile a C program (say `test.c`) for the ATmega1284p, one might use

```
avr-gcc -mmcu=atmega1284p -Os -o test.elf test.c
```

This would generate a file `test.elf`, which would then have to be converted to a binary (`.hex`) file using

```
avr-objcopy -O ihex test.elf test.hex
```

and then programmed into the ATmega1284 chip using

```
avrdude -p m1284p -c stk500 -P /dev/ttyS0 -e \
-U flash:w:test.hex
```

It is common for C programmers to use the program `make` to organize the operations required to convert a source program to an executable.
Typically, a file called `Makefile` is created, which contains rules that the make program follows to compile a program.
Following is the contents of a simple `Makefile` for a single C program targeting the ATmega1284p architecture:

```
#
# a simple makefile for a single program test.c
#
CC = avr-gcc -mmcu=atmega1284p -Os
OBJCOPY = avr-objcopy
BURN = avrdude -pm1284p -c stk500 -P/dev/ttyS0 -e

PROG=test

$(PROG).hex: $(PROG).elf
        $(OBJCOPY) -O ihex $< $@

$(PROG).elf: $(PROG).c
        $(CC) $< -o $@

clean:
        rm -f *.o $(PROG).elf $(PROG).hex

burn: $(PROG).hex
        $(BURN)  -U flash:w:$(PROG).hex
```

Actually, this file also contains instructions to write the executable code into the processor, using `avrdude`.
To compile the code, type `make`, to burn the code in the flash memory of the processor, type `make burn`.

## Counter/timers in the AVR

A *counter* is an n-bit register that increments whenever its clock input cycles. (It may either be on a rising or falling edge, for example.) Counters that are incremented by a regular, stable clock signal of known frequency can be used as *timers*.

All AVR processors have at least one 8-bit counter/timer; most have at least one other counter, either 8 or 16 bit. The most common configuration is two 8-bit and one 16 bit counter/timers.
This is what is found in the ATMega1284p.

Most timer/counters in the AVR are constructed with:

- a counter register, connected to an adder or incrementer,
- reset circuitry to set the counter to 0,
- a multiplexor that selects the clock source for the counter,
- a prescaler that divides the clock frequency by a fixed amount,
- an overflow signal that indicates when the counter exceeds its maximum value and returns to 0,
- a comparator (output compare unit) connected to the counter register, and a register holding a comparison value,
- control logic to direct the counter's operations.

Some counters can count both up and down.

The comparator can be used to reset the counter when the value in the counter register matches the value of the comparison register.

The overflow signal and the output of the comparator typically can be used to interrupt the microcontroller.

The output of the comparator is often connected to an external output pin. This output can be used as frequency or clock signal source.

Like the other on-chip peripherals, timers are controlled mainly through dedicated I/O registers. They require rather more complex configuration than the simple I/O ports, however.

Moreover, the different timers have different capabilities, so each timer is controlled differently.

Also, timers on different processors also have different capabilities, so code written for one processor may not work on another, or the code may require changes to work correctly.

The timers can generate a number of different waveforms. The waveform is invariably some period of time when the output is low, followed by some period of time when the output is high, which may then be repeated. (The opposite is also possible, as well.)

The major differences are how the different low and high periods are determined, and how the waveform is repeated.

Some waveforms are specifically designed for controlling a particular type of device.

Usually, the waveforms relate to the *mode* in which the timer is operating. The mode is normally chosen by setting some particular bits in a control register.

## General properties of AVR counter/timers

Most of the AVR timers have a number of things in common, and the documentation uses similar notation when describing them.
The term `BOTTOM`, for example, refers to the lowest value in the counter register (`0`).
`MAX` refers to the maximum value the counter can reach (`0xFF`) for an 8-bit counter.
`TOP` refers to the highest value in the count sequence. In some cases, this may be `MAX`, but it may also be a value in a counter register.

The counter registers are generally labeled similarly in the AVR processors; typically the count register itself is labeled `TCNTn` where `n` refers to the particular counter (counter 0, 1, or 2 in the ATMega1284p).

The output compare register(s) `OCRnx` are used to determine the value of `TOP` in some modes, and may cause the counter to reset or stop counting when the value in the register is reached.

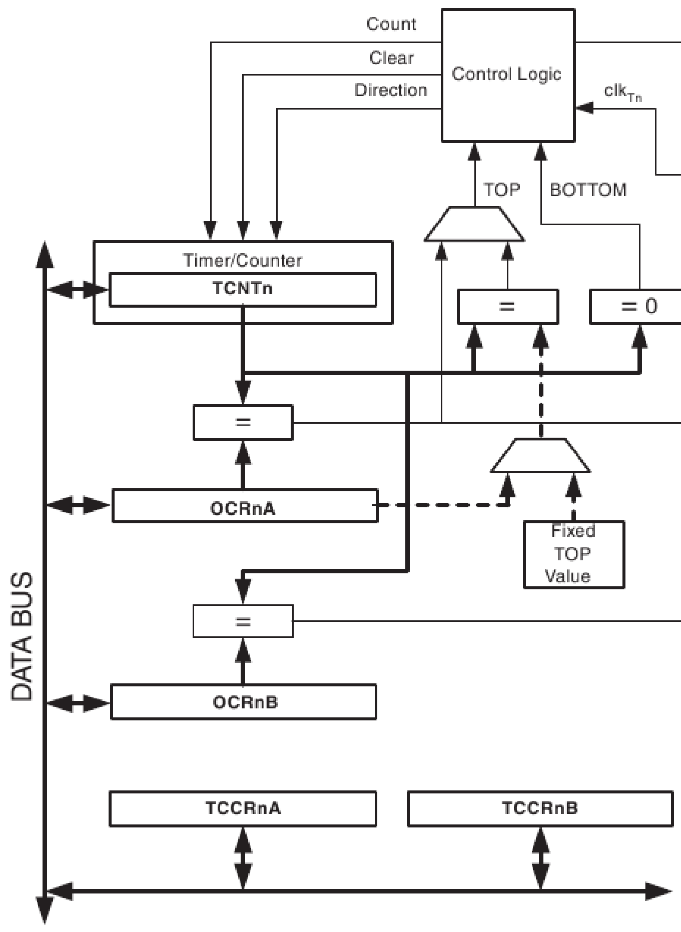
The timer control register(s) `TCCRnx` set the mode for the counter, the value for the prescaler, and other control functions.

The timer interrupt mask register `TIMSKn` determines which of the timer interrupts are enabled.

The timer interrupt flag register `TIFRn` indicates whether a condition which could have caused an interrupt has occurred. If the particular interrupt was enabled, then an interrupt would occur; otherwise, the condition could be checked in software by reading this flag register.

Following is a diagram of a typical AVR timer:

8-bit Timer/Counter Block Diagram



Note that there are *two* control registers, and two output compare registers.

We will next look at some typical operating modes for the timers.

**Normal mode:**

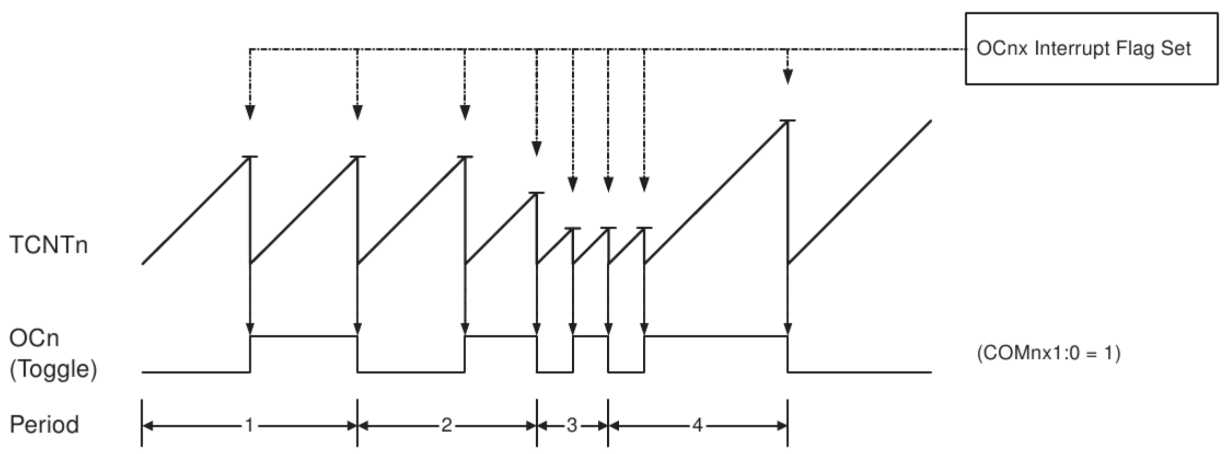
The simplest mode is normal mode. Here, the counter direction is always up (incrementing), and no counter clear is performed. The counter simply overruns when it passes its maximum 8-bit value (`TOP = 0xFF`) and then restarts from the bottom (`0`).
The Output Compare Unit can be used to generate interrupts at some given count value during the cycle.

**Clear timer on Compare Match (CTC) mode:**

In CTC mode, the `OCRnA` register defines the `TOP` value for the counter. The counter is cleared to zero when the counter value (`TCNTn`) matches the value in `OCRnA`.
The CTC mode allows greater control of the Compare Match output frequency. It also simplifies the operation of counting external events. The counter value (`TCNTn`) increases until a Compare Match occurs between `TCNTn` and `OCRnA`, and then the counter (`TCNTn`) is cleared. An interrupt can be generated each time the counter value reaches the `TOP` value by using the `OCFnA` flag.
To generate a waveform output in CTC mode, the `OCnA` output can be set to toggle its logic level on each Compare Match by setting the Compare Output mode bits to toggle mode.

Following is a timing diagram for the CTC mode:

CTC Mode, Timing Diagram



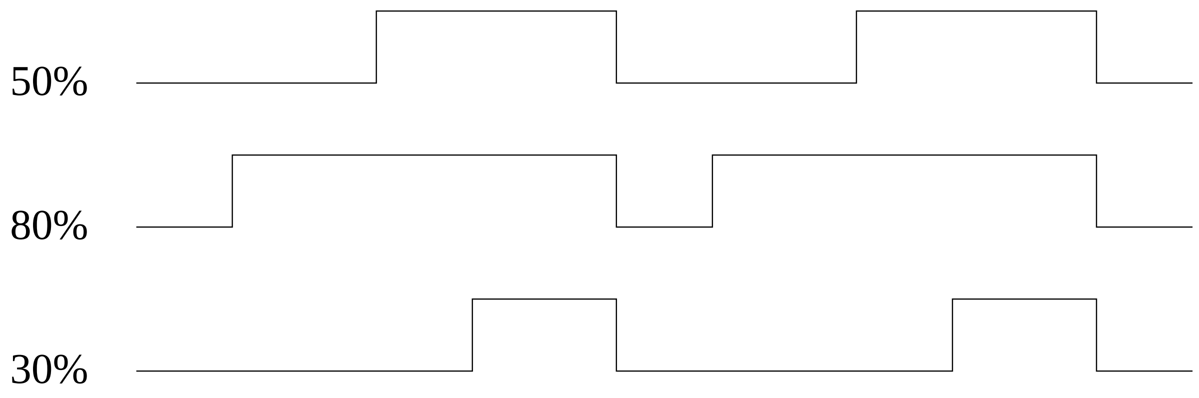
It is possible to generate a symmetric waveform with a maximum frequency of 1/2 the clock frequency.
In general, a waveform with frequency

$$f = \frac{f_{clock}}{2 \times N \times (1 + \texttt{OCRnA})}$$

Here $N$ is the factor from the prescaler.

## Pulse width modulation (PWM):

Pulse width modulation essentially is a waveform where the width of the (usually periodic) pulse is varied, or modulated, by some controller.
One use of PWM is to control a device. For example, it can be used to control the intensity of light output from a LED.



Consider the three waveforms above. The output is high (on) for different parts of the time in each case. It the output was connected to a LED, the brightness would be different in each case.
If it were a motor instead of a LED, the speed of the motor could be controlled similarly.
This is a common way to use digital output to control a device.
The percentage of time the output is *on* is called the *duty cycle* of the pulse.

Typically, in a PWM mode, the counter/timer will be set to output a value (say, low, 0) for the count values below that contained in some register (e.g., `OCRnx`), and then the other value (say, high, 1) until the counter reaches `TOP`.

The amount, or degree, of control afforded by a PWM controller is the number of different ratios of on/off that are possible.

For an 8-bit counter, this would correspond to a resolution of 1 part in 256. This is often expressed as "8 bit resolution.")

In some systems, it is possible to set both the trigger value (transition point from 0 to 1, say, and the total number of counts in a cycle.
In this case, the resolution would be smaller. E.g., if the total number of counts was 128, then the resolution would be 1 part in 128. (This might be stated as "7 bit resolution.")
If the total number of counts was 100, the resolution would be 1 part in 100, which might be stated as "6.6 bit resolution.")

For many devices, having a good linear resolution in the input may not necessarily produce the desired resolution in the output.
For example, in the case of apparent brightness from a LED, the human eye responds logarithmically, not linearly, so a 10% change in light energy is not the same as a 10% change in apparent brightness.
For small changes, however, many systems do behave in a linear fashion, and simple digital PWM is an effective way to control their behavior.

We will next look at specific AVR timers.

## The ATMega1284p timers:

The ATMega1284p has three timers, two 8-bit (`TIMER0` and `TIMER2`), and one 16-bit (`TIMER1`). We will describe the operation of the 8-bit counters first.
The following descriptions are incomplete; refer to the data sheets for detailed information.

## Timer/Counter0, an 8-bit Timer/Counter

The 8-bit Timer/Counter0 supports the following features:

- Two Independent Output Compare Units
- Double Buffered Output Compare Registers (during PWM)
- Clear Timer on Compare Match (Auto Reload)
- Glitch Free, Phase Correct Pulse Width Modulator (PWM)
- Variable PWM Period
- Three Independent Interrupt Sources (TOV0, OCF0A, and OCF0B)

The counter has a prescaler which can divide the input clock frequency by factors of 1, 8, 64, 256, or 1024.
The prescaler is quite useful for generating longer time periods, but the accuracy is still limited to 8 bits.

The counter can generate outputs on two output port pins, `OC0A` and `OC0B` (pins `PB3` and `PB4`, respectively.) It can accept input from one input port pin, `T0` (pin `PB0`).
These pins are shared with other functions, and must be set appropriately. (Also, obviously, only one functional unit can control an input or output at any given time.)
Typically, to set `OC0A` to output, one would write:

```
  DDRB |= _BV(DDB3)
```

which would only set the single bit (bit 3) in the `DDRB` register to 1.

The input is synchronized with the internal oscillator of the processor by an internal synchronizer. The input therefore must be stable for longer than a single clock cycle.

It is fairly common for the counter to provide no direct external output, but to interrupt the processor periodically in order to check the status of other inputs. (A data logging application, for example.)

The counter can generate three interrupts (see the interrupt vector table) independently, namely `TIMER0_COMPA`, `TIMER0_COMPB`, and `TIMER0_OVF`.

**Timer/Counter0 interrupt and flag bits:**

The three interrupts, `TIMER0_COMPA`, `TIMER0_COMPB`, and `TIMER0_OVF`, can be individually enabled and disabled by setting the appropriate bits in the mask register, `TIMSK0`.
The bits which enable these interrupts are labeled `OCIE0A`, `OCIE0B`, and `TOIE0` respectively.
When `OCIE0A` is set, an interrupt with vector `TIMER0_COMPA` is generated when `TCNT0` equals the value in `OCR0A`. Similarly for a match with `OCIE0B`, or an overflow.

TIMSK0:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| – | – | – | – | – | OCIE0B | OCIE0A | TOIE0 |
| R | R | R | R | R | R/W | R/W | R/W |

TIFR0:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| – | – | – | – | – | OCF0B | OCF0A | TOV0 |
| R | R | R | R | R | R/W | R/W | R/W |

The bits which are set in the flag register, `TIFR0`, are labeled `OCR0B`, `OCR0A`, and `TOV0` respectively.
These bits are set under the appropriate conditions (match or overflow) even if the corresponding interrupt is not enabled, and can be polled under program control.

## Controlling Timer/Counter0:

The basic operation of the counter is controlled with the two registers `TCCR0A` and `TCCR0B`.
There are four groups of bits in these registers which determine:

`COM` **bits** (compare output match).
These bits determine what happens on the output pins depending on the compare match.

`CS` **bits** (clock select).
These bits determine the clock source and the prescaler value.

`WGM` **bits** (waveform generation mode).
These bits determine the type of waveform generated.

`FOC` **bits** (force output compare).
These bits force an output compare match.

TCCR0A:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| COM0A1 | COM0A0 | COM0B1 | COM0B0 | – | – | WGM01 | WGM00 |
| R/W | R/W | R/W | R/W | R | R | R/W | R/W |

TCCR0B:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| FOC0A | FOC0B | – | – | WGM02 | CS02 | CS01 | CS00 |
| R/W | R/W | R | R | R/W | R/W | R/W | R/W |

The detailed function of these bits is tabulated in the documentation. We will look at some of their functionality.

The `CS` (clock select) bits:

| `CS02` | `CS01` | `CS00` | Description |
|---|---|---|---|
| 0 | 0 | 0 | Timer/counter stopped |
| 0 | 0 | 1 | I/O clock |
| 0 | 1 | 0 | (I/O clock)/8 (from prescaler) |
| 0 | 1 | 1 | (I/O clock)/64 (from prescaler) |
| 1 | 0 | 0 | (I/O clock)/256 (from prescaler) |
| 1 | 0 | 1 | (I/O clock)/1024 (from prescaler) |
| 1 | 1 | 0 | External input `T0`, rising edge |
| 1 | 1 | 1 | External input `T0`, falling edge |

Note that setting the `CS` bits to `0` stops the counter.
Only the system clock can be used as an input to the prescaler.

The prescaler is typically used when relatively long timing periods are required. For example, suppose we have the (default) internal clock frequency of 1 MHz., corresponding to a cycle time of 1 $\mu$s. If we want to interrupt the processor every 0.1 seconds (i.e., after 100,000 processor cycles) then we would need to divide the clock by 1024.

(Even then, we could not get exactly 0.1 s — $100,000/1024 = 97.656$. We could use 97 or 98 as the value for `TOP`, and be either too slow or too fast by 672 or 352 processor cycles, respectively.)

The `WGM` (waveform generation mode) bits:

| `WGM2` | `WGM1` | `WGM0` | Mode | `TOP` | Update of `OCRx` | `TOV` flag set on |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | Normal | `0xFF` | immediate | `MAX` |
| 0 | 0 | 1 | PWM, phase correct | `0xFF` | TOP | `BOTTOM` |
| 0 | 1 | 0 | CTC | `OCRA` | Immediate | `MAX` |
| 0 | 1 | 1 | PWM, fast | `0xFF` | TOP | `MAX` |
| 1 | 0 | 0 | RESERVED | `-` | `-` | `-` |
| 1 | 0 | 1 | PWM, phase correct | `OCRA` | TOP | `BOTTOM` |
| 1 | 1 | 0 | RESERVED | `-` | `-` | `-` |
| 1 | 1 | 1 | PWM, fast | `OCRA` | TOP | `TOP` |

The four operating modes, *normal*, *CTC*, *fast PWM*, and *phase correct PWM*, are described in detail in the documentation.
We will use the CTC (clear timer on compare match) mode in an examples following, and in the lab, so we will describe it in more detail. (We will use the PWM modes in a future lab.)

For Counter/Timer0, in CTC mode the value in the `OCR0A` register defines the value at which the counter is reset (`MAX`). Thus, `OCR0A` defines the maximum value reached by the counter, and also its resolution.

The `COM` (compare output match) bits depend on the particular mode for which the counter is programmed.

The COM (compare output match) bits in CTC mode:
These bits determine the function of the output pins associated with the counter. The following table applies to both normal and CTC mode, but not the PWM modes.

| COM0A1 | COM0A0 | Description |
|---|---|---|
| 0 | 0 | Normal operation, OC0A disconnected |
| 0 | 1 | Toggle OC0A on compare match with OCR0A |
| 1 | 0 | Clear OC0A on compare match with OCR0A |
| 1 | 1 | Set OC0A on compare match with OCR0A |

A similar table applies for the output OC0B, with B substituted for A.

Note that for the output to be visible at the pin, the DDR register for that pin must be set to output (1).

CTC mode is useful for generating a symmetric square wave output with a given period. This can be done by setting the value in OCR0A to ( 1 - half the required period), and choosing Toggle mode in the table above.
This would result in the output changing state every time the counter reached the count in OCR0A, which requires 1 + OCR0A steps.

This is, in fact, the technique used to generate the "music" in the AVR Butterfly (and the example shown in the first class.)

## Example to generate a single frequency using CTC mode

This example generates a square wave of frequency 2093 Hz (the note C7), assuming the processor clock frequency is 1 MHz.

The code is almost identical to the previous example of a buzzer which generated a tone when a button was pressed. The main difference is the addition of code that initializes timer/counter0, and the function `buzz()`, which now merely turns the timer on, waits until the button is no longer pressed (in a busy loop), and then turns the timer off.

The code is also reorganized, to look more like a typical C program. Following is the `beep-timer.h` file, which include the AVR include files, and the function prototypes:

```
#include <avr/io.h>
#include <util/delay.h>
#include<avr/interrupt.h>
#include<avr/sleep.h>

void initialize(void);
void int_handle(void);
void buzz(void);
```

Following is the code for the main program and functions:

```
#include "beep-timer.h"

// input (button switch) is on pin PB4
// output (speaker) is on pin PB3 (timer 0 output OC0A)

int main(void)
{
        initialize();  // set up timer and I/O ports

        // infinite loop, sleeping until button is pressed
        while (1) {
                sleep_mode();
                buzz();
        }
        return(1);
}
```

```
void initialize(void)
{
   // set up port B

   DDRB  = _BV(DDB3);       // set port B pin 3 as output
   PORTB = ~(_BV(DDB4));     // set pullups on inputs

   // set up PCI interrupts

   PCMSK1 |= _BV(PCINT12);   // enable PCI interrupt on PB4
   PCICR  |= _BV(PCIE1);     // enable PCI interrupt 1
   PCIFR  |= _BV(PCIF1);     // clear PCI interrupt flag 1

   // set up timer 0 to CTC mode

   TCCR0A = _BV(WGM01)|_BV(COM0A0);   // timer0 in CTC mode
                            // toggle OC0A on compare match
   TIFR1 = _BV(OCF1A);      // clear timer flag
   OCR0A = 238;             // set output frequency

   set_sleep_mode(SLEEP_MODE_PWR_DOWN);   // set sleep mode

   sei();                    // enable interrupts
}
```

```
EMPTY_INTERRUPT (PCINT1_vect)  // empty handler for PCI
                               // its function is to wake
                               // up the processor

/*
  Older code would have written this as

   ISR(PCINT1_vect)
   // empty handler for PCI; the only function is to
   // wake up the processor
   {
   }
*/

void buzz()
{
   TCCR0B = _BV(CS00);           // start timer, no prescale
   while(!(PINB & (_BV(PB4)))); // button still pressed?
   TCCR0B = 0;                   // turn off timer
   return;
}
```

## How did we calculate the value for `OCR0A`?

The desired frequency was 2093 Hz, which corresponds to a time period of 1/2093 seconds = 0.0004778 seconds, or 477.8 $\mu$s.

A half period is 238.89 $\mu$s.

`OCR0A` +1 corresponds to the half period in CTC mode, so the value should be 237.89, so setting `OCR0A` to 238 would give a good approximation of the desired frequency.

The actual frequency, assuming a 1MHZ clock, is $1,000,000/(2\times239)$ HZ or 2092.05 Hz.

In general, the frequency generated is

$$f = \frac{f_{clock}}{2 \times N \times (1 + \texttt{OCRnA})}$$

were $N$ is the factor from the prescaler (1 in this case).

## Timer/Counter2, another 8-bit Timer/counter

Timer/Counter2 is very much like Timer/Counter0. It has much the same structure, and is controlled similarly.
The main difference is that Timer/Counter2 can operate with an asynchronous input. (Timer/Counter0 could accept an external input, but it was synchronized with the internal processor's clock by a synchronizer (see Figure 17.1 in the ATMega1284p documentation.)
Asynchronous input to the timer/counter is input on pin `TOSC1` (PC6).
Another function for which this counter was designed is real-time clock (RTC), typically operating with a crystal of frequency 37.768 KHz. In fact, this timer has an oscillator driver specifically designed for this function. Pins `TOSC1` and `TOSC2` (PC6 and PC7) can be connected to this oscillator, internally.

One potential problem with an asynchronous counter is that the counter value may change at exactly the same time it is being read (or written) giving an inaccurate result.
The AVR can deal with this, however.
Timer/Counter2 has an additional register, the *Asynchronous Status Register* (`ASSR`).
This register determines the input source type (synchronous or asynchronous) and contains flags to indicate when the value in a register is stable in asynchronous mode.

## The *Asynchronous Status Register* (ASSR)

**ASSR:**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| – | EXCLK | AS2 | TCN2UB | OCR2AUB | OCR2BUB | TCR2AUB | TCR2BUB |
| R | R/W | R/W | R | R | R | R | R |

**Bit 6** When EXCLK is written to one, and asynchronous clock is selected, the external clock input buffer is enabled, and an external clock can be input on pin `TOSC1`.

The crystal oscillator will not run when this bit is set.

**Bit 5** When `AS2` is written to zero, Timer/Counter2 is clocked from the internal clock. When `AS2` is written to one, it is clocked from a crystal oscillator connected between pins `TOSC1` and `TOSC2`.

**Bit 4** When Timer/Counter2 operates asynchronously and `TCNT2` is written, this bit (`TCN2UB`) becomes set. When `TCNT2` has been updated from the temporary storage register, this bit is cleared by hardware. A logical zero in this bit indicates that `TCNT2` is ready to be updated with a new value.

**Bits 3** `OCR2AUB` is similar to Bit 4, but for register `OCR2A`.

**Bit 2** `OCR2BUB` is similar to the previous, but for register `OCR2B`.

**Bit 1** `TCR2AUB` is similar to Bit 4, but for register `TCCR2A`.

**Bit 0** `TCR2BUB` is similar to the previous, but for register `TCCR2B`.

See the data sheet for more information, especially if you attempt to use asynchronous mode.
The documentation outlines an algorithm for safely reading and updating asynchronous registers.

## Example of a real-time clock using timer/counter2 and a 32.768 KHz crystal

This example uses an external 32.768 KHz. crystal connected to the `TOSC` inputs of timer/counter2 to generate an interrupt every 1/128 seconds. The timer operates in `normal` mode, and generates an interrupt after every 256 cycles. (32,768/256 = 128 interrupts/second.) The interrupt handling updates a data structure to keep track of minutes and seconds.
Following is the file `rtc.h`:

```
#include <avr/io.h>
#include <util/delay.h>
#include<avr/interrupt.h>
#include<avr/sleep.h>

typedef struct{
unsigned char tick; // a 1/128 second tick
unsigned char second;
unsigned char minute;
            }time;
volatile time t;

void init_rtc(void);
void init_io(void);
void display(void);
```

Since this is the first interrupt handling routine to actually modify a value in the program, it is useful to comment about this.
Note that the data structure here consists of three `unsigned char` entities. This is because variables of type `char` are defined to be 8 bits, and the values stored in them will be less than 8 bits.
They are not characters, however, so it would be better to have an 8 bit integer type. The AVR gcc library defines some 8 bit integer types. The most common is `uint8_t`, which we could use here as well.
They are defined, and described, in the AVR libc documentation, under `<stdint.h>`, and include signed and unsigned fixed length integers of 8, 16, 32, and 64 bits.

The other, more important, thing to notice is that the structure is given the type `volatile`.
The type qualifier `volatile` tells the compiler that the entity it defines may be changed by something not contained directly in the code. An optimizing compiler may well notice that the variable is not changed by the part of code in which it is used, and "optimize" it out.
The `volatile` qualifier tells the compiler that it should not perform certain optimizations with that variable.

All variables modified within an interrupt handling routine should be qualified as `volatile`.
Interrupt routines should be as **short** as possible.

The program code (`rtc.c`) follows:

```
/*********************************************************/
// RTC  - gives minutes, and seconds with full resolution
// interrupts every 1/128 second
/*********************************************************/

#include "rtc.h"

int main(void)
{
   init_io();
   init_rtc();

   set_sleep_mode(SLEEP_MODE_IDLE);
   while(1)
   {
      sleep_mode();  //will wake up from timer
                     // overflow interrupt
      display();
   }
}
```

```
void init_io()
{
   DDRD = 0XFF;      // port D is output
   PORTD = 0X00;
}

void init_rtc()
{
   cli();                 // disable global interrupts
   ASSR |= _BV(AS2);      // set Timer/Counter2 to be
                          // asynchronous, driving a crystal
   TCCR2A = 0;            // normal mode
   TIMSK2 = _BV(TOIE2);  //set 8-bit Timer/Counter0 Overflo
                          // Interrupt Enable
   _delay_ms(1000);       // let oscillator stabilize
   TCNT2 = 0x00;          // set counter to 0
   TCCR2B = _BV(CS20);    // start counter, no prescaling
   TIFR2=0;               //clear any pending timer interrup

   sei();                 //enable global interrupts
}
```

```
void display()
{
   PORTD = ~t.second;
// PORTD = ~t.minute;
   _delay_us(10);   // not really necessary
}


ISR(TIMER2_OVF_vect)
{
   if (++t.tick==128)
   {
      t.tick=0;
      if (++t.second==60)
      {
         t.second=0;
         if (++t.minute==60)
         {
            t.minute=0;
         }
      }
   }
}
```

## Timer/Counter1, a 16-bit Timer/counter

Timer/Counter1 is also much like Timer/Counter0. It has much the same structure, and is controlled similarly.
The major differences are that it is a true 16 bit counter/timer, so two reads/writes are required to obtain or update timer values, and there must be some mechanism to ensure that the correct value is read even if the counter updates between the read operations.

Each 16-bit timer has a single 8-bit register (`TEMP`) for temporarily storing the high byte of the 16- bit access, shared among all 16-bit registers in the timer.
When the low byte of a 16-bit register is read by the CPU, the high byte of that register is copied into the temporary register in the same clock cycle as the low byte is read.
When the low byte of a 16-bit register is written by the CPU, both the high byte stored in the temporary register and the low byte written are copied into the 16-bit register in the same clock cycle.
Therefore, for a 16-bit read, the low byte must be read before the high byte.
For a 16-bit write, the high byte must be written before the low byte.
In C, the compiler takes care of this. However, if an interrupt occurs between the two reads or writes, the value could be corrupted. If this is a possibility, interrupts should be disabled for the read.

Typical code to prevent a corrupted read would be:

```
    cli();
    i = TCNT1;
    sei();
```

## The Input Capture Unit (`ICU`)

Timer/counter1 has a mode in which it can apply a time stamp to an external event. Essentially, if an external input (pin `ICP`, shared with `PD6`) changes state, the value of the timer at this event can be saved.

The value in the counter is written to the Input Capture Register (`ICR1`) and can be read low byte first, with the high byte simultaneously being stored in the `TEMP` register.

The input to the `ICU` can be either the Input Capture Pin (`ICP`), or one of the analog comparator outputs. (We will discuss the analog comparators later.)

The input triggers a time stamp on either the rising or falling edge of the input (programmable with bit `ICES1` of control register `TCCR1B`).

The input can be *filtered* by requiring that it be stable for 4 internal clock periods (again programmable, with bit `ICNC1` of `TCCR1B`).

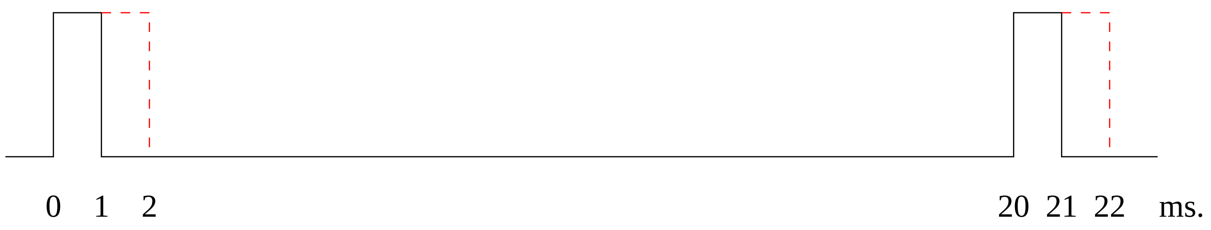
A capture event can also be used to trigger an interrupt, with vector `TIMER1_CAPT`.

Timer/counter1 has other interesting features; read the data sheet.

## A PCM example using Timer/counter 1 — controlling a servo motor

A servo motor is an interesting, and very useful device. It is a device with an output shaft that can be positioned at some specific angular position by sending a signal to the device.
The signal can be altered, and the position of the shaft varied.
They are used extensively in robotics, as well as in radio controlled planes, etc. for controlling ailerons, rudders, etc.
The typical "hobby" servo is quite powerful for its size. The one we will use in the lab is at the low end of the scale, and it has a torque of about 21 oz.-in. (it can lift 21 oz. at a point 1 inch from the shaft's axis.) It can typically rotate more than 180 degrees, but this varies by manufacturer.
A servo has a three wire connection. They are power (red), ground (black or brown) and control (white or orange).
For the small hobby servos, the control is a pulse of from 1 to 2 ms. in duration, repeated about every 20 ms. (50 pulses per second.) The duration of the pulse determines the position of the servo; 1.5 ms. is the neutral position (say, 90 degrees). 1 ms. would then correspond to 0 degrees, and 2 ms. would correspond to 180 degrees.
Essentially, the control signal is a PCM signal with a maximum duty cycle of 10%. (2 ms. in 20 ms.)
We will design a controller for such a servo.

We need to generate the following waveform:



There are a number of possibilities. For example, we could use a timer to interrupt the processor periodically (say, every 0.01 ms.) and then set a port bit appropriately, keeping track of the time using program variables. This would give an angular resolution of 1 part in 100, or about 1.8 degrees. (Effective control is for pulse widths between 1 ms. and 2 ms., for a duration of 1 ms.)
While this method may be effective if we were controlling several servos, we would prefer a "set and forget" approach, using a timer in PCM mode.
For a single device, this is simpler, and the processor could be doing other things. (If we had more devices than timers, however, the other approach might work well.)

If we use an 8-bit counter, the largest number of steps is 256. The control range (1 ms. to 2 ms.) is 5 % of this, or about 12 steps. This gives a minimum angular step of $180/12 = 15$ degrees (probably inadequate.)
Assuming a processor clock frequency of 1,000,000 Hz, using Counter 2 and dividing the clock by 128 with the prescaler, this would give a period of 32.768 ms. (which would probably be adequate.)

If we want finer control, or a period closer to 20 ms., we will have to use a counter with more bits (Counter 1).
Counter 1 can be used for PWM as an 8, 9, or 10 bit counter. With a 10 bit counter, we could have about 50 discrete steps, which might be adequate for some applications.

With a 16 bit counter, the resolution can be as high as 1 part in $2^{16} = 65536$.
Potentially, for our application, the resolution would be 5% of that, (1 part in 3276) which is still rather more than necessary.
For a 1 MHz. clock, 20 ms. corresponds to 20,000 clock periods, which would also be more than sufficient resolution (1000 steps).

There are several PCM modes in which the `TOP` value for the counter is set by a register, either register `OCR1A` or register `ICR1` (if input capture is not being used.) We will use `ICR1` to define `TOP`, arbitrarily.

```
    ICR1 = 20000;    // set period
```

There are also three different PWM modes; fast PWM, Phase Correct PWM, and Phase and Frequency Correct PWM. We could use any of those, so will use the simplest (fast PWM.)

For input to the counter, we will use the 1 MHz. internal clock. Looking at Table 16.6, bit `CS10` must be set in the control word.

Looking at the possible modes in Table 16.5, mode 14 is fast PWM with `TOP` defined by `ICR1`, so we will use this mode. (`WGM13`, `WGM12`, and `WGM11`, set).
To generate the pulse, we can use one of the output compare registers, with output initially set to 1, and then cleared when the register count is reached. We will choose (arbitrarily) register `OCR1A` as the compare register, and set the output, pin `OC1A`, (see Table 16.3) to be "set at `BOTTOM`, clear on MATCH." (I.e., a 1 until the count is reached, then 0.)
This requires bit `COM1A1` to be set in the control register.

```
    TCCR1A = _BV(COM1A1)|_BV(WGM11);
    TCCR1B = _BV(WGM13)|_BV(WGM12)|_BV(CS10);
```

The value in register `OCR1A` can range from 1000 to 2000.

```
    OCR1A = 1000;    // set at minimum
```

This is the full timer configuration to generate the required PWM output.
The servo is controlled by setting the value in `OCR1A` to any value between 1000 and 2000.

To make the code more interesting, we will "step through" all 1000 of the possible outputs to the servo. After each step, the value in `OCR1A` is incremented. When it reaches 2000, we reset it to 1000.

We can use the compare match interrupt for this. Since `OCR1A` is matched when this interrupt is generated, rewriting it with a larger number immediately could cause a spurious interrupt when the value was reached again.
Counter/timer1 has a provision to buffer updates to the compare registers, and looking at Table 16.5, we see that `OCR1A` is updated when the counter is reset (at `BOTTOM`.)
The minimum time between the interrupt and the end of the pulse is about 18 ms., so it is certain that the register will be updated for every pulse.
To enable the compare match interrupt, we set the timer interrupt mask, and clear the corresponding interrupt flag before enabling interrupts. (The last step is not necessary, but prudent.)

```
TIMSK1 = _BV(OCIE1A); // allow compare match A interrupt
TIFR1  = _BV(OCF1A);  // clear interrupts
```

In this example, when the processor is not updating the counter, it is placed in sleep mode to conserve power.

**Following is the** `include` **file,** `servo.h`**:**

```
#include <avr/io.h>
#include <stdint.h>
#include <avr/sleep.h>
#include <avr/interrupt.h>

void timer1_init( void );
```

**The main program,** `servo.c`**:**

```
#include "servo.h"

volatile uint16_t position = 1000;

int
main( void )
{
    set_sleep_mode(SLEEP_MODE_IDLE);
    timer1_init();

    while ( 1 ) {
        sleep_mode();
    }
    return 0;
}
```

**Following is the timer initialization, described earlier:**

```
void
timer1_init( void )
{
    /* output is clear on match, and set at bottom,
     * mode is 14, top is ICR1, prescale = 1
     */
    TCCR1A = _BV(COM1A1)|_BV(WGM11);
    TCCR1B = _BV(WGM13)|_BV(WGM12)|_BV(CS10);

    /* assuming a 1 Mhz system clock, a 20 ms period
     * is  0.020 / 0.000001 = 20000
     */
    ICR1 = 20000;         // set period
    OCR1A = 1000;         // set at leftmost position

    TIMSK1 = _BV(OCIE1A); // allow compare match A interrup
    TIFR1  = _BV(OCF1A);  // clear interrupts

    DDRD  |= _BV(PD5);    // enable timer1 PWM output
    sei();
}
```

The only additional code here is to set the DDR for the timer output bit to allow output, and to enable global interrupts.

The interrupt handler:

```
ISR(TIMER1_COMPA_vect)
{
    if (position < 2000) position++;
    else (position = 1000);
    OCR1A = position;
}
```

Actually, there is no need to set the position in the interrupt handler. It could be done in the main program (which now does very little).

Which would be better? Why?

Could we control another servo motor independently using the same timer? (Remember that there is another compare register, `OCR1B`, and another output pin, `OC1B`.)

Although we could continue discussing timers and their uses for much longer, we will move on to other devices.

It is worthwhile to think about other applications for timers as we continue, and to read about other timer functionality in the datasheet.

We will return to timers briefly when we discuss the control of several other devices (e.g., motors.)

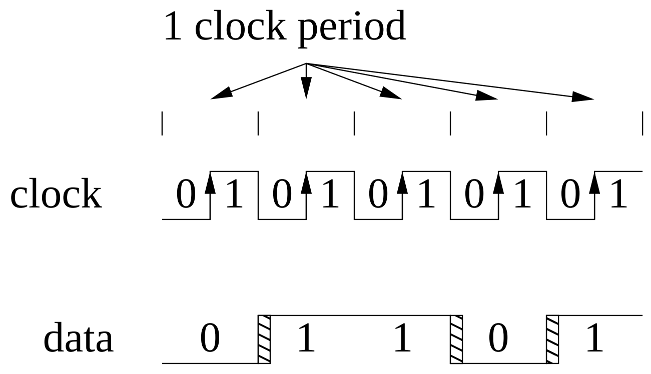
## Serial data communication

In order to make communication between two points cheaper, reducing or minimizing the number of wires connecting the points is essential.
This is obvious in the large scale, but is true on virtually every scale — in integrated circuits, interconnections require area that could be otherwise used by circuitry.
There are two general classes of serial communication, called *synchronous* and *asynchronous* communication. The distinction is based on whether or not a *clock* signal is transmitted with the data.
Asynchronous communication uses fewer wires (no clock signal) but may not allow large data packets. It may be the only option for communication when there is only one "channel" — for example, most remote controls for household audiovisual equipment (the TV remote) use bursts of infrared light (IR) to communicate information.

Many things can be used to represent digital data; for example, a logic 0 or 1 can correspond to a low or high voltage, or vice versa.
Presence or absence of electric current, or even the direction of current flow can also be used to represent data, as well as magnetic polarization (N-S or S-N) — this is used on magnetic disks, which are also a kind of serial device.
The presence or absence of light pulses (or even pulses of a given frequency) can also be used (as in the TV remote.)

## Synchronous communication

The following diagram shows a typical waveform for a clock and data for synchronous data communication. The clock indicates when the data can be read; typically one clock edge (e.g., the rising edge, indicated by a $\uparrow$ in the diagram). Data is typically changed on the other clock edge, and is not considered stable at that point.

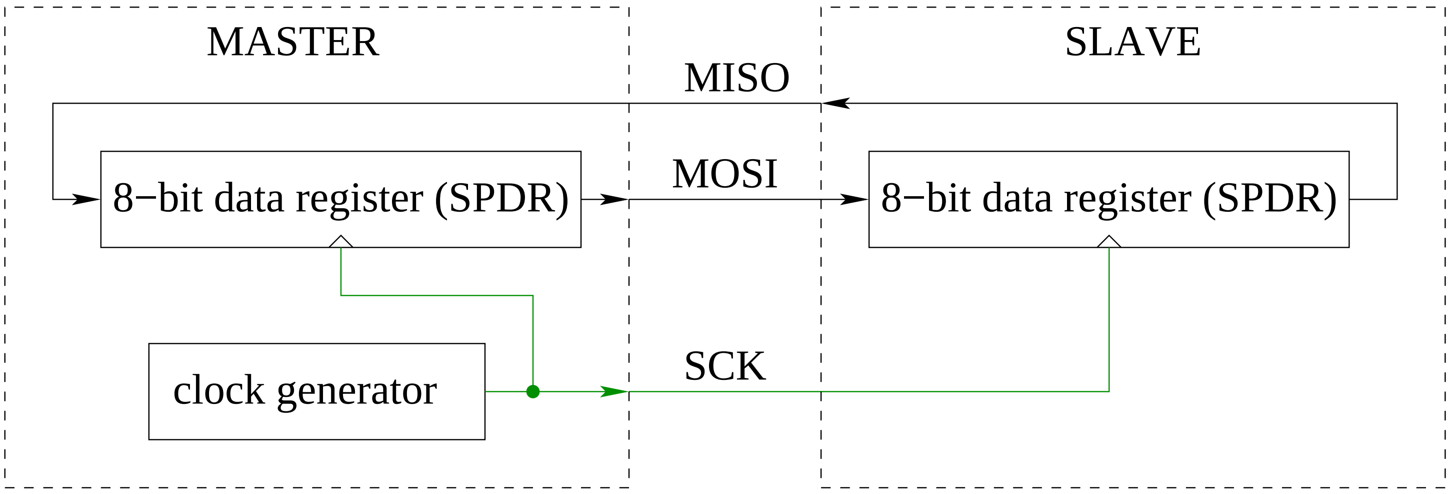


One bit of data is sent each clock period.
There is one clock, and one data transmitter, but there can be more than one receiver. The clock source and transmitter can change from time to time in some systems.
Typically, the device that controls the clock is called the *master* and the other nodes are called *slaves*.
In some systems, all slaves listen to all transmissions (an address may be encoded in the transmission). In other systems, a separate input is used to address the particular slave (or slaves) targeted by the transmission.
Many Atmel processors have hardware support for several serial communication methods. We will initially discuss two of them.

## The Serial Peripheral Interface (SPI):

The SPI function in the ATMega devices is a hardware synchronous interface to devices supporting the SPI protocol. It is a simple protocol, probably best thought of as a pair of shift registers connected together. Shifting is clocked by the master device, and data from the master is shifted to the slave through the single output pin labeled `MOSI` — Master Out Slave In. The clock is output on pin labeled `SCK`.

*At the same time as data is shifted from the master to the slave* the slave shifts data to the master on the pin labeled `MISO` (Master In Slave Out).



A fourth signal, labeled `SS` (Slave Select) is used to address a particular slave. This is an input in slave mode, on the pin labeled `SS`. In master mode, any output pin can be used to select a slave.

In the ATMega1284p, pin `SS` is `PB4`. `MOSI`, `MISO`, and `SCK` are pins `PB5`, `PB6`, and `PB7`.

There is one interrupt associated with the AVR SPI function, `SPI_STC` (SPI Serial Transmission Complete). It is set at the end of transmission (1 byte) provided that interrupts are enabled in the SPI control register, and global interrupts are enabled.

There is no real "standard" for SPI communication. It arose in industry as a simple, robust means of communication, easily implemented in both hardware and software.

There are a number of different possibilities:

Data can be transmitted either low order bit first, or high order bit first (often called *little endian* and *big endian*).

The data bits themselves can be read on either a rising (low to high, $\uparrow$) clock edge, or a falling (high-to-low, $\downarrow$) clock edge.

The data can also be read on the leading edge or trailing edge of the clock (the phase of the clock).

All these possibilities can be accommodated in the hardware of the AVR SPI function, and can be programmed independently using the SPI control registers.

The main constraints in the AVR hardware are that the data word is 8 bits only, and the clock frequency in slave mode must be a maximum of 1/4 of the processor clock frequency.
In practice, because of the variability of clock speeds for different AVRs (some are specified at only 10% accuracy if the internal clock is used) this limit is not pressed.

## The SPI registers

The SPI uses only three internal registers in the AVR. They are the data register, `SPDR`, used for data input and output.
As bits are shifted out one end of this register from the master, they are shifted in from the slave.
The SPI control register, `SPCR`, sets all of the SPI communication options.

SPCR:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| SPIE | SPE | DORD | MSTR | CPOL | CPHA | SPR1 | SPR0 |
| R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |

**Bit 7** **`SPIE`:** `SPI` Interrupt Enable

This bit enables the SPI interrupt.

**Bit 6** `SPE`: `SPI` Enable

When this bit is written to one, the SPI is enabled. This bit must be set to enable any SPI operations.

**Bit 5** `DORD`: Data Order

When this bit is written to one, the LSB of the data word is transmitted first, otherwise the MSB of the data word is transmitted first.

**Bit 4** `MSTR`: Master/Slave Select

This bit selects Master SPI mode when written to one, and Slave SPI mode when written logic zero.

**Bit 3** `CPOL`: Clock Polarity

When this bit is written to one, `SCK` is high when idle. When zero, `SCK` is low when idle. (See Figures 18-3 and 18-4 in the data sheet for timing diagrams).

**Bit 2** `CPHA`: Clock Phase

The settings of the Clock Phase bit (`CPHA`) determine if data is sampled on the leading (first) or trailing (last) edge of SCK. (Again, see Figures 18-3 and 18-4 in the data sheet for timing diagrams).

**Bits 1–0** `SPR1`, `SPR0`: SPI Clock Rate Select 1 and 0

These two bits control the `SCK` rate of the device configured as a Master. They determine the divisor for the processor clock to generate the `SPI` clock frequency. Table 18-5 in the datasheet shows the divisor for each combination of `SPR1` and `SPR0`.

They have no effect on the Slave.

The SPI status register, `SPSR`:

SPSR:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| SPIF | WCOL | – | – | – | – | – | SPI2X |
| R | R | R | R | R | R | R | R/W |

**Bit 7** `SPIF`: SPI Interrupt Flag

When a serial transfer is complete, the `SPIF` Flag is set. If `SPIE` in `SPCR` is set and global interrupts are enabled, an interrupt is generated.

If `SS` is set as an input and is driven low when the `SPI` is in Master mode, this will also set the `SPIF` Flag.

`SPIF` is cleared by hardware when executing the corresponding interrupt handling vector.

Alternatively, the `SPIF` bit is cleared by first reading the `SPI` Status Register with `SPIF` set, then accessing the `SPI` Data Register (`SPDR`). This also applies to the `WCOL` bit.

**Bit 6** `WCOL`: Write COLlision Flag

This bit is set if the SPI Data Register (`SPDR`) is written during a data transfer.

**Bit 0** `SPI2X`: Double SPI Speed Bit

When this bit is set to logic one the `SPI` speed (`SCK` frequency) is be doubled when the SPI is in Master mode (again see Table 18-5).

## SPI communication example

Following is a simple polled I/O program to transfer data from the switches on the SDK-500 (through PORT D) to the LEDs (through PORT C) using the SPI.
Note that the function that sends the data also returns the received data from the slave.

**First**, the `include` file `spi_poll.h`:

```
#include <avr/io.h>
#include <util/delay.h>

void Init_IO (void);
void Init_Master (void);
unsigned char Master_Send (unsigned char byte)
```

**Next**, the code to initialize the input and output ports:

```

void Init_IO (void)
{
    DDRD  = 0x00;    // set port D to input
    PORTD = 0xFF;    // set pullups on port D
    DDRC  = 0xFF;    // set port C as output
    PORTC = 0xFF;    // initial output - LEDs off
}
```

**Next**, the functions to manage the SPI.
First, the initialization function:

```
//**********************************************************
// Modified from code for appnote AVR151
//**********************************************************

// Initialization Routine Master Mode (polling)
void Init_Master (void)
{
    char IOReg;

    // set PB4(SS), PB5(MOSI), PB7(SCK) as output
    DDRB    = _BV(PB4)|_BV(PB5)|(_BV(PB7);

    // enable SPI in Master Mode with SCK = CK/4
    SPCR    = _BV(SPE)|_BV(MSTR);

    IOReg   = SPSR;    // clear SPIF bit in SPSR
    IOReg   = SPDR;
}
```

**Next**, the function to send a single character from the Master.
(It also returns the character from the slave.)

```
unsigned char Master_Send (unsigned char byte)
{
    SPDR  = byte;                     // send byte
    while (!(SPSR & (_BV(SPIF))));  // wait until byte
                                      // is sent
    return(SPDR);
}



int main (void)
{
    Init_IO ();
    Init_Master ();              // Initialization (polling)

    while (1){
        if (PIND != 0xFF){
            PORTC = Master_Send (PIND);
            _delay_us(250);
        }
        else PORTC = 0xFF;
    }
}
```

## Testing the SPI

How can we test the SPI interface?

Clearly, we can set up two SDK-100 boards, program one in SPI master mode (as in the previous code) and the other in slave mode. This would mean two boards, with their power supplies (the ground connections, at least) connected together. This may not be unsafe normally, but I usually try to isolate different power supplies from each other.
It is inconvenient to use two boards, however. We only have one SDK-500 each, and if we can manage with it we would not have to write a second program!
One thing that is commonly done to test serial communication is to connect the output of the device back to its own input.
(This is called a "loopback" connection).

We could first run the program with the `MISO` input connected to Vdd or ground (checking to ensure that we received all 1's or 0's), then connecting the `MISO` input to the `MOSI` output.
We should then see as output whatever value was input from the switches.

We will use this idea again when we implement an asynchronous serial interface.

Atmel has an excellent Application Note on the setup and use of the SPI.

How could we make the previous example interrupt driven?
Thus is a simple device, so we only need to set the bit to enable SPI interrupts in the control word.
Simply replace the line setting `SPCR` in the initialization function to:

```
SPCR  = _BV(SPIE)|_BV(SPE)|_BV(MSTR);
```

This sets the SPI in interrupt mode.
Global interrupts also need to be set (at the end of the initialization function):

```
sei();
```

We also need to ensure that we don't try to send a second byte before the first is finished. For this, we would normally set a variable in the "send" function, and clear it in the interrupt handling routine.

TinyAVR devices generally do not have hardware for SPI.
They do have a serial interface (the Universal Serial Interface, or USI) which implements some of the function of the SPI in hardware.

How could we implement the SPI function in software?

## SPI controlled devices

There are many peripheral devices that use the SPI as control inputs. They include:

- programmable resistors
- analog to digital converters (ADCs)
- Digital to analog converters (DACs) (e.g., the Microchip MCP4921 12 bit DAC)
- RF communication devices (e.g., the Nordic Semiconductor NRF24l01 transceiver)
- display controllers (e.g., the Maxim Semiconductor MAX7221 LED controller)
- RFID scanners
- smart-card readers

We will briefly look at the data sheets for some such devices. (The data sheets are in the links page from the course home page.)

## Asynchronous Communication

In synchronous communication, the clock provides the necessary synchronization between the transmitter and receiver. As long as the clock information is available, data bits can be correctly transferred.

In asynchronous communication, the receiver has to provide the clock function using information from the transmitted signal. This usually involves some agreement on timing for each bit of the transmission (a fixed bit time, or baud rate) and a method for establishing the start of a transmission.

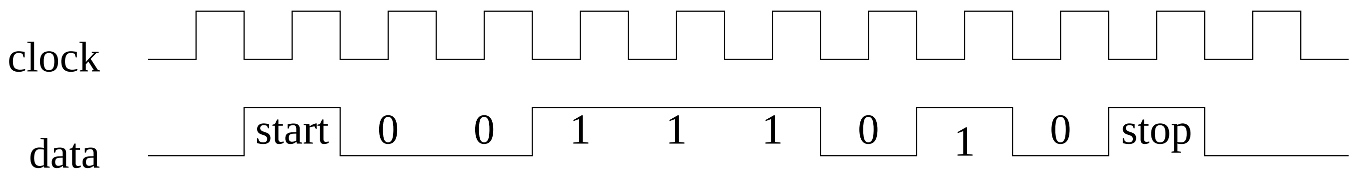
For simple asynchronous communication, the transmitting and receiving device must previously have established the data transfer rate (baud rate). Typically, one or more "start bits" is then sent, to synchronize with the internal clock of the receiver, and the data follows.

The send and receive clocks must be reasonably accurate if the data stream is long. (A 2% timing inaccuracy in each device limits the data stream to about 12 bits — the maximum permissible error is less than half a clock period.)

Some serial communication protocols (e.g., ethernet) require the extraction of clock information from the data stream. Typically, they would have a longer preamble for synchronization, and circuitry to recover clock information from the data.

A common asynchronous serial protocol is to send data in individual bytes, with one start bit, and one or two stop bits following the data byte. The data byte may also have a parity bit associated with it for error detection.

This is the data format commonly used with the **RS-232** standard for serial binary data. In fact, this standard does *not* specify the data encoding, but does specify the electrical signal characteristics such as voltage levels, signaling rate, timing, maximum stray capacitance and cable length.



Only the `data` is actually sent; the `clock` is internal to the transmitter.

Some systems accommodate automatic baud rate detection by sending a specific bit pattern, and measuring the time between two events (say, between two rising edges, or two falling edges). Most, however, do not.

Serial communication can also be either *simplex* (single direction) or *duplex* (simultaneous transmit or receive).
For example, SPI is inherently duplex, although some devices may only receive or transmit, depending on their function.

## The USART — asynchronous and synchronous communication:

The USART (Universal Synchronous/Asynchronous Receiver/Transmitter) is a very common peripheral for many computer systems.
It enables several different types of synchronous and asynchronous communication. Its most common function is to implement a standard asynchronous serial interface called **RS-232**.
The USART itself supports

- full duplex operation (independent serial receive and transmit registers)
- master or slave clocked synchronous operation
- high resolution baud rate generation
- serial frames with 5 – 9 data bits and 1 or 2 stop bits
- hardware odd or even parity generation and parity check
- data overrun and framing error detection
- noise filtering, with false start bit detection and a digital low pass filter
- three separate interrupts on TX Complete, TX Data Register Empty and RX Complete
- a multi-processor communication mode

We will first discuss the general operation of the device, then implement a simple asynchronous RS-232 port.

## The USART registers

The USART has a pair of data registers, `UDR0`, with a single address. There are separate registers for reading and writing, so the device can transmit data and receive data at the same time. As data is being transmitted, data can also be read.
The receive buffer consists of a two level FIFO. The FIFO changes state whenever the receive buffer is accessed.
There are three control registers for the USART, `USCR0A`, `USCR0B`, and `USCR0C`.

USCR0A:

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | RXC0 | TXC0 | UDRE0 | FE0 | DOR0 | UPE0 | U2X0 | MCPM0 |
| | R | R/W | R | R | R | R | R/W | R/W |
| Initial value | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

**Bit 7** `RXC0`**:** USART Receive Complete

This flag bit is set when there is unread data in the receive buffer and cleared when the receive buffer is empty (i.e., does not contain any unread data). If the Receiver is disabled, the receive buffer will be flushed and consequently the `RXC0` bit will become zero.

**Bit 6** `TXC0`: USART Transmit Complete

This flag bit is set when the entire frame in the Transmit Shift Register has been shifted out and there are no new data currently present in the transmit buffer (`UDR0`). It is automatically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location.

**Bit 5** `UDRE0`: USART Data Register Empty

This flag indicates if the transmit buffer (`UDR0`) is ready to receive new data. If UDRE0 is one, the buffer is empty, and therefore ready to be written. UDRE0 is set after a reset to indicate that the Transmitter is ready.

**Bit 4** `FE0`: Frame Error

This bit is set if the next character in the receive buffer had a Frame Error when received. I.e., when the first stop bit of the next character in the receive buffer is zero. This bit is valid until the receive buffer (`UDR0`) is read. The `FE0` bit is zero when the stop bit of received data is one.

**Bit 3** `DOR0`: Data OverRun

This bit is set if a Data OverRun condition is detected. A Data OverRun occurs when the receive buffer is full (two characters), a new character waiting in the Receive Shift Register, and a new start bit is detected. This bit is set until the receive buffer (`UDR0`) is read.

**Bit 2** `UPE0:` USART Parity Error This bit is set if the next character in the receive buffer had a Parity Error when received and the Parity Checking was enabled at that point (`UPM01` = 1). This bit is valid until the receive buffer (`UDR0`) is read.

**Bit 1** `U2X0:` Double the USART Transmission Speed

This bit only affects asynchronous operation. Writing this bit to one will reduce the divisor of the baud rate divider from 16 to 8 effectively doubling the transfer rate for asynchronous communication.

**Bit 0** `MPCM0:` Multi-processor Communication Mode

This bit enables the Multi-processor Communication mode. See the data sheet for details.

USCR0B:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| RXCIE0 | TXCIE0 | UDRIE0 | RXEN0 | TXEN0 | UCSZ02 | RXB80 | TXB80 |
| R/W | R/W | R/W | R/W | R/W | R/W | R | R/W |

**Bit 7** **`RXCIE0`:** RX Complete Interrupt Enable

Writing this bit to one enables interrupt on the `RXC0` Flag. A USART Receive Complete interrupt will be generated if the `RXCIE0` bit is written to one, the `RXC0` bit in `UCSR0A` is set, and global interrupts are enabled.

**Bit 6** **`TXCIE0`:** TX Complete Interrupt Enable

Writing this bit to one enables interrupt on the TXC0 Flag. A USART Transmit Complete interrupt will be generated if the `TXCIE0` bit is written to one, the `TXC0` bit in `UCSR0A` is set, and global interrupts are enabled.

**Bit 5** **`UDRIE0`:** USART Data Register Empty Interrupt Enable Writing this bit to one enables interrupt on the `UDRE0` Flag. A Data Register Empty interrupt will be generated if the `UDRIE0` bit is written to one, the `UDRE0` bit in `UCSR0A` is set, and global interrupts are enabled.

**Bit 4** **`RXEN0`:** Receiver Enable

Writing this bit to one enables the USART Receiver. The Receiver will override normal port operation for the `RxD0` pin when enabled. Disabling the Receiver will flush the receive buffer invalidating the `FE0`, `DOR0`, and `UPE0` Flags.

**Bit 3** **`TXEN0`:** Transmitter Enable

Writing this bit to one enables the USART Transmitter. The Transmitter will override normal port operation for the `TxD0` pin when enabled. The disabling of the Transmitter (writing `TXEN0` to zero) will not become effective until ongoing and pending transmissions are completed, i.e., when the Transmit Shift Register and Transmit Buffer Register do not contain data to be transmitted. When disabled, the Transmitter will no longer override the `TxD0` port.

**Bit 2** `UCSZ02` : Character Size

The `UCSZ02` bit combined with the `UCSZ01:0` bits in `UCSR0C` sets the number of data bits (Character SiZe) in a frame the Receiver and Transmitter use.

**Bit 1** **`RXB80`:** Receive Data Bit 8

`RXB80` is the ninth data bit of the received character when operating with serial frames with nine data bits. Must be read before reading the low bits from `UDR0`.

**Bit 0** **`TXB80`:** Transmit Data Bit 8

`TXB80` is the ninth data bit in the character to be transmitted when operating with serial frames with nine data bits. Must be written before writing the low bits to `UDR0`.

USCR0C:

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | UMSEL01 | UMSEL00 | UPM01 | UPM00 | USBS0 | UCSZ01 | UCSZ00 | UCPOL0 |
| | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Initial value | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

**Bits 7:6** `UMSEL01:0`: USART Mode Select

These bits select the mode of operation of USART0. See Table 19-4 in the datasheet. For our purpose, `UMSEL00` is 0 for asynchronous, and 1 for synchronous communication.

**Bits 5:4** `UPM01:0`: Parity Mode

These bits set the type of parity generation and check. If enabled, the Transmitter automatically generates and sends the parity of the transmitted data bits within each frame. The Receiver generates a parity value for the incoming data and compares it to the `UPM0` setting. If a mismatch is detected, the `UPE0` Flag in `UCSR0A` will be set. See Table 19-5 in the datasheet.

**Bit 3** `USBS0`: Stop Bit Select

This bit selects the number of stop bits (1 or 2, for `USBS0` = 0, 1 respectively) to be inserted by the Transmitter. The Receiver ignores this setting.

**Bit 2:1** **`UCSZ01:0`:** Character Size

These bits combined with the `UCSZ02` bit in `UCSR0B` set the number of data bits (Character SiZe) in a frame the Receiver and Transmitter use. See Table 19-7 in the datasheet.

**Bit 0** **`UCPOL0`:** Clock Polarity

This bit is used for synchronous mode only. used. It sets the relationship between data output change and data input sample, and the synchronous clock (`XCK0`). See Table 19-8 in the datasheet.

### The Baud Rate Registers

The other two control registers are the baud rate control registers, `UBRR0H` and `UBRR0L`.
They are effectively a single 12-bit register, which acts as a divider for the system clock to generate the required baud rate.
There are "standard" baud rates in common use; generally they are successive doublings of 1200 (2400, 4800, 9600, ...).
For these standard baud rates, and some common clock frequencies, the datasheet lists appropriate values for `UBRR0`.
These values can also be calculated readily, as:

$$\frac{\text{clock frequency}}{16 \times (\text{baud rate})} - 1$$

## USART example — asynchronous data, polled I/O

We will choose a baud rate of 9600, and a clock frequency of 8 MHz. (low clock frequencies give larger errors). In this case, the preceding equation gives a value for `UBRR0` of 51.0833, so choosing the value of 51 gives an error of $< 0.2\%$.

We will also choose to have an 8 bit word, no parity bit, and one stop bit.

This is a common configuration for RS-232 communication, although higher data rates can also be achieved.

Note that the datasheet claims a possible error of 10% in the clock frequency using the internal clock of the ATMega1284p, so it is possible that our baud rate calculation may not work. (It has worked for every case I have tried, so far, though.)

In order to achieve this, first we have to set the internal clock to 8MHz. Actually, this is the default, but it is divided by 8 to give 1MHz. with an internal clock divider. We can unset this under program control (more about this later).

There is a function in AVR LIBC to do this:

```
clock_prescale_set(clock_div_1);
```

Or it can be done writing to the clock control register directly:

```
   CLKPR = _BV(CLKPCE);   // enable clock scalar update
   CLKPR = 0x00;          // set to 8Mhz
```

Now we can set the baud rate (we will use different code later):

```
UBRR0H = 0;
UBRR0L = 51;  // same as UBBR0 = 51;
```

Now we can set the control registers:

```
/* no 2x mode, no multi-processor mode */
UCSR0A = 0x00;

/* interrupts disabled, rx and tx enabled, 8-bit data */
UCSR0B = _BV(RXEN0) | _BV(TXEN0);

/* Asynch. mode, no parity, 1-stop bit, 8-bit data */
UCSR0C = _BV(UCSZ01) | _BV(UCSZ00 );

 DDRD |= _BV(1) ; // enable tx output
```

Next, we will write functions to implement an interface to the US-
ART.
We will not use some of the code; it is there as an example.

First, the `include` file `usart0.h`:

```
#ifndef USART0_H
#define USART0_H

#include <stdint.h>
#include <avr/io.h>

void init_IO( void );

void usart0_init( uint16_t baud );

void usart0_put( uint8_t b );

uint8_t usart0_get( void );

int16_t usart0_get_delay( uint16_t max_delay );

#define BAUD_RATE(F_CPU, baud)
                 ((uint16_t)(F_CPU/(16L*(baud))-1))

#endif
```

Note that a formula for calculating the baud rate is defined. We do not actually use this; it is there as an example of a `define`.

Next, the USART initialization function:

```
#include "usart0.h"
#include <avr/power.h>

void
usart0_init( uint16_t baud )
{
    DDRD |= _BV(1) ; // enable tx output

    UBRR0 = baud;    // note the 16 bit assignment

    // no 2x mode, no multi-processor mode
    UCSR0A = 0x00;

    // interrupts disabled, rx and tx enabled, 8-bit data
    UCSR0B = _BV(RXEN0) | _BV(TXEN0);

    // Asynchronous mode, no parity, 1-stop bit, 8-bit data
    UCSR0C = _BV(UCSZ01) | _BV(UCSZ00 );
}
```

Next, the I/O initialization:

```
void
init_IO( void )
{
//  clock_prescale_set(clock_div_1);
    CLKPR = _BV(CLKPCE); // enable clock scalar update
    CLKPR = 0x00; // set to 8Mhz

    PORTC = 0xff; // all off
    DDRC  = 0xff;  // show byte in leds
}
```

Next, the `get` and `put` functions, similar to the standard C functions `getc` and `putc`.

```
void
usart0_put( uint8_t b )
{
    // wait for data register to be ready
while ( (UCSR0A & _BV(UDRE0)) == 0 )
;
// load b for transmission
UDR0 = b;
}

uint8_t
usart0_get( void )
{
// poll for data available
while ( (UCSR0A & _BV(RXC0)) == 0 )
;
return UDR0;
}
```

This will wait forever, if no character arrives.
Sometimes, we want a function that will end after some fixed time (a timeout).

The following function accepts a timeout (`max_delay`) as an argument. The timeout function is performed by a loop counter.

```
int16_t
usart0_get_delay( uint16_t max_delay )
{
// poll for data available, with timeout
while ( (UCSR0A & _BV(RXC0)) == 0  && max_delay != 0) {
max_delay--;
}
if ( (UCSR0A & _BV(RXC0)) == 0  ) {
return -1;
}
return UDR0;
}
```

Next, the main program:
It accepts input from a terminal, and echoes the value entered back to the terminal. If it is a letter, it changes the case from lower to upper or upper to lower. (There are C functions to do this, but the code here makes use of a property of the ASCII code.)
It also displays the input character in the LEDs through PORTC.

```
int
main( void )
{
    uint8_t b;
    init_IO();
    usart0_init( 51 ); // 9600 baud at 8MHz

    while ( 1 ) {
        b = usart0_get();
        PORTC = ~b;
        if ( b >= 'A' && b <= 'Z' ) {
            b ^= 0x20; // convert to lower case
        }
else if ( b >='a' && b <= 'z') {
            b &= 0xDF; // convert to upper case
        }
        usart0_put( b );

        if ( b == '\r') { // add a newline after return
            b = '\n';
            usart0_put( b );
        }
    }
    return 0;
}
```

In order to use this program with a terminal, it is necessary to have inputs that are compatible with the standard terminal outputs.
The SDK-500 has a second serial port, labeled `RS232 SPARE` for this purpose. The input and output connections to this from the board are also labeled `RS232 SPARE`, and are in the row of pins between switches `SW3` and `SW4`. They can be connected to the port pins `RXD0` and `RXD1` (pins `PD0` and `PD1`).

The RS-232 standard requires that the voltage levels for 1 and 0 be approximately -12V and 12V; much higher than standard logic levels. Generally, voltages of -5V and +5V would be useable, but even those are outside the levels produced by logic devices.
There are ICs available which take normal logic levels (0 – 5V or 0 – 3.3V) and produce the required voltage levels for RS-232. Perhaps the most popular are the MAX232 series from Maxim Semiconductor. A typical datasheet is available in the links page.

## Storing data in the program space (flash memory)

It is possible to store constant data in the program memory space. In fact, when variables are initialized, the initialization values have to be stored in program memory, then copied to RAM every time the program is started.

Since the ATMega128p has much more program memory (128 K of flash memory) than RAM (8K bytes), it is useful to store constant data in the program memory space. In assembler, the register pair `Z` (registers 30 and 31) can be used as a pointer to program memory. This can allow data stored in program memory to be read.

AVR Libc has functions which provide interfaces for access to data stored in program space (flash memory). These "Program Space Utilities" are contained in `<avr/pgmspace.h>` and provide access to characters, 16 bit words, and 32 bit words. Additionally, there are functions to manipulate strings stored in flash memory. In fact, many of the standard string functions have a variant dealing with program space memory.

The basic function for reading a byte from program memory is
`pgm_read_byte(address)`
where `address` is the address of the byte in program memory.

Typically, data is stored in program memory as type `const`, usually in an array. Following is an example of about 2 seconds of music, stored as a duration (time, about 0.01 s.), followed by a note (value for a timer/counter compare register):

```
const uint8_t mary[] PROGMEM = {  // mary had a little lamb
32,127,32,142,32,159,32,142,28,127,8,1,28,127,
8,1,60,127,0,1};
```

This defines an array called `mary[]` of unsigned 8-bit integers in program memory. As for C arrays in RAM, `mary` is a pointer to the start of this array.
Typically, the first two elements of this array would be read as:

```
time = pgm_read_byte(mary);
note = pgm_read_byte(mary+1);
```

Using arrays rather than pointers,

```
time = pgm_read_byte(&mary[0]);
note = pgm_read_byte(&mary[1]);
```

Recall that the argument to `pgm_read_byte` is an address, or pointer.

## Strings in program space

For an example of one of the program space string functions, consider the standard string compare function,
`int strcmp(const char *s1, const char *s2)`.
which compares two strings, and returns an integer $< 0$ if string `s1` is lexicographically less than `s2`, $= 0$ if they are equal, and $> 0$ if `s1` is lexicographically greater than `s2`.
The program space equivalent is
`int strcmp_P(const char *s1, PGM_P s2)`
where `PGM_P` is defined as `const prog_char *` — a pointer to a character in program memory.

There are similar program memory variants for many of the common string functions.
A common way to define program memory strings is to use `PSTR(s)` which is defined as `((const PROGMEM char *)(s))`.

These functions (and character strings in program memory) are often used when communicating with terminal I/O devices.
In particular, in using the C standard I/O library (`<stdio.h>`), constant character strings are common.

See the AVR Libc documentation for more details.

## The AVR-gcc standard I/O library (`stdio.h`)

The C language defines a **`stdio`** library that performs standard input and output. Typically, the default input and output devices are the user's keyboard and monitor (devices which handle character I/O). Embedded systems don't usually have those, so input and output functions must be defined for whatever devices are available.

*Streams* are an abstraction between the program and an I/O device. This facilitates defining common methods of sending and receiving data amongst the various types of devices available.
There are two types of streams: text and binary. The AVR-gcc library does not really distinguish between them.
In C, normally the standard streams **`stdin`**, **`stdout`**, and **`stderr`** are predefined as the keyboard and monitor. In AVR-gcc, those devices are not present, so they are not provided.

It is possible, however, to define functions with appropriate properties to provide the **`stdio`** functions. In fact, the **`usart0_get()`** and **`usart0_put()`** functions from the earlier example are fine. They merely need to be associated with the appropriate stream (file buffer). AVR-gcc provides a facility for this.
The macro **`fdev_setup_stream(stream, put, get, rwflag)`** is provided to do this.
**`put`** and **`get`** are output and input functions, respectively.
**`rwflag`** determines whether the stream is for input, output, or both.

There is an alternate form, `FDEV_SETUP_STREAM()` in which all data initialization happens during C start-up.

A typical use is as follows:

```
static FILE my_stream =
FDEV_SETUP_STREAM (my_putchar, my_getchar, _FDEV_SETUP_RW);
```

The following associates `stdin` and `stdout` with the stream:

```
stdout = &my_stream;
stdin = &my_stream;
```

## Formatted I/O in C

The standard C library provides routines to parse input and output streams and perform numeric conversions for input and output.
For example, the following example parses the `stdin` stream to find a decimal integer followed by another decimal integer:

```
scanf( "%d %d", &x, &y );
```

Note that the arguments here are a text string (the *format string*), and *pointers* to the values to be input. Why?
The following sends a stream of three decimal integers to `stdout`:

```
printf( "%d %d %d\n", x, y, x+y );
```

Here, the arguments are the format string and the values.

The C `stdio` library has several variants of these functions.
The AVR-gcc library has a very useful extension to the `stdio` library which allows the format string to be stored in program (flash) memory:

```
scanf_P( PSTR("%d %d"), &x, &y );
printf_P( PSTR("%d %d %d\n"), x, y, x+y );
```

This is useful for the (usual) case where the format string is a constant value.

The variant which parses a character string is particularly useful (in the following, `buf` is a character array):

```
sscanf_P( buf, PSTR("%d %d"), &x, &y );
sprintf_P( buf, PSTR("%d %d %d\n"), x, y, x+y );
```

Here, `buf` is an array in read/write memory, and the (constant) format string is in program memory.
Following is a simple example of using the `stdio` library. It uses some of the functions we have defined earlier.
The program takes input from a terminal on `usart0` (`stdin`), sends it through the SPI port, and returns the value received from the SPI port to the terminal (`stdout`).

**First**, the `include` file:

```
#include <stdio.h>
#include <avr/io.h>
#include <util/delay.h>
#include <avr/pgmspace.h>

#include "spi_poll.h"
#include "usart0.h"
```

**Next**, the initialization function:

```
void Init()
{
    clock_prescale_set(clock_div_1)
    // CLKPR = _BV(CLKPCE); // enable clock scalar update
    // CLKPR = 0x00; // set to 8Mhz
     _delay_ms(30);

    DDRC = 0XFF;    // set port C to output
    usart0_init(); //open UART0
    Init_Master(); //open SPI
}
```

**Next**, set up the `stdio` streams:

```
static int my_putchar(char c, FILE * stream)
{
        if (c == '\n')
                usart0_put('\r');
        usart0_put(c);
        return 0;
}

static int my_getchar(FILE * stream)
{
        return usart0_get();
}

int8_t usart0_rx_data_ready()
{       // test for data available
        return (UCSR0A & _BV(RXC0));
}


static FILE my_stream =
FDEV_SETUP_STREAM (my_putchar, my_getchar, _FDEV_SETUP_RW);
```

**Next**, the main program:

```
int main()
{
    unsigned char data; // register to hold sent
                        // and received char
    Init();
    stdout = &my_stream;
    stdin = &my_stream;
    printf_P(PSTR(" type character to send \n"));

    //main program loop
    while(1)
    {
        // check UART status register to see if data has
        // been received.  If so, process.
        while(usart0_rx_data_ready())
        {
            data = getchar(); // get data from UART
            PORTC = data;     // display data on LEDs
            data = Master_Send(data);  // send/receive data
            printf("%c", data); //print the received data
         }
    }
}
```

For further examples, please consult the documentation for AVR-gcc.

There is a more complete example in the Examples section of the documentation. Specifically, "Using the standard IO facilities."
A modified version (for the ATMega1284p) is available through the local link.
Both examples use LCD displays with a HD44780 LCD controller. A driver for this controller is implemented in the code.

The `stdio` library is rather large, and since most products target the smallest possible microcontroller (with the least memory) these functions are not used often.
Typically, a function that directly manipulates the USART, and customized for the specific application, would be written.

The most common use of the `stdio` functions is during the code development stage, for debugging.

## The HD44780 LCD controller

Character based LCD displays are a common display device for microcontrollers and embedded systems. They are low cost, relatively easily interfaced to a microcontroller, and use little energy.

The most common small text displays (typically 2–4 lines of 8 to 40 characters) use the HD44780 LCD controller, or an equivalent. This controller supports a dot matrix character display of $5 \times 8$ pixels. (It also supports a dot matrix character display of $5 \times 10$ pixels, but this is rarely used.)

The HD44780 uses a parallel bus, with several (3) control signals. It operates in two data modes, transferring either 4 or 8 bits at a time. Internally, there are rows of data memory (called `DDRAM`), and character generator memory (`CGRAM`). Rows have a start address of `0x00`, `0x40`, `0x14`, and `0x54` respectively, for a 4 row display. Characters in a row increment from the row start address, so address `0x17` is the 4th character in row 3.

The controller has a simple "language" with about 8 "instructions." It also has timing constraints for writing data into the memory, and reading data from the memory.
There are several good web pages related to the programming of the HD44780 and equivalents, as well as drivers for the device.

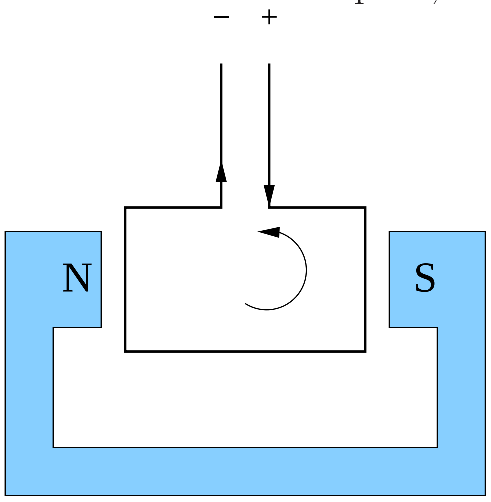
## Error detection and correction

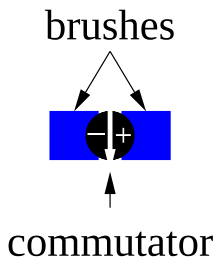
One of the “problems” with serial data communication is that it is often used in a “noisy” environment — there may be single or multiple bit errors in the transmission.
You may have noted that the serial interfaces do *not* have error detection or correction built in. (The USART supports the use of a *parity bit* for error dection, but it is rarely used.)
Basically, error detection or correction is achieved by redundancy. If you communicate using any “standard” protocol, the type of redundancy is specified.
Commonly, error checking is done with a *cyclic redundancy check* (CRC) polynomial. The protocol would also specify what happens in the case of a failed checksum — how retransmission is requested, etc.
Most higher speed serial communication standards (e.g., ethernet) have some error detection protocol defined as part of the standard.

The avr-libc modules contain some 16 bit CRC implementations for common CRC polynomials. See `<util/crc16.h>`

## DC Motors

When an electric current flows around a loop, it generates a magnetic field. One direction (perpendicular to the loop) will be magnetic North, the other, South. If the coil is placed between the poles of a permanent magnet, it will twist (or rotate) with the North attracting the induced South pole, and vice versa.





Making several coils of wire around some material that can be easily magnetized (e.g., iron) increases the force of this attraction.

If we have a way of changing the direction of flow of the current in the coil, then we can keep the coil moving in one direction. This can be done in several ways; the simplest is to use a *split ring* or *commutator* so that the winding on one side is always powered from the same side of the supply. The split ring commutator is connected to the coil by *brushes*.

Most permanent magnet DC motors have many loops of wire, wound around two or three iron cores, and spin very fast. See `http://en.wikipedia.org/wiki/Electric_motor` for more information.

## Properties of DC Motors

Small DC motors have several interesting properties:

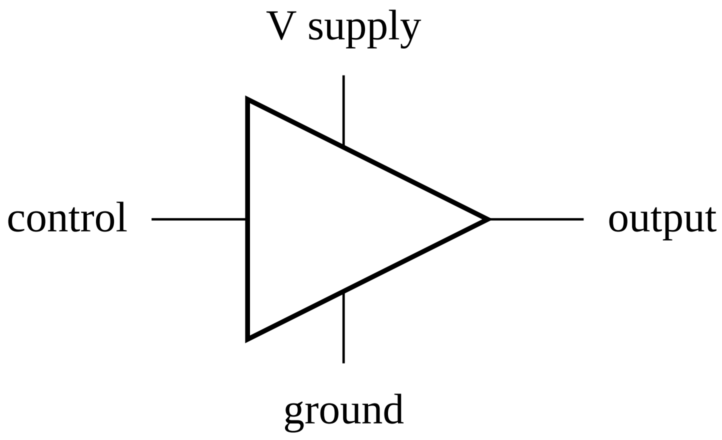
- Reversing the voltage will reverse the rotation direction.
- A motor will spin faster as the voltage is increased.
- The current consumed by a motor drops as the speed increases.
- A stalled motor consumes the maximum current — it is then a resistor.
- A motor's speed can be controlled with PWM, which effectively creates an average voltage.

Of course, a permanent magnet motor is also a generator, if no voltage is applied to the motor.
Even when the motor is running, a *back emf*, or voltage, is generated. The effective voltage supplied to the motor is reduced because of this. This becomes important when switching the motor on and off, because this backwards voltage can damage the circuitry connected to the motor. (This happens for any magnetic, or inductive, element in a circuit.)
When we use a motor in a circuit, normally we provide a ground path for this back EMF with a diode. Some motor drivers have such diodes built in.

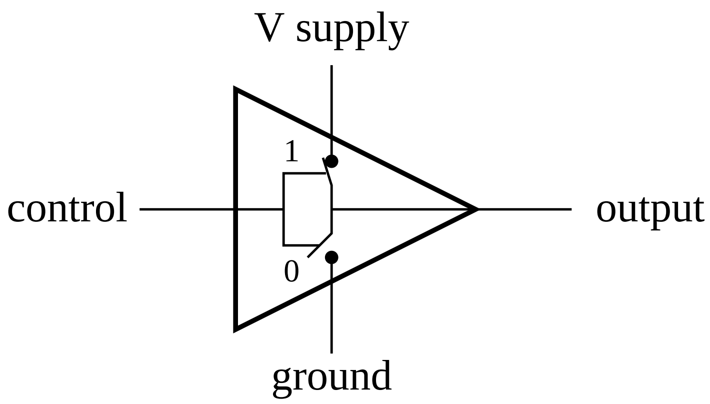
## Motor driver circuits

Most processors can source or sink a limited current; typically about 40 ma. Therefore, addition circuits, called driver circuits, are required to drive higher current loads.

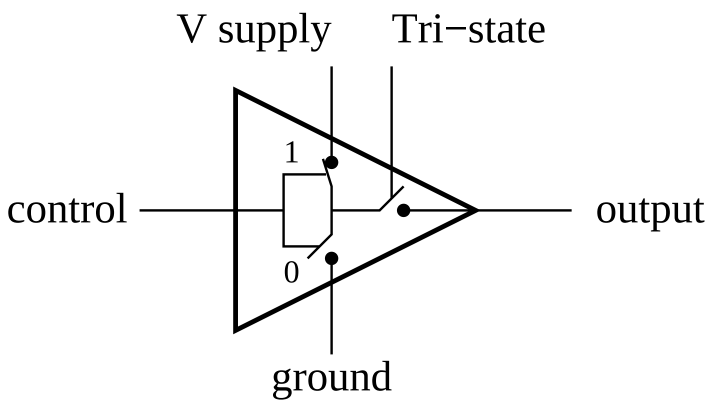
A *buffer* is a device which provides higher voltage and/or current to a device than its control input can supply. They are usually drawn as follows:



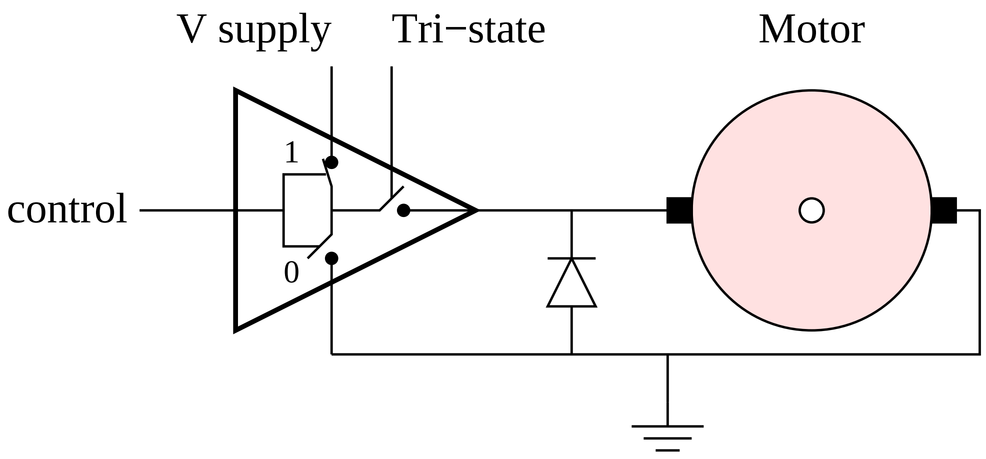
Internally, a switch representation is:



Some buffers may also be *tri-stated*. An output in tri-state is electrically disconnected from the load.



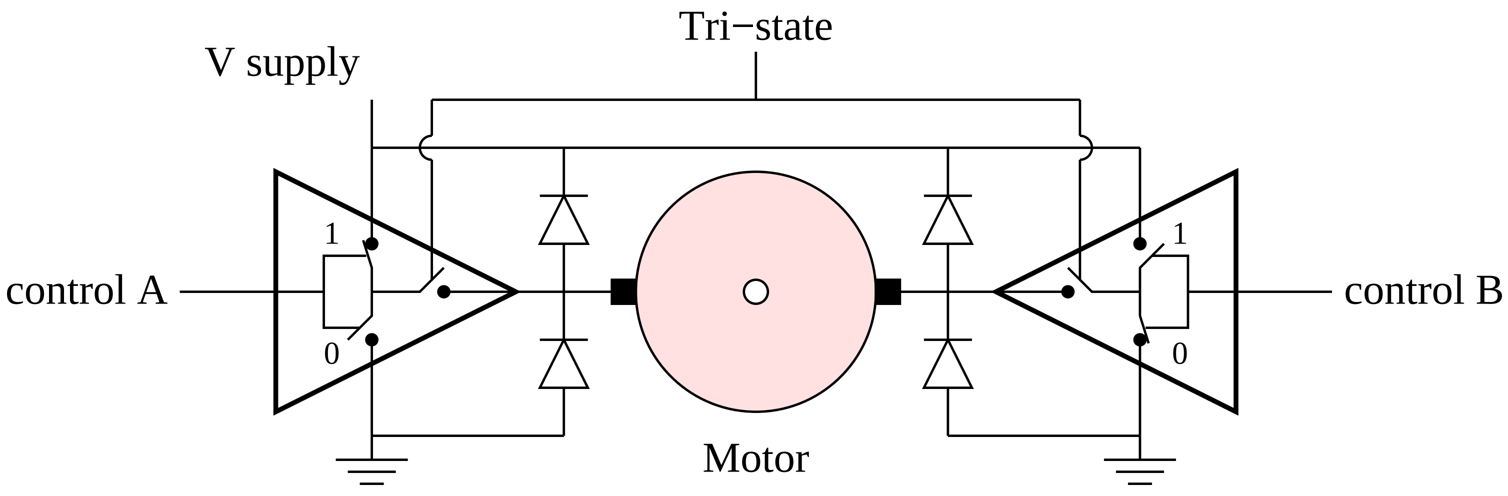
A typical application, controlling a motor, would be:



This configuration allows the speed of the motor to be controlled by PWM. Note the diode to protect the buffer from the back EMF when the output is off or tri-stated.

Here, the motor can only rotate in one direction. If we could switch the inputs, we could have the motor rotating in either direction — bi-directional control.

With two buffer/drivers, we can have bi-directional control. The following configuration is called a H-bridge:



The controls are:

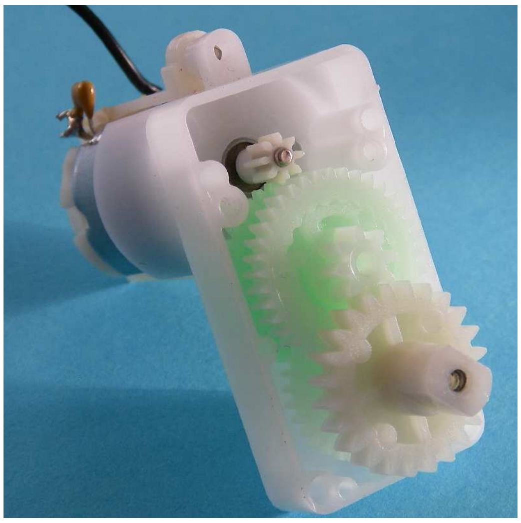
- A=0, B=0: motor brakes
- A=0, B=1: motor turns in one direction
- A=1, B=0: motor turns in the other direction
- A=1, B=1: motor brakes
- Tri-state: motor coasts

In this configuration, the motor can be controlled using PWM, by setting the direction, and pulsing the tri-state input.
The pulses would have to be fast enough to keep the motor running smoothly.

Small permanent magnet DC motors normally spin at high speed (over 1000 RPM). Typically, they are geared down by a factor of 100 or more, using a train of gears. Servo motors are similarly geared down, in order to increase the torque they can provide.

The particular motor we will use in the lab has a nested gear train with a total reduction ratio of 224 to 1. With this reduction, it rotates at 38 RPM (1.6 seconds per revolution).
It operates at 5V DC, draws about 50 ma. free running, 600 ma. stalled, and produces about 50 oz.-in. torque.

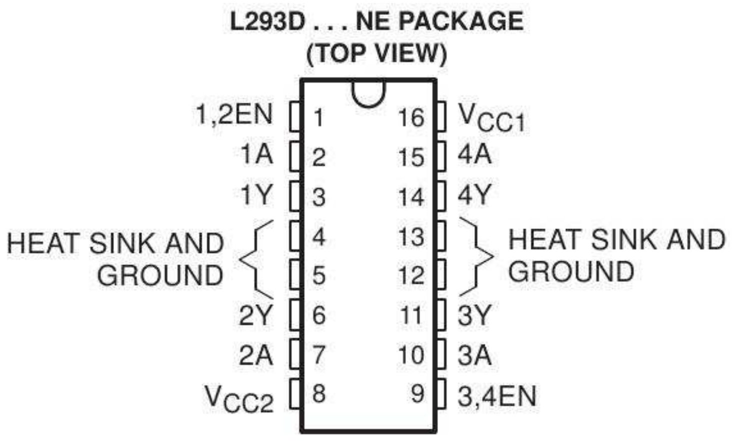
A picture of its gear train follows:



In this picture, the front cover is removed, and you can see most of the gear train, and the attachment point for a wheel.

## The L293D quad half-H driver

The L293D driver is an integrated circuit containing four tri-state buffers similar to those described in the previous diagrams.
In fact, it has two pairs of buffers, each pair sharing a tri-state input (enable, or `EN`).

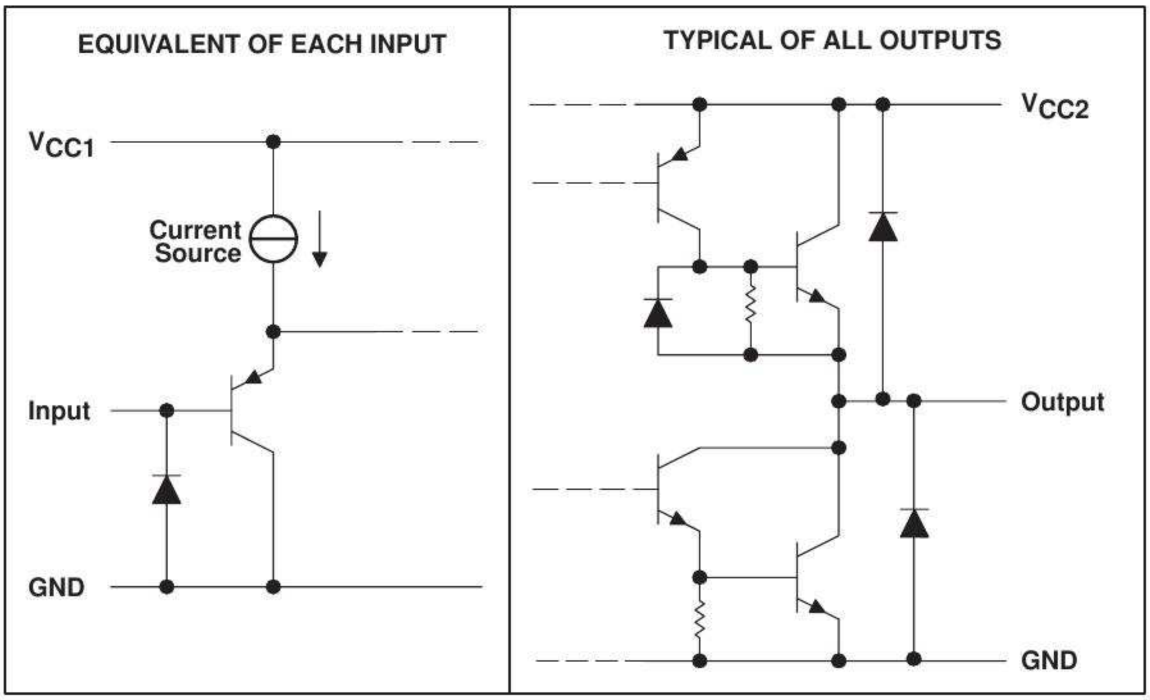


| INPUTS | | Output |
|---|---|---|
| A | EN | Y |
| H | H | H |
| L | H | L |
| X | L | Z |

where Z is tri-state (high impedance).

This device can supply up to 600 ma. and has separate voltage inputs for logic ($V_{CC1}$) and device ($V_{CC2}$). The voltage supplies are separate, but they share a common ground connection. It also has built-in diodes, as in the previous diagram for the H-bridge.

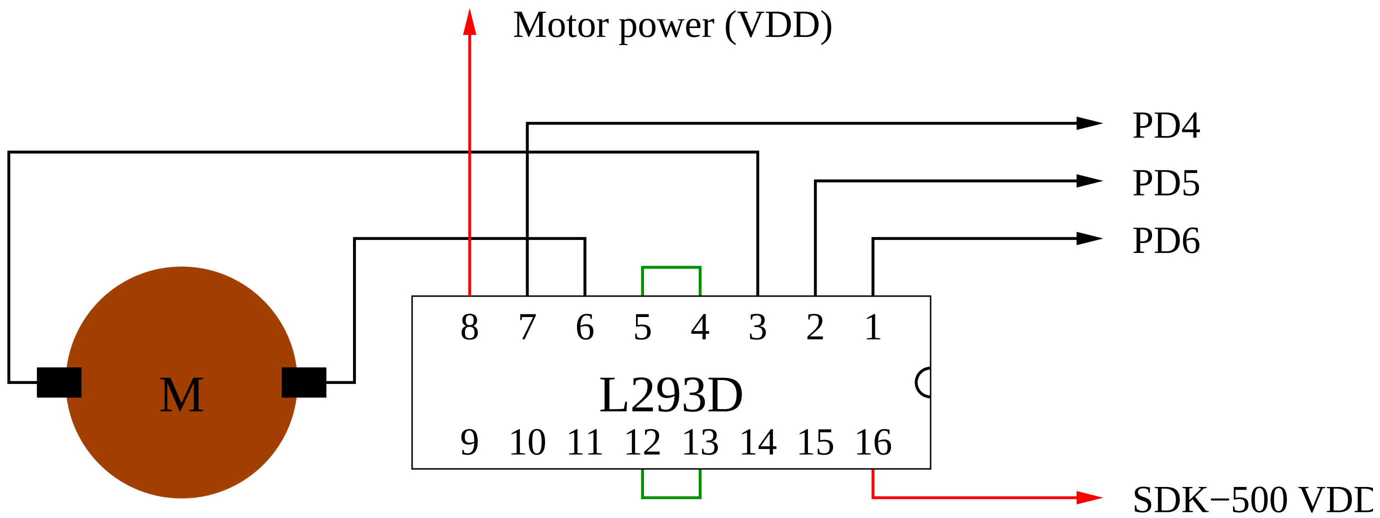
Typical input and output circuits for this device is shown in the following diagram. The transistors here behave as switches, so the output circuit is very similar to the switch circuit shown earlier:



Note the separate power supplies ($V_{CC1}$ and $V_{CC2}$) for input and output. Also, note the protection diodes.

The pairs of transistors in the upper and lower part of the output stage are equivalent to a pair of complementary switches.

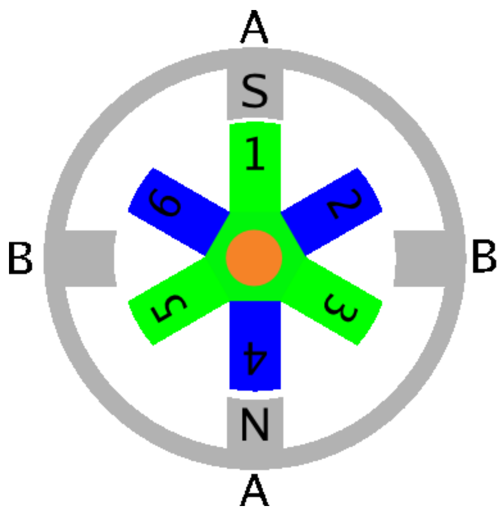
The following diagram shows a L293D H-bridge driver configured to control a DC motor:



Note that one L293D can control two DC motors, without requiring any other components like bypass diodes.
This device is a popular controller for robotic devices.

## Stepping motors

A stepping motor moves from one stable position to the next stable position.
The diagram following depicts a stepping motor with 6 rotor poles and 4 stator poles. The rotor is constructed with permanent magnets. The green poles are North, and the blue poles are South. The North poles are above the South poles, and the magnet is aligned with the shaft.
The stator poles labeled A are actually the same coil of wire. The B poles are another distinct coil. The A poles form an electro-magnetic that has two polarities, either N-S or S-N depending on the direction of the current in the coil. The same is true for the B poles.



Stepping motor with two coils/four wires are called bipolar stepper motors. How many polarities are possible? How many positions are stable?

## Motion of a stepping motor - steps

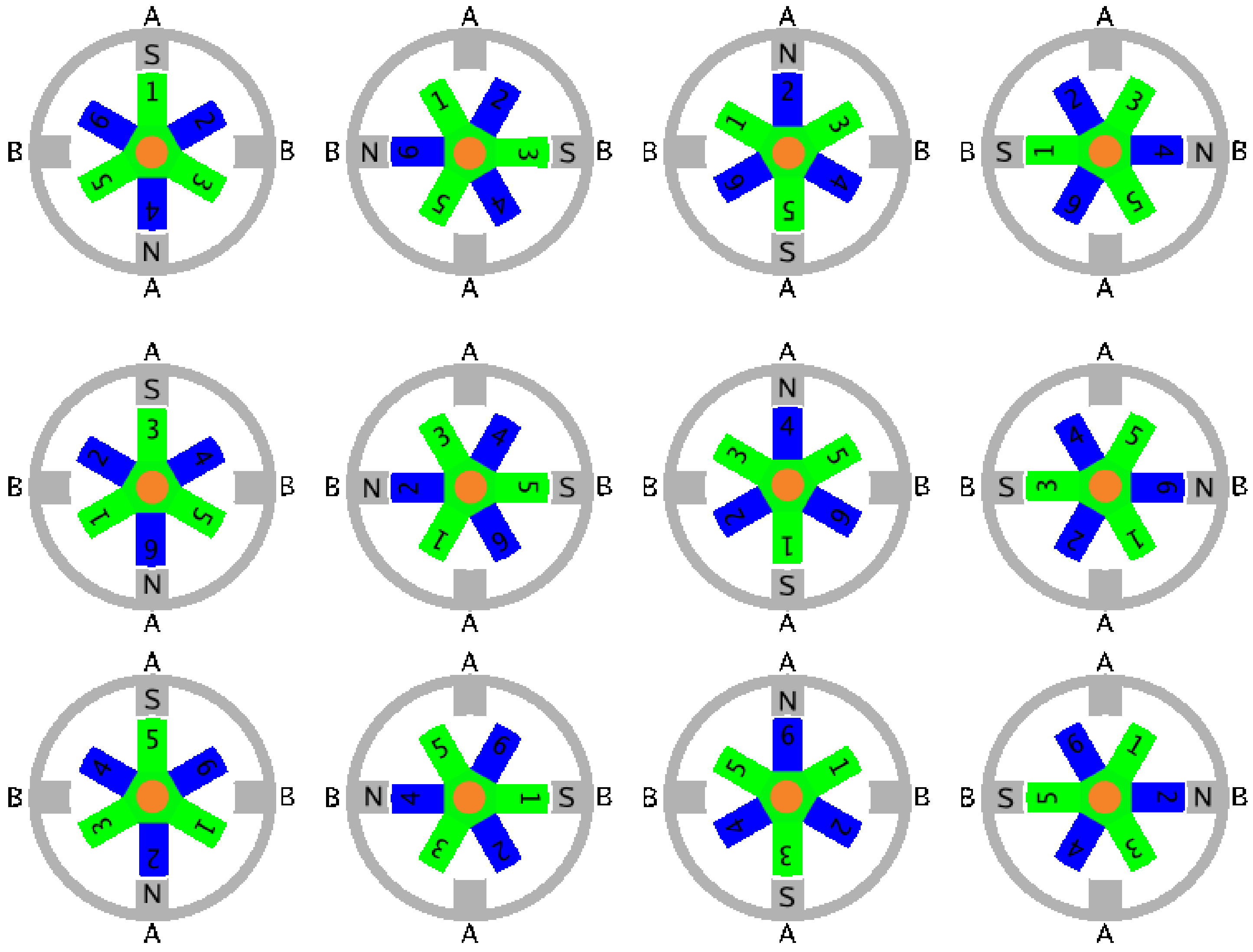
The rotor in a stepping motor moves when the polarity of the stator poles are changed (i.e., the current direction is changed.)
A possible set of steps form a initial position where A is energized S-N (top-bottom) is:

- Coil A is de-energized and coil B is energized N-S (left-right). Since the right pole is S, the closest north pole (3) rotates counter-clockwise towards it. The closest south pole (6) rotates towards the left north pole. The rotor has rotated 30 degrees in the counter-clockwise direction.
- To rotate a further 30 degrees counter-clockwise, coil B is de-energized, and coil A is energized N-S.
- The next counter-clockwise rotation is achieved when coil A is de-energized, and coil B is energized S-N.

Notice that the rotor can rotate clockwise or counter-clockwise, depending on the polarity the other coil when it is energized.
The following diagram shows a full sequence of these steps. They are called *half steps*, for a reason we shall see shortly.

## Half step positions for a 6 rotor/4 stator stepping motor:

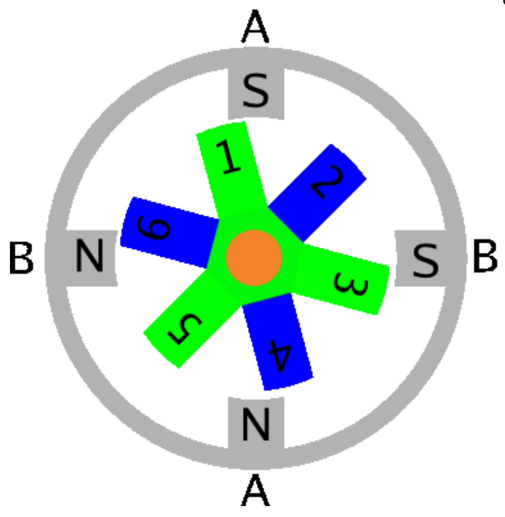
This stepping motor will complete a full rotation after 12 steps. The configuration of the coils is repeated three times.



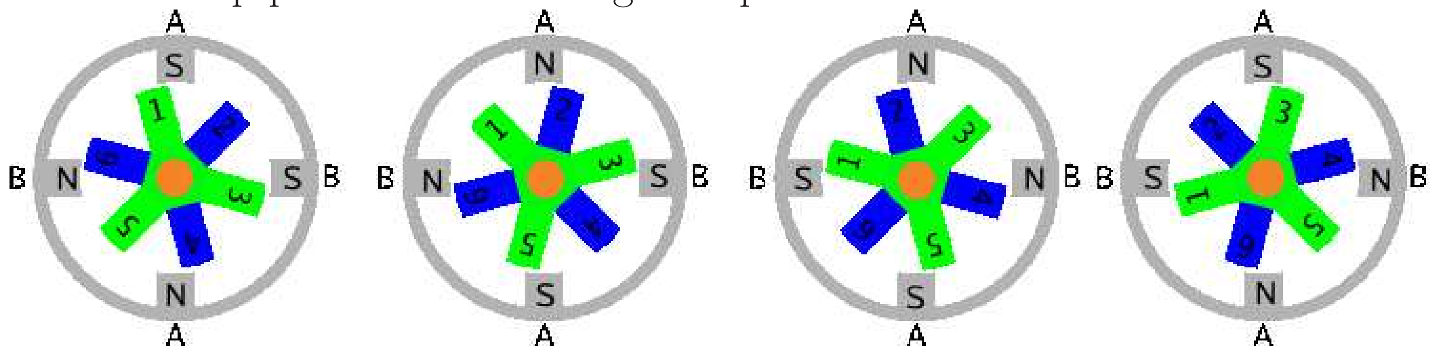
In this example only one coil is energized at a time; this called the half step position.
A full step position happens when both coils are energized.

**Full step positions:**

Both coils are energized in a full step position. The holding torque for a full step is more then for a half step, because two coils are energized instead of one coil.
In the following diagram, pole 1 is attracted to the top pole, and pole 3 is attracted to the right pole.



The closest full and half step positions differ by 15 degrees.
The next full step position is reached by switching the polarity of one of the coils.
Notice that the rotor moves through a half step position to get to the next full step position. For this configuration, the nearest full and half step positions are 15 degrees apart.

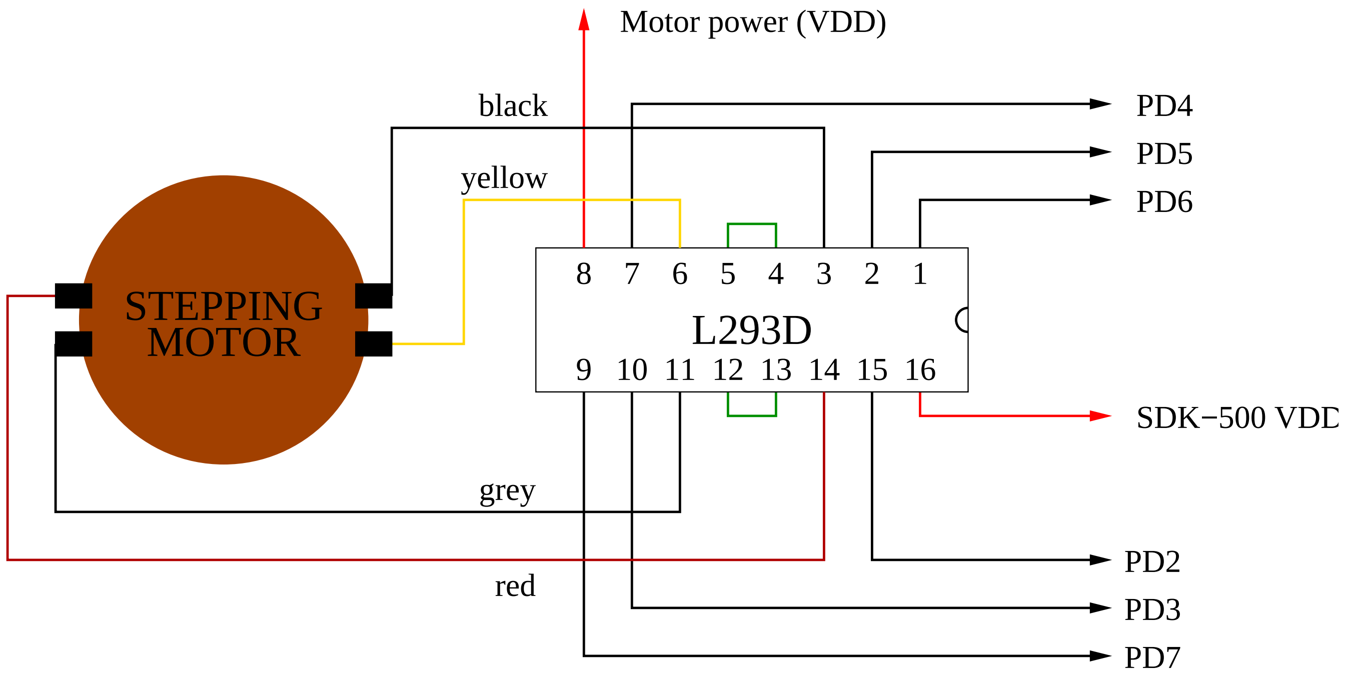


## Driving A Bipolar Stepper with the L293D

Recall that the DC motor had only one coil, and used two of the drivers from the L293D to achieve bi-directional operation.

A bipolar stepping motor has two coils, so it can be controlled using all four of the drivers from the L293D.

A combination of full and half steps can be used to increase the total number of available steps.



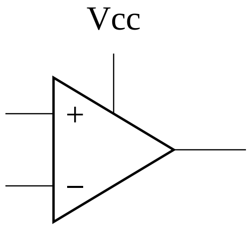
Both enable lines can be set to 1 for full steps, or one can be set to 0 for half steps.

There are only two possible polarizations for each coil, (0,1) or (1,0).

## The Analog comparator

An analog comparator simply compares two analog (voltage) inputs, and outputs a 1 or 0 depending on whether or not one of the inputs (usually called the positive input) is greater than the other (negative) input.

The circuit symbol for a comparator is the following:



There is one analog comparator on most AVR processors, and typically the direct inputs are labeled `AIN0` (+ input) and `AIN1` (− input.)

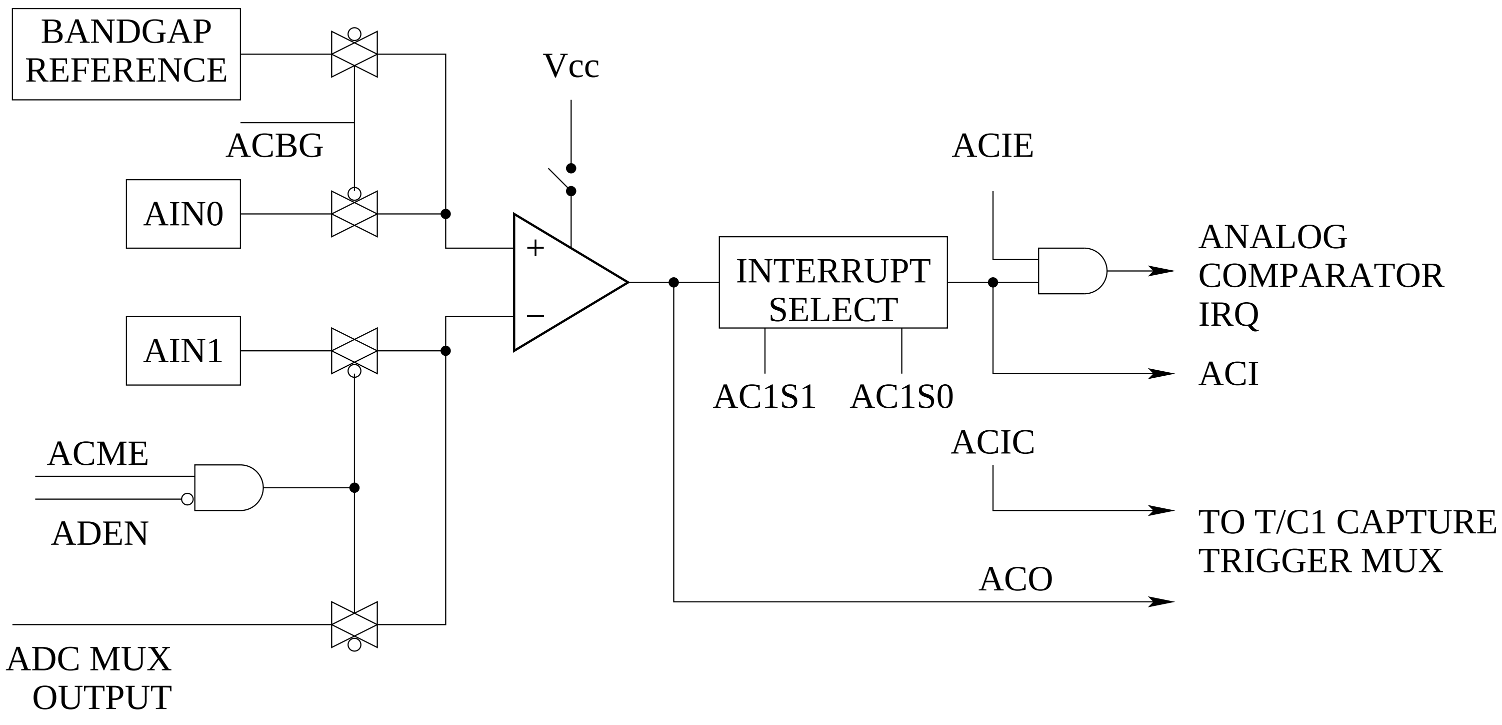
Also, typically, it can be programmed to have inputs from several different sources.

Normally the device has one or more "reference voltages" which can be programmed for use as the + input.

The most common of these is a "bandgap reference" which has a stable output of approximately 1.1 volts. It is also used by the brown-out detector.

It is not operational all the time, (it is started up on first use) and has a startup time of about 40 $\mu$s. (Refer to table 25.6 on page 331.) This typically means that the first comparison is incorrect, and should be discarded.

Following is a schematic of the analog comparator internal connections in the ATmega1284p:



Note that there are two potential inputs to the + terminal of the comparator, `AIN0` or the bandgap reference.

The − input also has two sources, the `AIN1` pin, and a multiplexer which is shared by the analog-to digital converter. Use of this MUX is controlled by bit 6 of the register `ADCSRB`. Setting it to logic 1 enables it for use with the comparator, provided the ADC is switched off by setting the `PRCA` bit to 0 in the power reduction register (`PRR`).

The output from the comparator can generate an interrupt, or trigger the "input capture" function in timer/counter 1.

This is useful for measuring the time from some initial action until the comparator is triggered.

## Analog-to-digital conversion

The AVR processors also invariably have an analog-to-digital converter, with 10 bit resolution. (Some older models have only 8 bit resolution, and the newest Xmega devices have 12 bits.)
The ADC can operate in two modes — differential and single-ended.
In single-ended mode, the ADC digitizes the voltage difference between the single input and analog ground, relative to the chosen reference.
In differential mode, the ADC digitizes the difference between the two inputs, relative to the chosen reference.
In differential mode, the differential signal can be amplified by a factor of 10 or 200 before it is digitized (with some reduction in accuracy).

The ADC requires a clock input of between 50KHz and 200KHz in order to provide the full 10 bit accuracy, and has an on-chip prescaler for the system clock to select an appropriate clock frequency.
It requires 13 ADC cycles for a normal conversion, for a maximum conversion rate of about 15K samples/s. ($65-260\mu$s per conversion).

The ADC is a successive approximation converter, with an on-chip "sample and hold" circuit which holds the input at a constant voltage for the duration of a conversion.
Figure 23.1, on page 244 shows a schematic of the ADC in the ATmega1284p.

The ADC can use several sources for its reference voltage — the `AREF` input pin, the analog supply voltage (`AVCC`), the 1.1 V bandgap reference, or a 2.56 V internal reference.

The ADC is enabled by setting the ADC enable bit (`ADEN`) in register `ADCSRA`.
A single conversion is started by writing 1 to the ADC start conversion bit, `ADSC`. This bit will stay at 1 until the conversion is completed, at which time it will be set to 0 by internal hardware.
The ADC can be programmed to start conversion by "auto triggering" on an interrupt source, or to interrupt the processor after the AD conversion has completed.

The registers `ADCSRA` and `ADCSRB` are the control and status registers for the ADC, register `ADMUX` controls the input source, the reference voltage selection, and the gain (for differential mode).
It also determines how the output is represented, depending on the value of bit 5 (`ADLAR`). If this bit is 1, the output registers `ADCH` and `ADCL` are left justified, otherwise they are right justified.
If only 8 bits are required, it is usual to set `ADLAR` to 1, and only read `ADCH`.
The default reference voltage is the value at the `AREF` pin.
Page 255, tables 20.3 and 20.4 show the appropriate bits for reference voltage and input selection to the ADC, respectively.

The register `ADCSRA` has the following structure:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| ADEN | ADSC | ADATE | ADIF | ADIE | ADPS2 | ADPS1 | ADPS0 |

Writing 1 to bit 7 (`ADEN`) enables the ADC.

Writing 1 to bit 6 (`ADSC`) starts a conversion.

Bit 5, `ADATE`, enables automatic triggering from the source selected by the `ADTS` bits in `ADCSRB`.

Bit 4, `ADIF`, is set when an AD conversion is complete. It is cleared by hardware if the corresponding interrupt handler is executed, or by writing a 1 to it.

If bit 3, `ADIE` is set, and global interrupts are enabled, the ADC Conversion Complete interrupt is generated when ADIF is set by completing a conversion.

Bits 2–0, (`ADPS2, ADPS1, ADPS0`) are the ADVC prescaler bits, which determine the division factor between the clock frequency and the ADC. The factors range between 2 and 128.

The ADC frequency should be between 50K and 200K cycles/s.

For a 1 MHz clock, typically division by 8 (bits set to `011`) is used, giving an ADC frequency of 125 KHz.
See Table 21.5, p.250 for the ADC prescaler values.

The register `ADCSRB` has the following structure:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| - | ACME | - | - | - | ADTS2 | ADTS1 | ADTS0 |

For `ADCSRB`, bit 6 (`ACME`) is the analog comparator multiplexor enable bit; it must be set to 0 for the ADC to use the input MUX.

Bits 2–0 (`ADTS2, ADTS1, ADTS0`) select the auto trigger source. Refer to Table 20.6 on page 259.

The register `ADMUX` has the following structure:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| REFS1 | REFS0 | ADLAR | MUX4 | MUX3 | MUX2 | MUX1 | MUX0 |

Bits 7–6 (`REFS1, REFS0`) select the reference voltage. See Table 20.3 on page 255.

Bit 5 (`ADLAR0`) selects whether or not the output, registers `ADCH` and `ADCL` are left justified.
Bits 4–0 (`MUX4 ...0`) select the input source. For differential inputs, they also select the gain.

## Example code for 8-bit analog to digital conversion

The following initializes the ADC:

```
/* enable the ADC and start the first (dummy) conversion
   ADC clock set to divide by 8 (125 KHz at 1 MHz)  */

ADCSRA = ( 1 << ADEN  ) | ( 1 << ADSC  )
         | ( 1 << ADPS1 ) | ( 1 << ADPS0 ); // divide by 8

/* busy wait for conversion to complete */

while ( ADCSRA & ( 1 << ADSC ) );
```

Something like this would normally be done in the initialization function, if the code required frequent use of the ADC.
Note that the default input source/reference voltages are used. What is actually used is irrelevant, because the result is not actually read.

The STK 500 provides a reference voltage for the ADC, which can be set independently from the supply voltage — it must be set to less than the supply voltage.
It can be disconnected from the processor by removing the `AREF` jumper on the STK 500 board. This *must* be done if any of the internal references are used.

The following returns a single 8-bit ADC conversion on the specified input.
It returns only the high byte of the output, in left-justified mode.

```
/*  ADCIN - a function for 8-bit ADC
    on a specified input channel */

uint8_t ADCIN( uint8_t channel )
{
/*  set the ADC reference, set for 8-bit results,
    set the desired channel number, enable the ADC,
    and start the conversion */

ADMUX  = _BV(ADLAR) | channel;  // use external reference
ADCSRA = _BV(ADEN) | _BV(ADSC);

while ( ADCSRA & _BV(ADSC) );   // busy wait for ADC

return ADCH;                    // return 8 bit result
}
```

Note that the port number (`channel`)is used directly as the MUX selection bit field (`MUX2 ...0`).
Generally, it is better to enable interrupts, and have the processor sleep (in ADC Noise Reduction, or Idle mode) while the ADC is operating.

If one of the internal references (the internal 1.1V or internal 2.56V references, or if `AVCC` is used as a reference, the `AREF` pin should be connected to a grounded capacitor. This capacitor acts as a filter for the reference voltage, and reduces potential switching transients on the reference voltage. A normal decoupling capacitor (0.01 to 0.1 $\mu$F is typical.)
The internal reference voltages are "nominal" values, and have a weak dependence on power supply voltage. If absolute accuracy is required, a calibrated external reference is necessary. The reference voltage can be measured at the `VREF` pin.

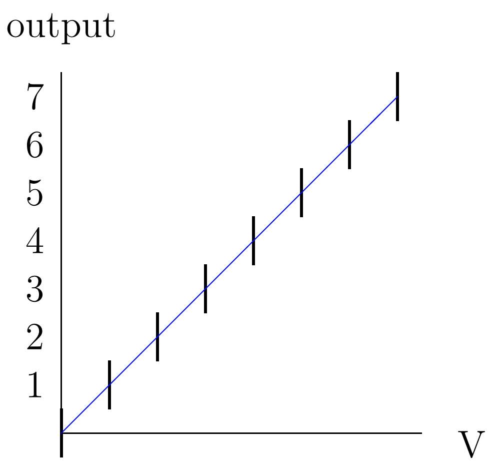
If 10x or 200x gain is selected (available in differential mode only) then only the 2.56V internal reference should be used. Of course, using an external reference is more common in this case.
At higher gain, the ADC accuracy decreases; it is 7 bits at 200x gain, and 8 bits for 10x gain.

The ADC was designed for use with signals with an output impedance of less than 10K ohms; the input to the ADC is a "sample and hold" capacitor, and high impedance sources cannot supply sufficient current to charge this capacitor. High impedance sources require an external buffer/driver (e.g., a voltage follower.)

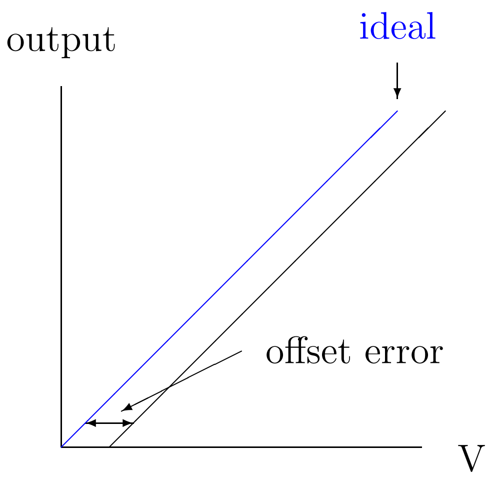
Table 25.8 in the documentation details many of the ADC characteristics.

## Types of errors in AD conversion

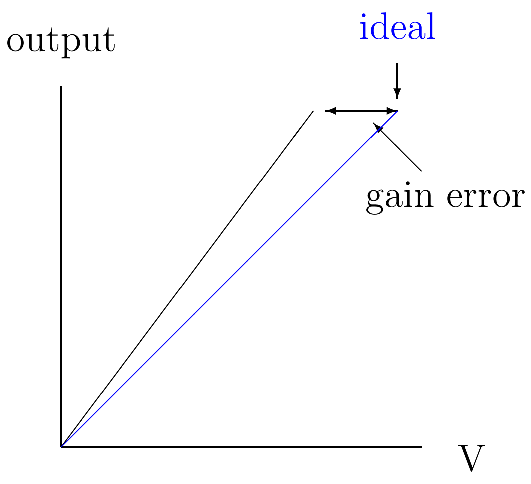
Ideally, an ADC produces exactly the value of the voltage, within the quantization error for the point.



Possible errors are:



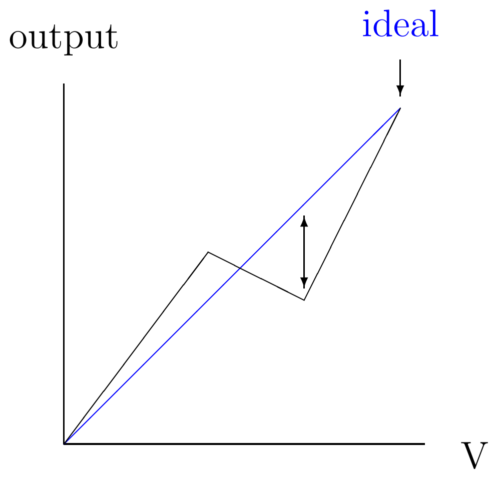
Offset error



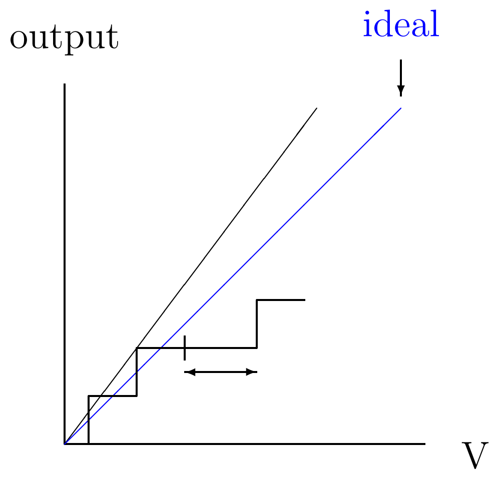
Gain error

Offset error is the error in the first transition from the ideal; gain error is the error in the last transition, after accounting for offset error.

Possibly the most troubling errors are the non-linearity errors, both the integral and differential non-linearity errors.



Integral non-linearity



Differential non-linearity

These errors are problematic because they can cause higher voltage inputs to produce a lower number than a lower voltage.
Also, it is possible that there will be output codes that are never assigned to an input (missing codes).

Often an ADC is specified as "monotonic" — this means that the digitized values always either remain the same or increase for all inputs.
They can also be specified as "no missing codes" — this means that there is some value which digitizes to each possible output value.

## Conserving power — e.g. battery operation

There are several ways to reduce power consumption in the Atmel processors:

- Reduce the operating voltage
- Reduce the clock speed
- Turn off unused logic blocks
- Put the processor in a sleep mode

Figures 29.2 and 29.3 (p. 345 of the ATmega1284p datasheet) shows the relationships among the operating frequency, voltage and current. Recall that power increases as the *square* of the current.
Often, the operating voltage is determined by other factors; e.g., a requirement to interface with other devices.

Reducing the frequency also reduces the computational performance, so this may not be feasible in some applications. In others, it may be feasible to dynamically switch the clock frequency using the clock prescale register (`CLKPR`). It allows clock division by factors of 1 to 128, in powers of 2.
In assembler, the high order bit of `CLKPR` must be set to 1, and all others to 0; then `CLKPR` must be written with the scale factor within 4 cycles, otherwise `CLKPR` is unchanged.
In the AVR libc, the function
```
clock_prescale_set(clock_div_1)
```
sets the clock division factor. (Default is `clock_div_8`.)

## The power reduction register — turning off unused logic

The power reduction register (`PRR`) provides a way to stop the internal clocks for various peripheral devices.
In the ATmega1284p, there are 7 internal clocks which can be controlled by setting bits in this register.

In AVR libc, control of the power reduction register is provided by
`<avr/power.h>`
and include functions like
`power_all_disable()`, `power_all_enable()`
`power_usart0_disable()`, `power_usart0_enable()`
`power_twi_disable()` , `power_twi_enable()`
`power_timer2_disable()`, `power_timer2_enable()`
`power_timer1_disable()`, `power_timer1_enable()`
`power_timer0_disable()`, `power_timer0_enable()`
`power_spi_disable()`, `power_spi_enable()`
`power_adc_enable()`, `power_adc_disable()`

Setting the control bits in the register freezes the operation of the peripheral devices affected. When unset, operation continues normally.

The `PRR` can also be manipulated directly by setting its bits directly, but this leads to less portable code.
(In this case, however, the size of the code may be reduced if several devices are enabled or disabled simultaneously.)

Table 29.1 and 29.2 (page 350 0f the ATmega1284p manual) shows the current consumption for the internal I/O modules.

## Sleep modes

Perhaps the most useful way to conserve power is to place the processor in one of the sleep modes when it is not necessary to have the processor run.
This is done by setting the appropriate bits in the sleep mode control register (SMCR).
An interrupt can bring the processor out of a sleep mode in 6 cycles, if the internal oscillator is used.
There are five (or 6) sleep modes, and the mode using the fewest required internal clocks should be chosen.

The "deepest" sleep mode is *power down*, in which all internally generated clocks are disabled, so only external, asynchronous events can interrupt the processor (e.g., pin change interrupts). In this mode, the processor consumes only microamps of current (decreased by a factor of 1000.

Another power save mode, the *power save mode* is similar, but if counter/timer2 is enabled, it keeps running.

The modes *standby* and *extended standby* are similar to power down and power save modes respectively, but they also keep the external oscillator running. These modes are used with an external crystal.

The *idle* mode keeps all the internal clocks running except the CPU clock, and the clock used to write the flash memory. Typically, the idle mode uses about 1/3 the current of the active mode.

The final mode, *ADC reduction*, is used to reduce the noise when an analog to digital conversion is being performed, in order to reduce the noise in the conversion.
We will discuss this in more detail later, but for now note that the noise fed back into the power circuitry can briefly alter the voltage supplied to the processor, and consequently affect the reference voltage for the ADC.
This mode stops the I/O, CPU, and flash memory clocks, but permits the timers and ADC to continue.

In AVR libc, the sleep functions are supported by
`<avr/sleep.h>`
with the functions `set_sleep_mode()` and `sleep_mode()`.
These functions should be used carefully — read the AVR libc documentation!

Although less preferable (and less portable) the sleep modes can also be managed by writing the appropriate bit patterns to the register `SCMR`.

The following table shows the arguments that are defined for the function `set_sleep_mode()` for the ATMega1284p:

```
SLEEP_MODE_IDLE
SLEEP_MODE_ADC
SLEEP_MODE_PWR_DOWN
SLEEP_MODE_PWR_SAVE
SLEEP_MODE_STANDBY
SLEEP_MODE_EXT_STANDBY
```

Table 10.1 in the datasheet shows the clock domains that are active in each of the sleep modes.

## Resetting the processor

There are five ways of resetting the processor:

1. Power on reset — the processor is reset when the power is initially turned on
2. The reset pin — one external pin can be programmed to reset the processor when the pin is set low, and released. (The ATmega1284p has a dedicated reset pin – pin 9.)
3. Brown-out reset — when the voltage drops below some threshold, *and* brownout detection (BOD) is enabled, the processor resets. (We have disabled BOD.)
4. Watchdog timer reset — if the watchdog timer is enabled and the watchdog period expires, then the processor resets. (We have disabled the watchdog timer.)
5. JTAG reset — more on this later (time permitting).

## The brownout detector (BOD)

The brownout detector is enabled using the three `BODLEVEL` fuses. Three different voltages can be selected for BOD; 1.8V, 2.7V, and 4.3V.

When the supply voltage to the processor drops below the selected voltage, the brown-out reset is activated.

When the voltage rises above the BOD voltage (there is a hysteresis of about 50mV), the timeout delay timer starts, and the processor restarts at the end of this timeout period. The timeout period depends on the clock source selected by the fuses (`CLKSEL0 ...3`) and the startup time (`SUT0, SUT1`) fuses.

Figure 11.1, p. 45 of the ATmega1284p manual shows a diagram of the reset logic.
Figure 11.5, p.49 a timing diagram for BOD.

The BOD circuitry allows for reasonable recovery after a temporary power failure.

The register `MCUSR` has a bit which is set after a BOD reset, and which can be read to determine the reason for a reset. We will look at this register later.

## The watchdog timer (WDT)

A watchdog timer is a standard technique for forcing a system to recover from some uncontrolled state.
You may have well used the same idea to bound the wait time for an event in some code you have written.
Basically, the system has a register (or variable) — the watchdog register— which is periodically reset by the system if it is functioning normally.
If the system is functioning abnormally, the watchdog timer interrupts the process when the counter reaches some maximum value before being reset.
Normally, the watchdog timer is an *independent* process that restarts whenever the watchdog register is reset.

In the ATmega1284p, the watchdog timer is clocked from a separate on-chip oscillator with a frequency of 128 KHz.
The WDT can be set to reset the processor, interrupt the processor, or both.
The timer has a programmable prescaler, which allows the timeout period to be set from 16 ms. to 8 s, in steps doubling in size.

The `WDTON` fuse can be programmed so that the WDT is always on, and cannot be disabled by software. (Default is that `WDTON` is unprogrammed.)
In AVR libc, WDT control is provided by `#include <avr/wdt.h>`

## Controlling the WDT

In normal operation, the WDT is reset using the watchdog reset (`WDR`) instruction. This must be done before the watchdog period in order to prevent the watchdog reset or interrupt.

In AVR libc, this is done using the function `wdt_reset()`

The WDT is controlled by the register `WDTCSR`, and individual bits in this register determine the mode of operation (interrupt or reset), and the value of the WDT prescaler.
See p.59 of the ATmega1284p documentation for a detailed description of the bits in the `WDTCSR`.

Similar to setting the clock frequency divider, there is a time limited procedure (4 cycles) for setting the WDT prescaler.
The default is the minimum time, about 15 ms.

In AVR libc, the WDT timeout is set by the function
`wdt_enable(timeout)`
where the argument `timeout` is a defined constant of the form `WDTO_15MS`.

The time following `WDTO_` is one of the allowed times for the WDT; e.g., `WDTO_120MS`, `WDTO_250MS`, `WDTO_1S`.
The WDT can be disabled with `wdt_disable()`

The ATMEL documentation recommends disabling the WDT in the initialization routine, even if it is not used. There is a possibility that the WDT could be accidentally enabled, (say, by a brown-out condition or — never for us, of course — a programming error like a stray pointer) and then cause spurious resets from that point on.

This is done by clearing the watchdog reset flag (`WDRF`) in the register `MCUSR`, and the watchdog enable (`WDE`) bit in the `WDTCSR`.

In AVR libc, the following can be used:

```
MCUSR = 0;
wdt_disable();
```

Note that the WDT need not be used as a watchdog timer; for example, in interrupt mode, it can be used as a "slow" timer to wake up the processor from a sleep mode.

In interrupt and system reset mode, the interrupt is executed, but the interrupt enable flag must be reset after the interrupt (it is cleared by hardware when the corresponding interrupt vector is executed), or the next watchdog timeout will cause a reset.
This allows the watchdog timer to be used for a dual purpose — timing and as a watchdog timer.

As a timer, the 128KHz watchdog timer is not as accurate as the other timers, however.

## The `MCUSR` register

This register provides information on which reset source caused the processor to reset. There is an individual bit set whenever any of the five possible resets occurs.
All the bits, except power-on reset, are reset when a power-on reset occurs.
The register can be read to determine the source of a reset.
This is useful because the programmer may wish to do different things depending on the type of reset.
(For example, the programmer may want to log power-on resets in eeprom memory.)

In order to turn off the watchdog timer, the watchdog timer flag in `MCUSR` must be reset. This effectively overrides the WDE (watchdog enable) bit in `WDTCSR`.
The purpose of this is to ensure multiple resets during a failure condition, and a safe reset afterward.

## Timing accuracy — calibrating the clock

Atmel processors can have four different types of timing sources:

1. An internal oscillator (default)
2. An external crystal
3. An external ceramic resonator
4. An external oscillator

The different clocking methods have different accuracies, and different applications.

The external oscillator is used for testing, and when several processors must be synchronized.

The internal oscillator (which we have been using) has an initial accuracy of about 10%, but can be calibrated using the oscillator calibration register (`OSCCAL`).
The frequency of the internal oscillator depends on the supply voltage and operating temperature. (See Figures 29.87 and 29.88 on pp. 390–391 for plots of frequency vs. operating voltage and temperature, respectively.

In the newer Atmel devices, the **`OSCCAL`** values are two overlapping ranges, depending on the value of the high order bit.
(See Figure 29.89 on p. 391 for a plot of frequency vs. `OSCCAL` value.)

Using `OSCCAL`, it is possible to calibrate the internal oscillator to within approximately 1%, for a given voltage and operating temperature.

Ceramic resonators are cheap timing elements, with accuracy of about 0.5%. They provide reasonable accuracy for low cost, and are relatively easy to design with.
The processor fuses can be set to match the characteristic of the resonator; see Table 9.4 on page 33.
A typical data sheet for a ceramic resonator is available in the “links” page for the course.

External crystals are more expensive (still quite cheap) but much more accurate, with accuracies potentially measured in parts per million. A data sheet for a typical crystal is also available in the “links” page.
The latest Atmel processors support two modes of operation for a crystal; “full swing mode”, and “low power oscillator” mode.
See pages 31 to 34 of the ATmega1284p data sheet for details.

Both ceramic resonators and crystals require that the processor drive an off-circuit device, resulting in more power consumption.
Also, both devices require some time to stabilize, so the start-up time is relatively long.

## Calibrating the on-chip oscillator

ATMEL has several application notes on ways to automatically calibrate the internal oscillator using the `OSCCAL` register.

Application note AVR053 has code to calibrate the oscillator from the STK500.

Application note AVR054 describes how the oscillator can be calibrated via the UART.

Both those techniques depend on the accuracy of an external timing source, and are applied for a fixed operating voltage and temperature.

Application note AVR055 describes how the oscillator can be calibrated using a low frequency crystal connected to one of the timer inputs.
In the ATmega1284p, timer 2 can drive a 32.768 KHz crystal (commonly used for quartz clocks) in order to implement a real-time clock. This application note describes how to use this crystal to automatically calibrate the internal oscillator. This is useful because the processor can recalibrate itself under different operating conditions (operating voltage and temperature.)
32.768 KHz crystals are cheap and accurate.

## EEPROM memory

In addition to the program memory (128K bytes of flash memory) and data memory (8K bytes volatile read/write memory), the ATmega1284p has 4K bytes of EEPROM memory.
Like flash memory, EEPROM memory is non-volatile (it retains its value after power is cycled). However, it can be reprogrammed at least ten times as often as flash memory (100,000 times vs. 10,000 times).
All three memory spaces are linear, regular, and independent.
Typically, the EEPROM memory is used for several purposes:

- data logging — recording values that should survive power cycling
- data that changes infrequently — calibration data, for example, that may be updated periodically
- configurational information — user dependent data that may need occasional updating

The EEPROM memory is addressed using the EEPROM address registers in I/O space (registers EEARH and EEARL).
The register EEPROM data register (EEDR) contains the data to be read or written.
There is also an EEPROM control register, EECR.
Reading from the EEPROM requires 4 cycles.
Writing the EEPROM typically requires approximately 3.4 ms.

The EEPROM control register, EECR, has 6 bits:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| - | - | EEMP1 | EEMP0 | EERIE | EEMPE | EEPE | EERE |

Bits 5, 4 are the EEPROM mode bits, (EEPM1, EEPM0) which determine whether the operation is an erase and write (00), erase only (01), or write only (10).

Bit 3 (EERIE) is the EEPROM interrupt enable (it works like other interrupt enable bits).

Bit 2 (EEMPE) is the “master programming enable” bit. It must be set to 1 to allow the EEPROM to be written. Moreover, EEPE (bit 1) must be set to 1 within 4 cycles to cause the writing of the value in EEDR to be written into the address selected by EEARH and EEARL.
EEMPE is automatically set to 0 after 4 cycles.

Bit 1 (EEPE) is the write strobe for the EEPROM. It usually follows the setting of EEMPE, after the address and data bits are correctly set up.
EEPE is set to 0 when the EEPROM write has completed. (This requires many machine cycles, and other instructions may be executed in the meantime.)

Bit 0 (EERE) is the read strobe. A read cannot occur if a write is in progress, so EEPE should be polled before attempting a read. In fact, the address cannot be changed until a write has been completed, either.

Interrupts can interfere with the writing of the EEPROM, so usually interrupts other than EEPROM interrupts are disabled.
The general procedure for reading from the EEPROM is:

1. Test EEPE. If 0, proceed.
2. Set up read address.
3. Set EERE to 1. When this operation is completed (one cycle, EEDR has the data after the processor is halted for 4 cycles)

For writing the EEPROM, (assuming that there are no writes to flash memory in progress) the operations are:

1. Test EEPE. If 0, proceed.
2. Set up WRITE address in EEARH and EEARL.
3. Set up data to be written in EEDR.
4. Set EEMPE to 1.
5. Set EEPE to 1 within 4 cycles. This means that an interrupt here will cause this operation to fail, so interrupts should be disabled first.

EEPROM and flash memory cannot be written at the same time, so if this is a possibility, read the manual!

## EEPROM programming in C

The AVR libc library has functions to handle writing to EEPROM in `<avr/eeprom.h>`.

Since EEPROM functions (especially write, or read after write) can require many cycles, the function `eeprom_is_ready()` can be used to poll the state of the EEPROM.

There are functions to safely read and write EEPROM memory, in both byte and word forms, and extended versions of the functions for block reads and writes.
The typical form is `type eeprom_read_xxxx(*arg)`

Explicitly, the following functions are available:

```
uint8_t  eeprom_read_byte (const uint8_t *addr)
uint16_t eeprom_read_word (const uint16_t *addr)
void     eeprom_read_block (void *pointer_ram,
                     const void *pointer_eeprom, size_t n)
void     eeprom_write_byte (uint8_t *addr, uint8_t value)
void     eeprom_write_word (uint16_t *addr, uint16_t value)
void     eeprom_write_block (const void *pointer_ram,
                      void *pointer_eeprom, size_t n)
```

These functions are non-re-entrant, so interrupts should be disabled before they are used. (They modify IO registers.)

Another useful function is `eeprom_busy_wait()` which does what the name suggests.

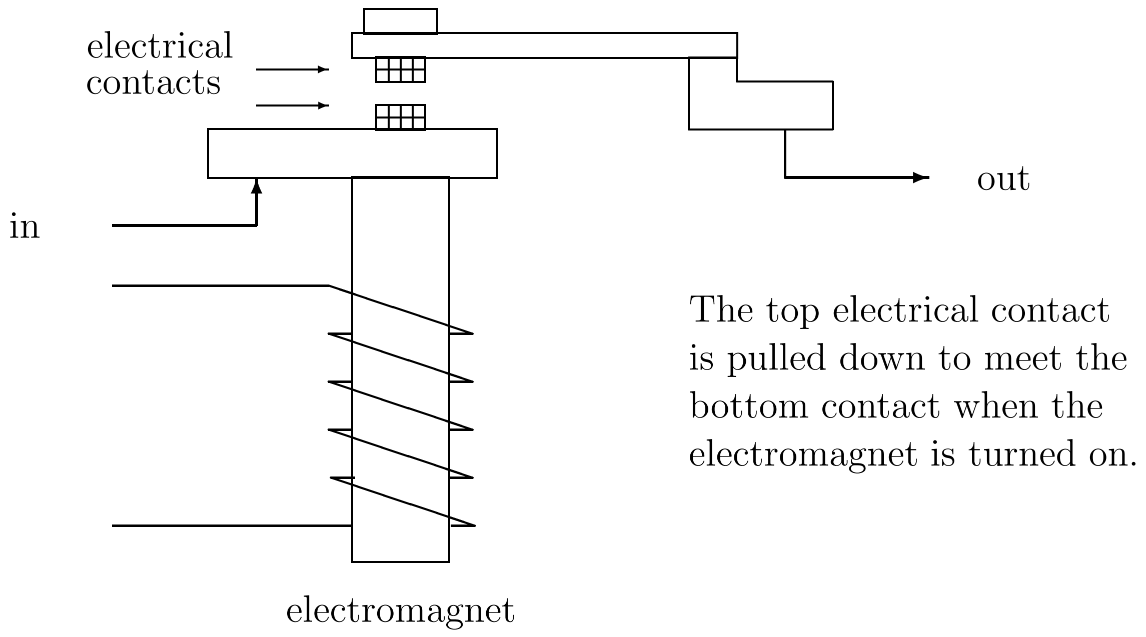
## Switches

One problem with the microprocessor ports is their limited current carrying capacity.

One way to control larger currents is to have the microprocessor control a switch — in this case, the switch should be capable of being activated by the current or voltage provided by the port.

So, what can we use as as switches? One of the earliest “electronic” switches was the electromagnetic relay:

Electromagnetic Relay


electrical contacts
out
in
The top electrical contact is pulled down to meet the bottom contact when the electromagnet is turned on.
electromagnet

This device was actually used to build computing devices, and was used extensively for telephone switching.

It is still useful for switching high voltages and currents.

It’s main drawbacks are its size, speed, and power consumption.

Actually, there are relays that can switch at fairly high speeds (for mechanical devices) but those devices cannot handle high currents, since the moving parts have to be very small to be fast.

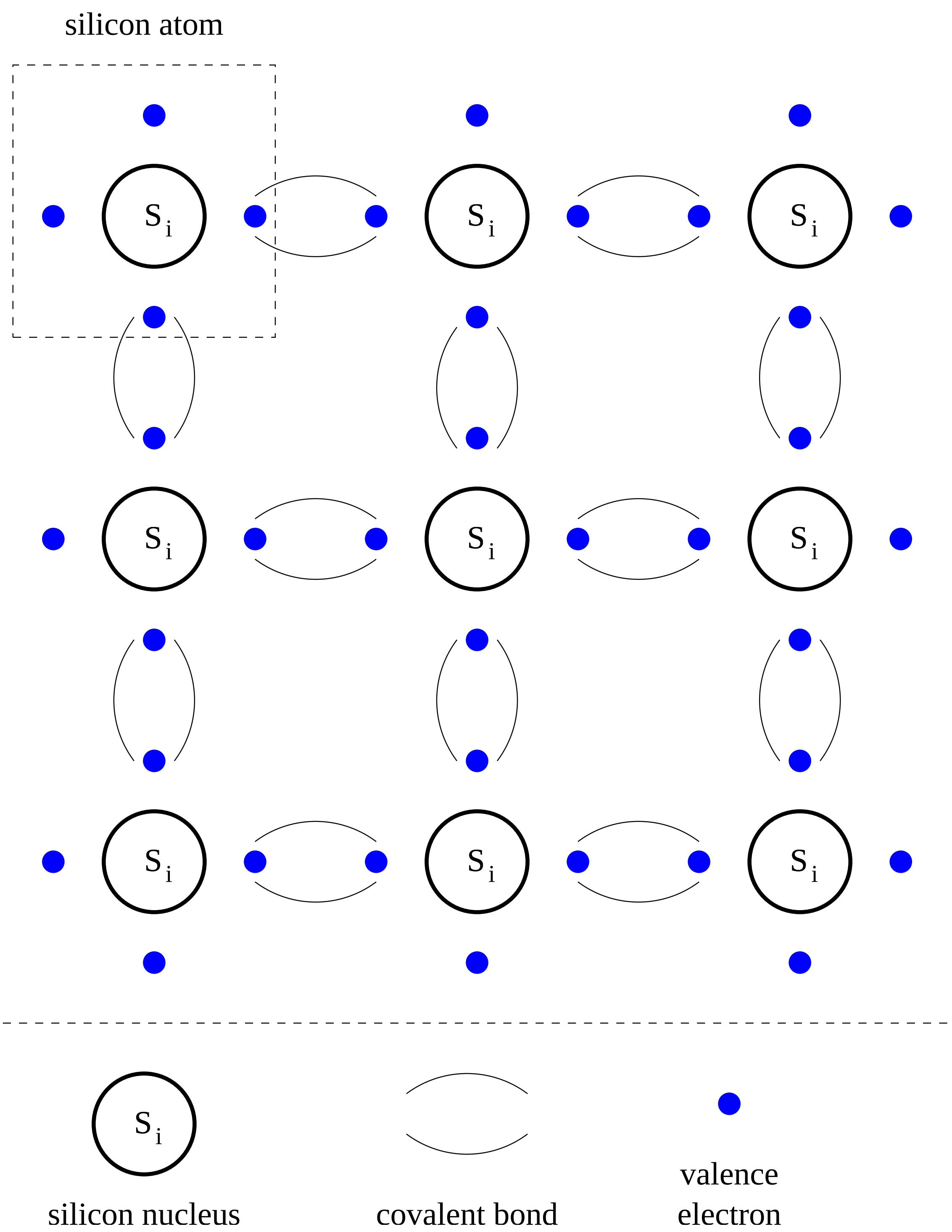
A purely electronic (solid state) switch would be desirable; potential switching speeds could approach the speed of the microprocessor. (After all, the processor itself is basically a complex set of switches.)

Fortunately, there are several types of solid state switches, and we will look briefly at how two of those work.

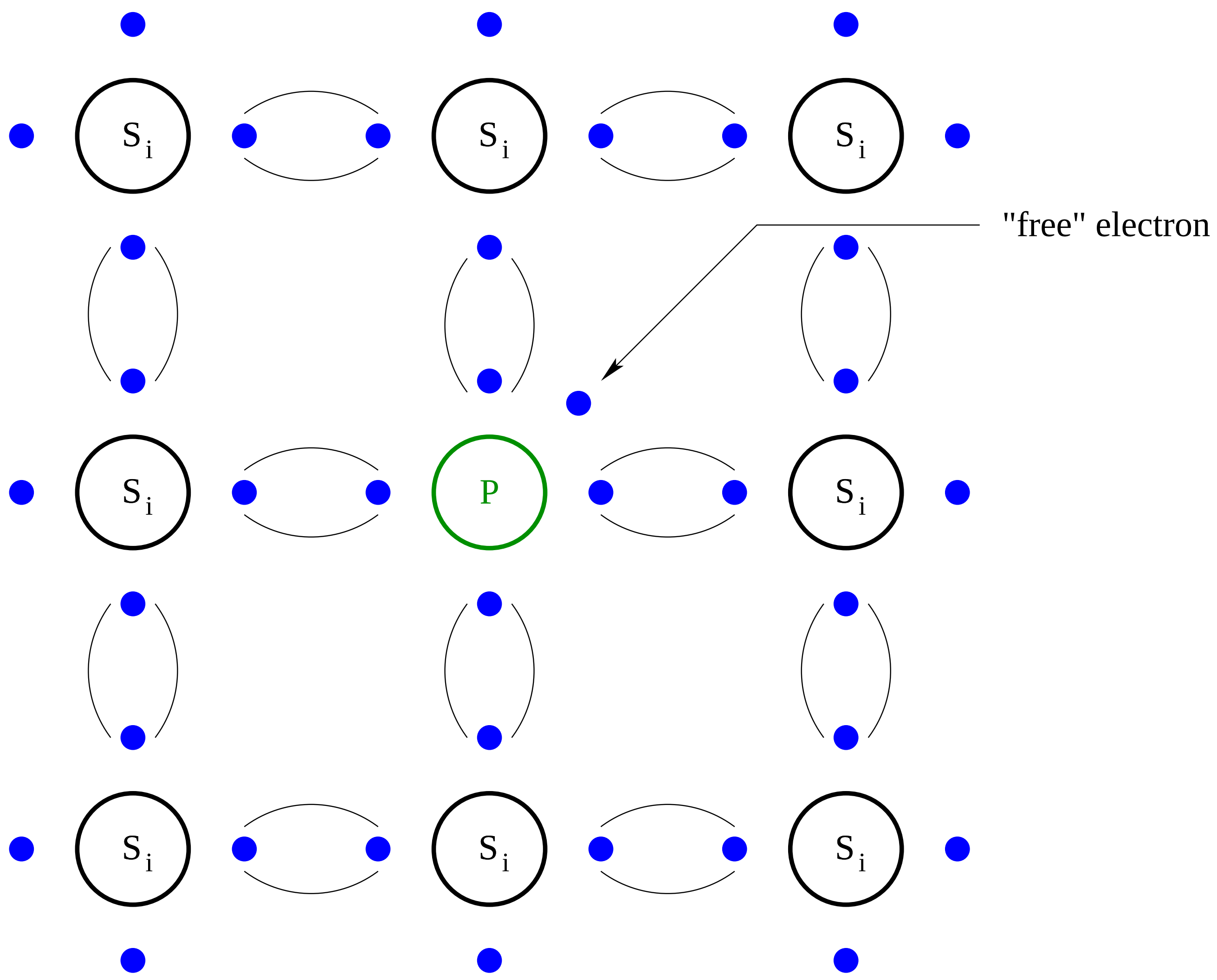
The following explanations and descriptions of solid state devices are very sketchy, and are just to give an idea of how these devices operate.

## The “solid state” world — semiconductors

### A semiconductor crystal lattice — “pure” silicon

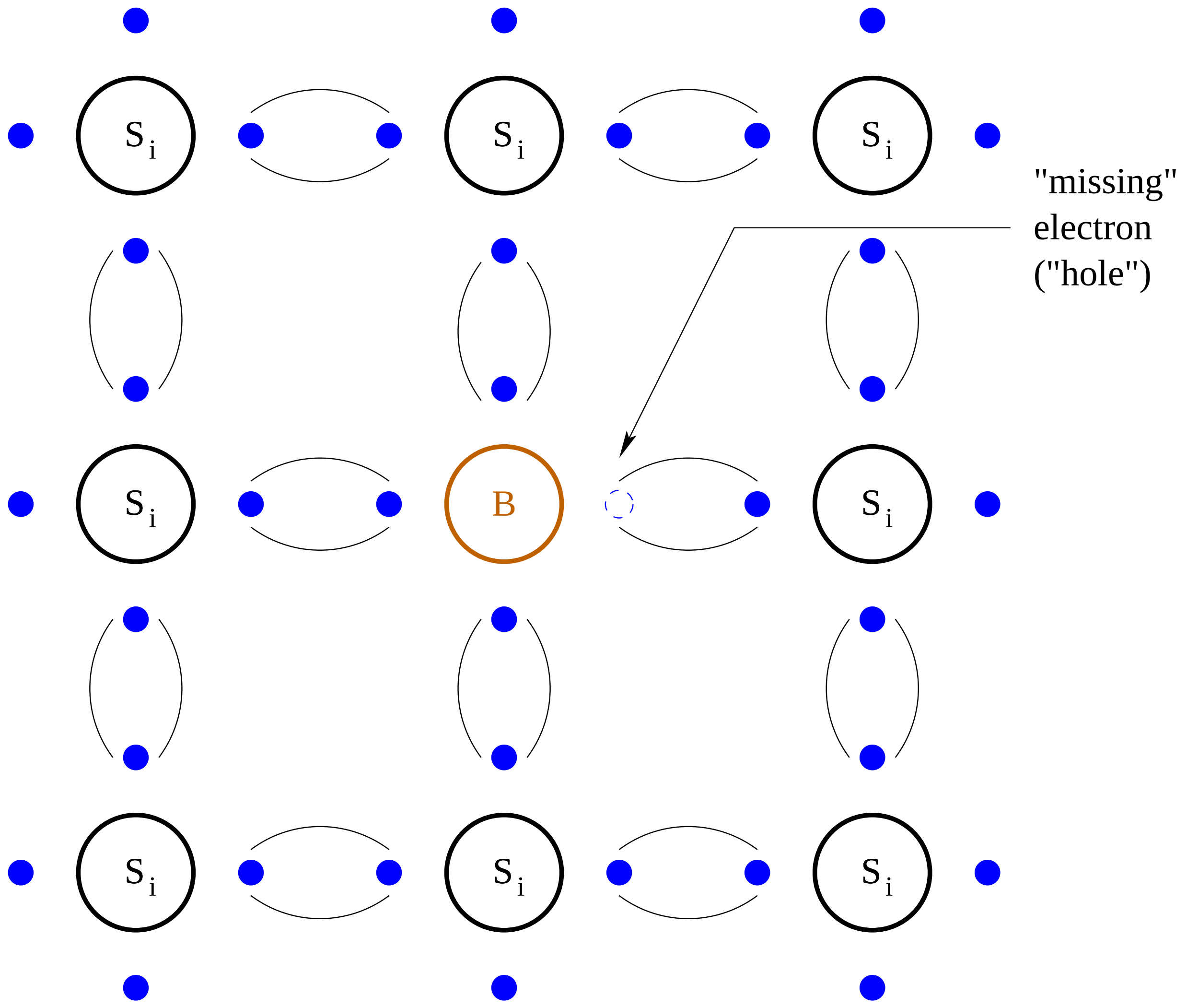


N–doped (*i.e.* 5 electron material) silicon crystal



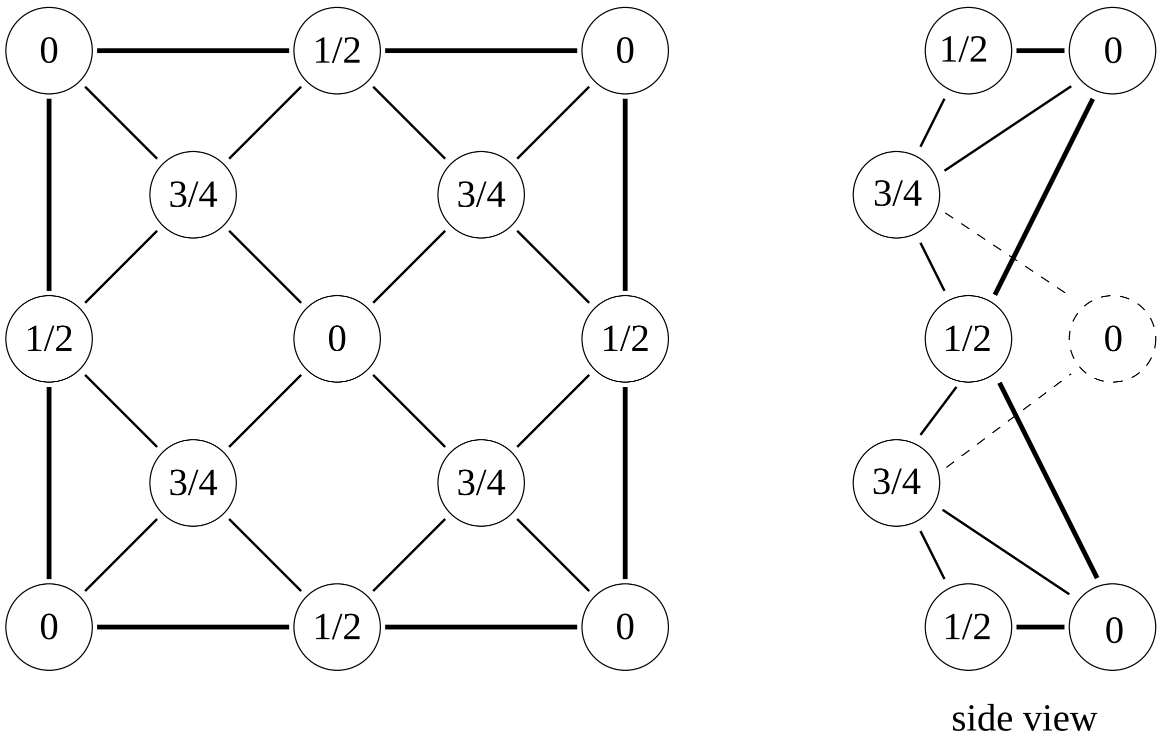
Phosphorous doped — the “extra” electron is relatively free to move about in the crystal.

P–doped (*i.e.* 3 electron material) silicon crystal



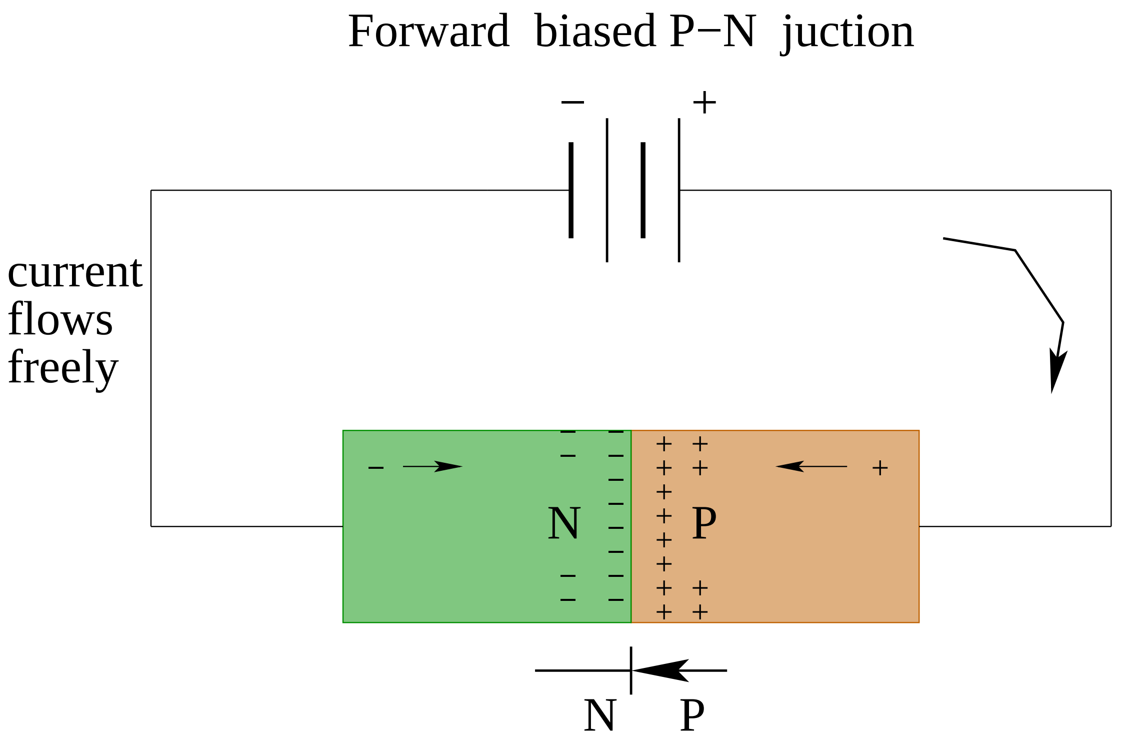
Boron doped — the "hole" is relatively free to move about in the crystal.

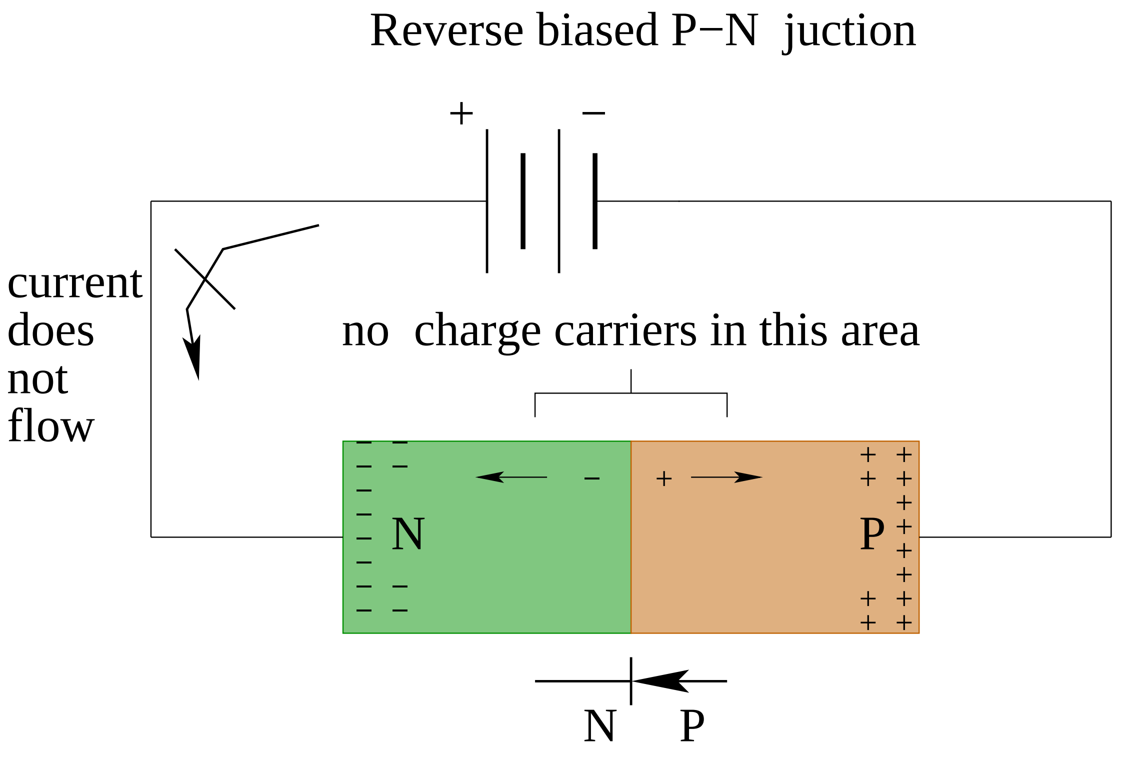
Actually, silicon has a diamond crystal structure, which has the following projection on a cube face:



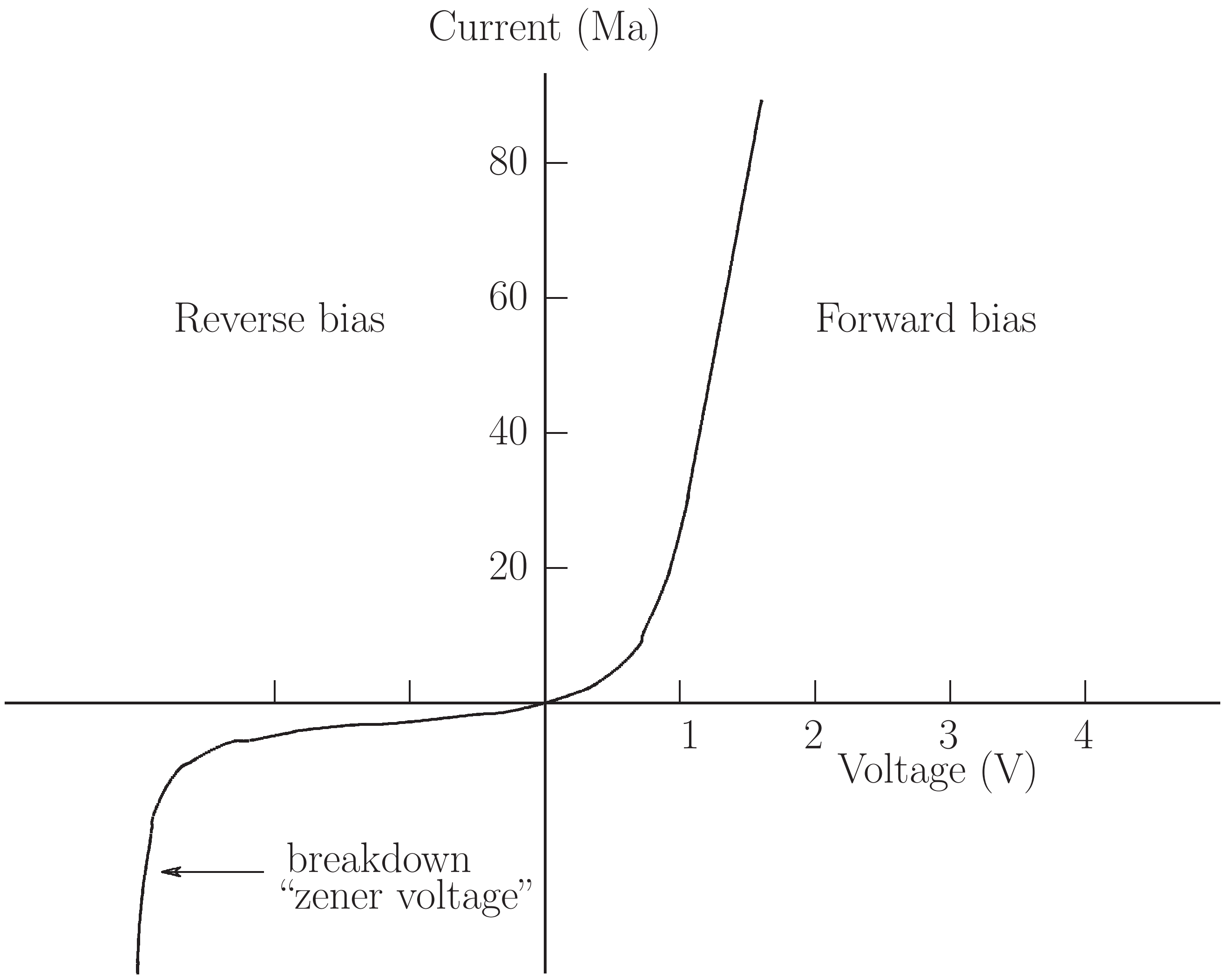
The numbers in the nuclei denote height above the base in units of a cube edge.

The P-N junction as a diode:

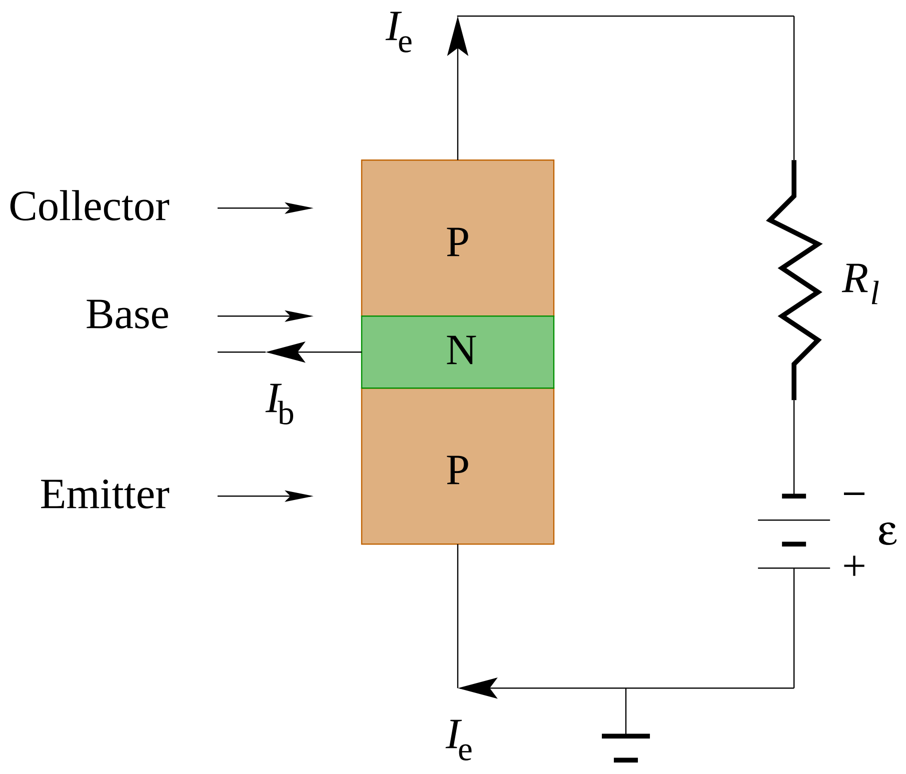




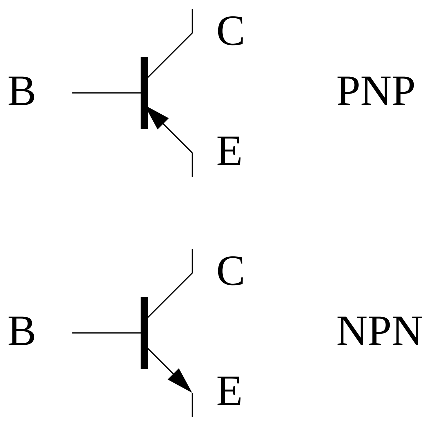
The electrical characteristics of a semiconductor diode:



## The bipolar PNP transistor



Circuit symbols for transistors



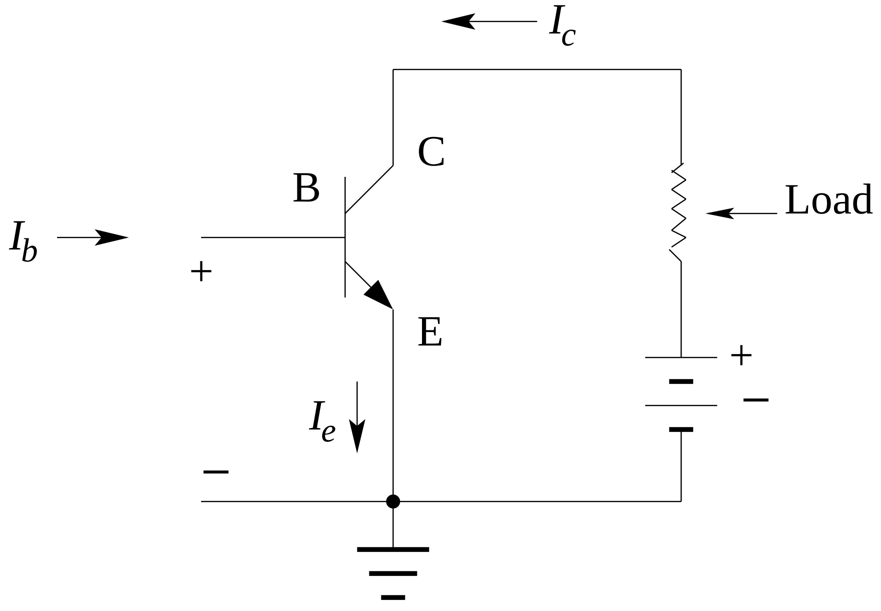
In the PNP transistor configured as shown, with *no* connection to the base, the emitter–base junction is forward biased, so holes flow from the emitter to the base. However, since the base is thin, *most* of the holes continue on to the collector, where they are "attracted" by the negative terminal of the battery. If there is no base current, electrons are "lost" from the base by 2 processes:

1. electrons flow across the emitter–base junction to the + terminal of the battery.
2. electrons combine with holes and disappear.

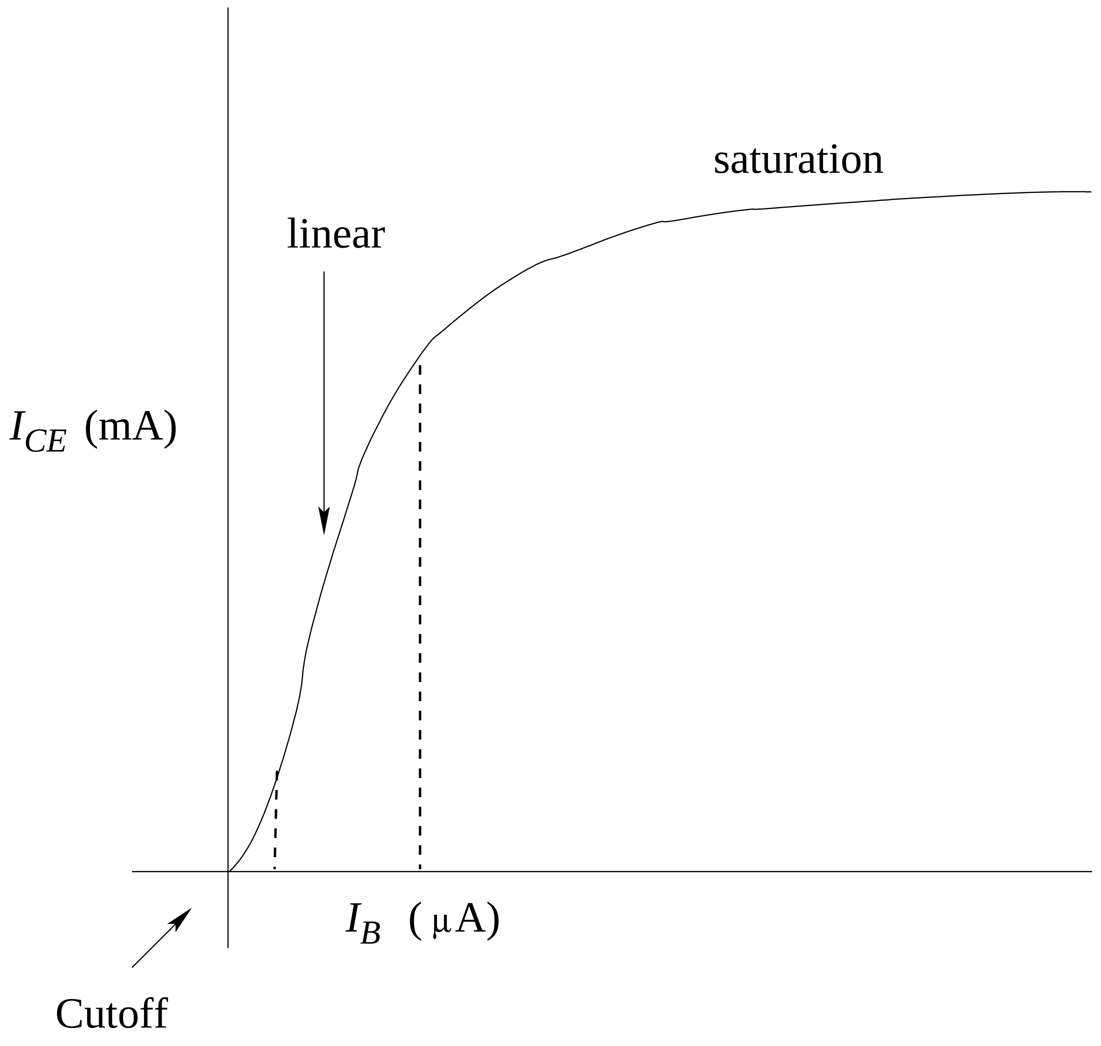
As a result of these two processes, a positive charge builds up in the base which completely stops the positively charged holes from passing through the base. If we inject electrons *into* the base, however, we can cancel, to some extent, this inhibiting positive charge. A small number of electrons injected into the base allows a much larger number of holes to pass through, so this PNP transistor can act as an *amplifier* — it acts as a *current* amplifier.

A similar transistor can be constructed from an NPN "sandwich." It operates in a *complementary* way — electrons attempt to flow from the collector through the base and into the emitter. Injecting holes into the base region permits current to flow.

Typical NPN transistor configuration (the "common emitter amplifier")

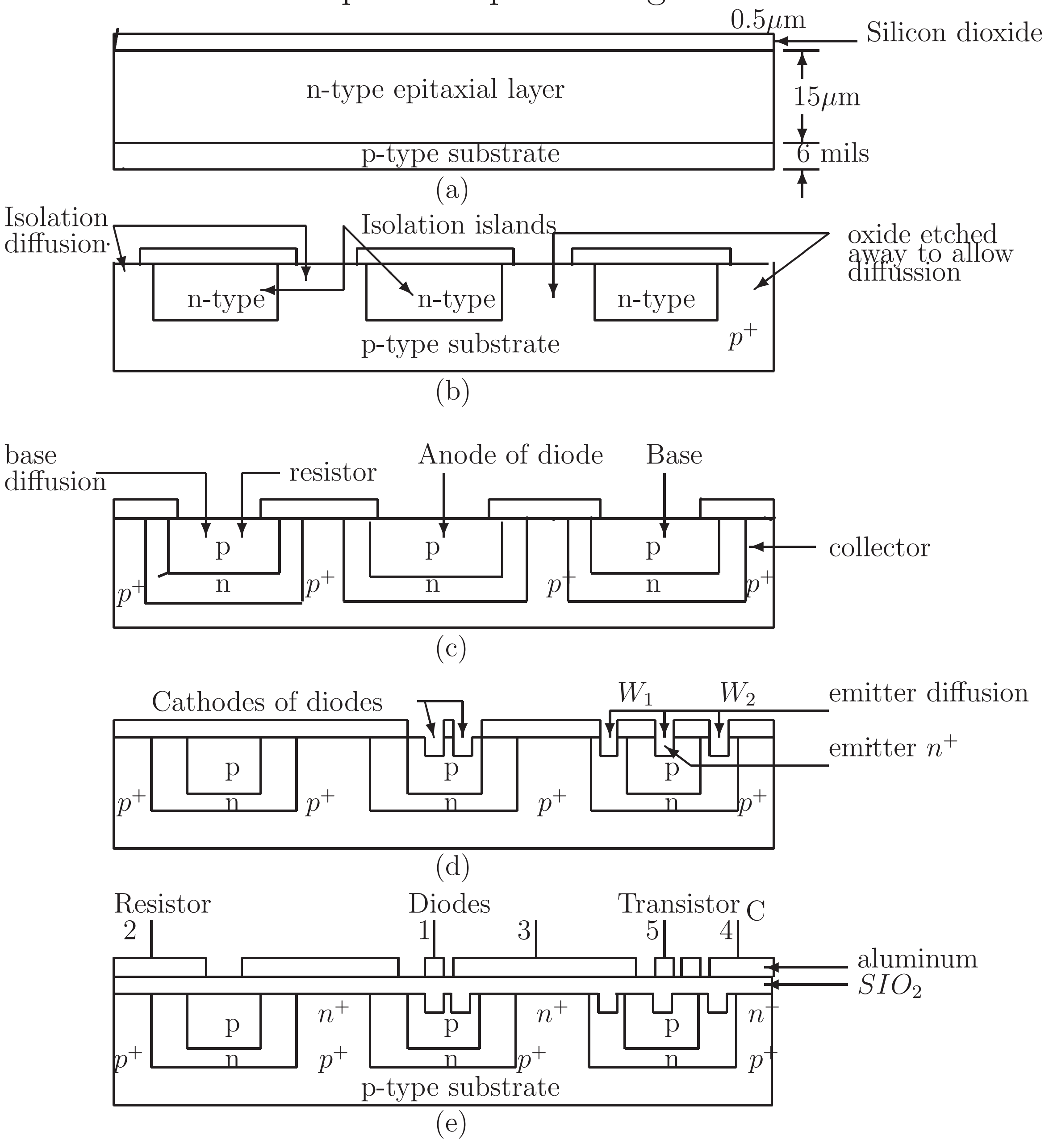


For an NPN transistor, the current gain, $\beta$, ranges from 10 – 400, typically.



$I_B$ is often measured in $\mu$amps; $I_{CE}$ may be measured in *milli*amps. Note that current flow is due to *both* electrons and *holes*. Here, electrons are the minority charge carriers, and holes are the majority charge carriers. Also, although the bipolar transistor *appears* to be symmetric, in general, the emitter and collector are *not* interchangeable, and the bipolar transistor is unidirectional, for current.

## The manufacture of planar bipolar integrated circuits

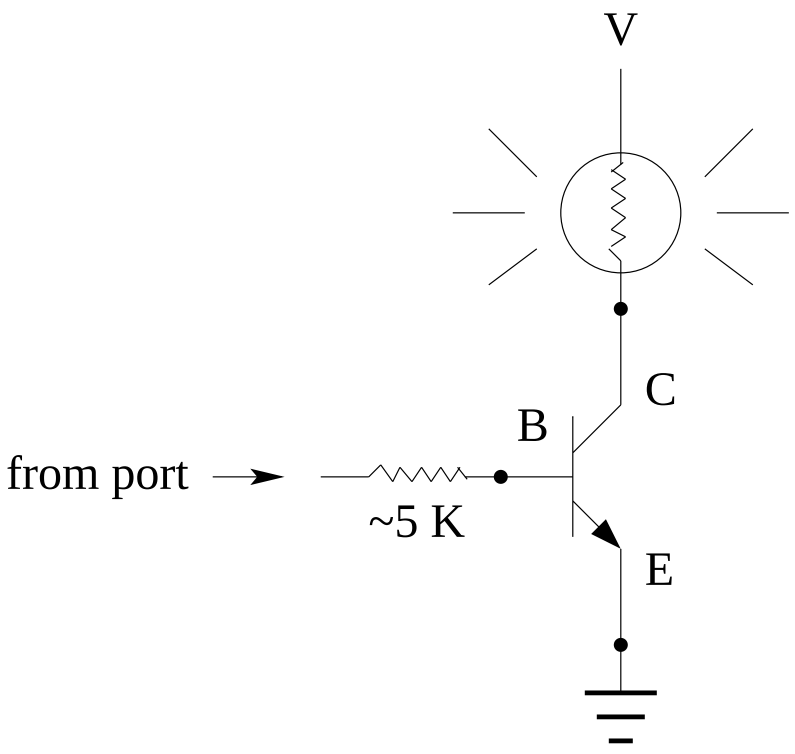


Transistor isolation is maintained by the reverse-biased P-N junction between the collector and the substrate.

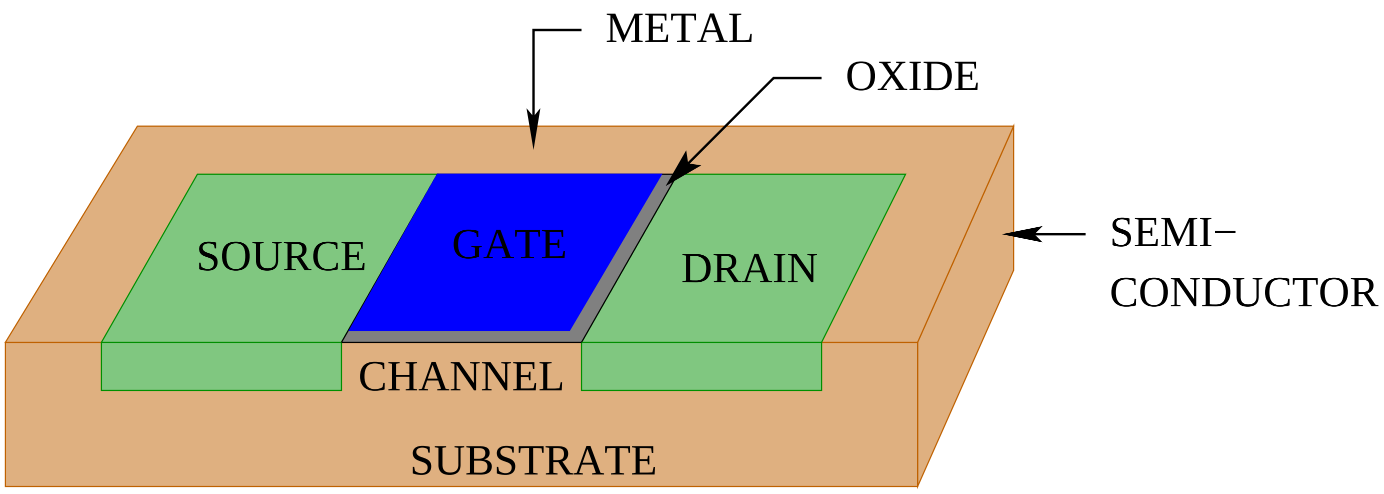
As a switch, the bipolar transistor can be thought of as a current activated device. Essentially, providing a small current to the base region allows a much larger current to flow between the collector and emitter.

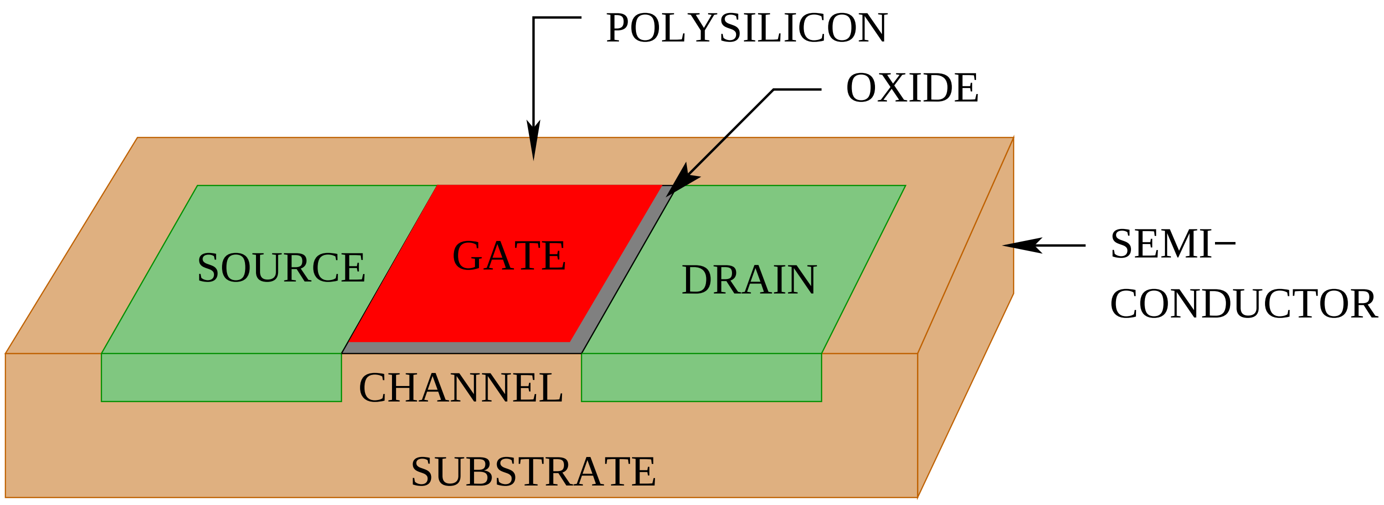
Note that in a bipolar transistor, conduction is due to both types of charge carriers — electrons and holes.

A common configuration for an NPN transistor controlling a high current device (e.g., a DC lamp) is as follows:



A second type of transistor is also commonly found: the Field-Effect Transistor, or FET. The commonest from of FET is the metal-oxide-semiconductor (MOS) FET, or MOSFET. (Actually, a more appropriate name is insulated-gate FET, or IGFET, because metal is no longer used in the gate region. The material used in the gate region is usually polycrystalline silicon, called *poly*.)
The area under the gate is called the *channel*.

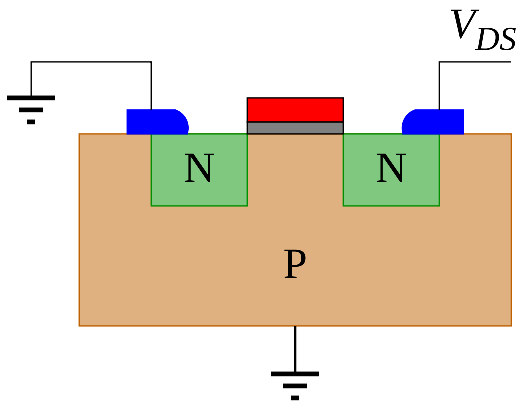




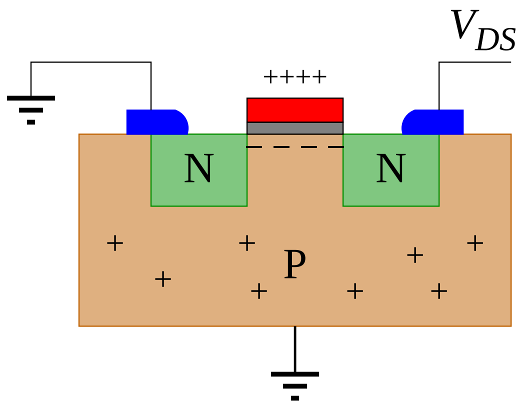
Enhancement mode MOSFET

This is an N-channel enhancement mode MOSFET.

Here a positive charge on the gate induces a negative charge in the P-doped material immediately under the gate (*i.e.* in the "channel"). When more electrons are attracted into this region than there are holes, the channel acts like an N-type material, and current can flow between the *source* and *drain*.



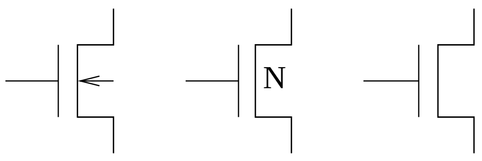
(a)

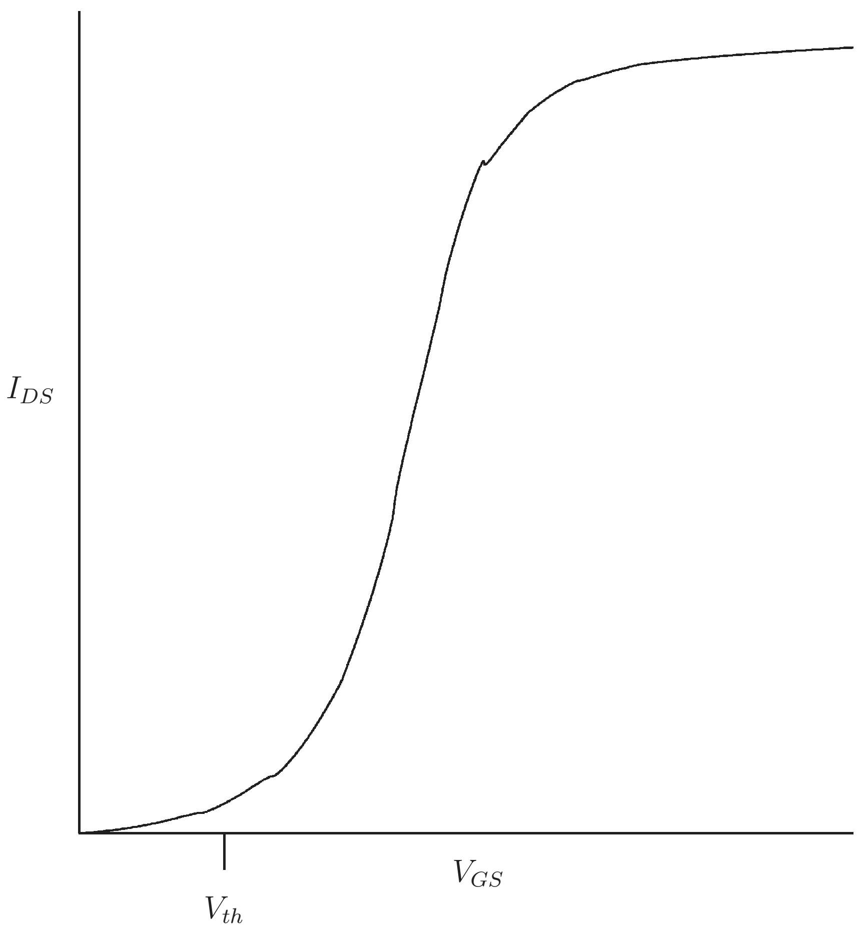


(b)

There are several "standard" circuit symbols for NMOS transistors:

NMOS circuit symbols

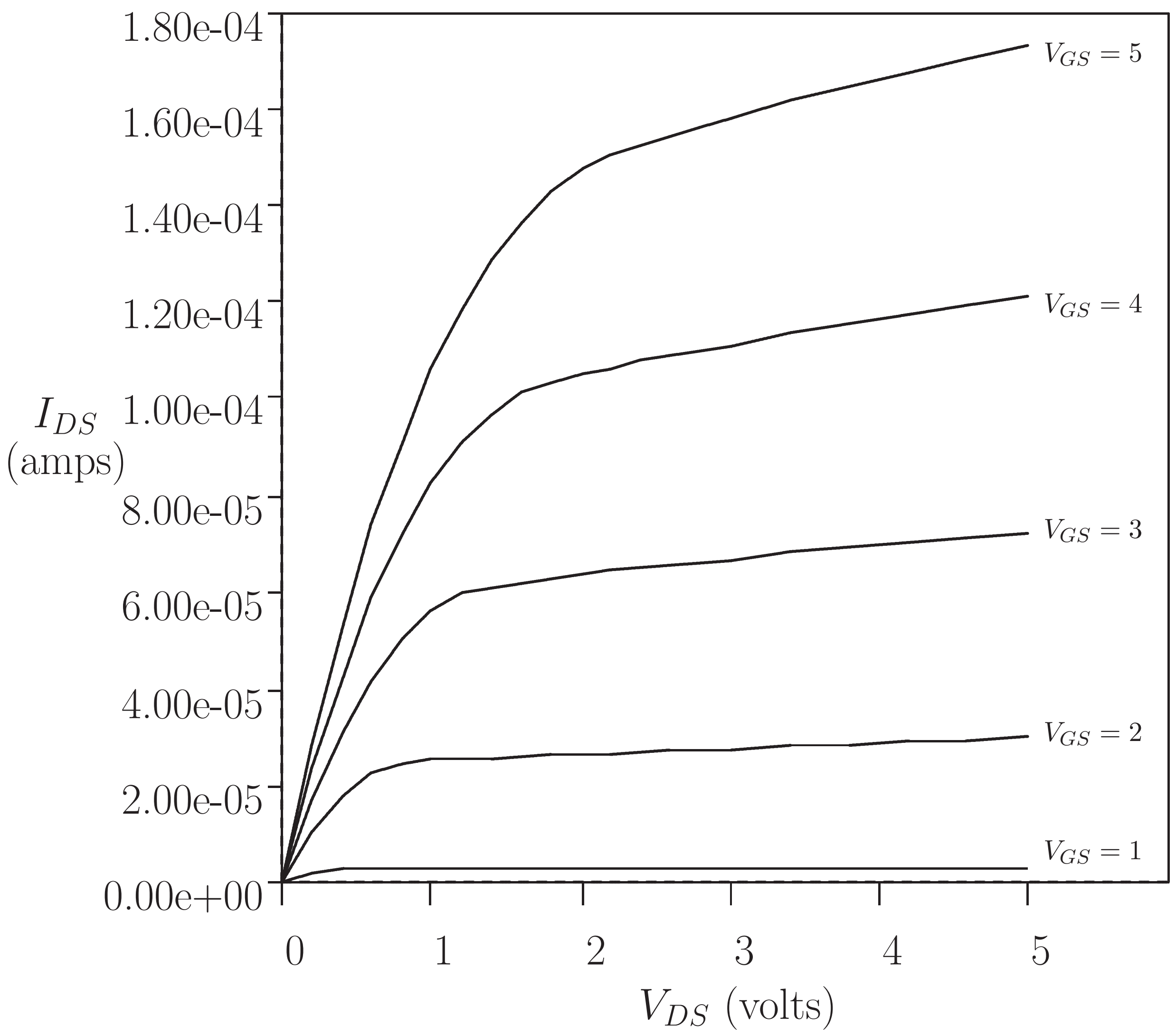




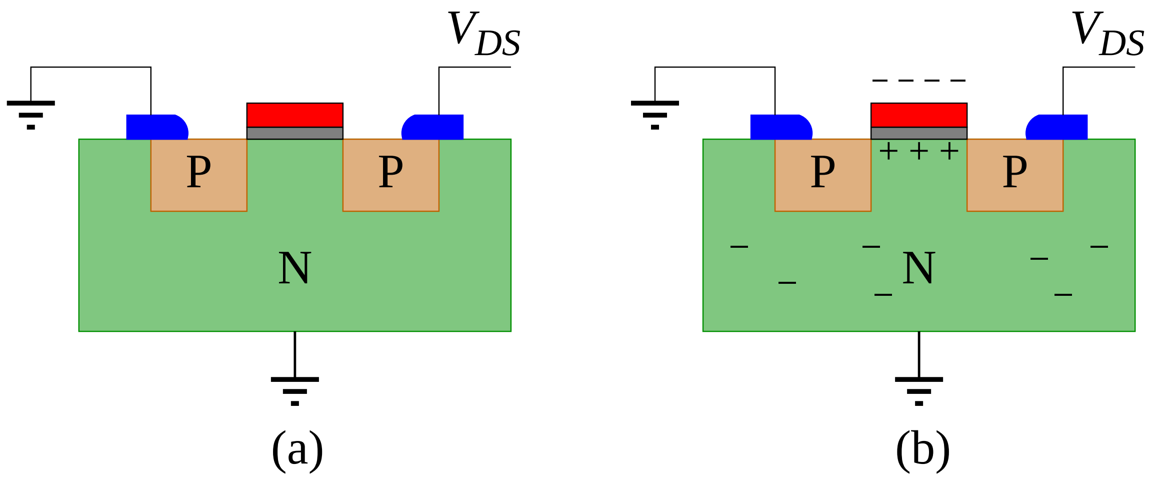
Note that there is no current flowing for $V_{GS}$ less than about 1 Volt. The value of $V_{GS}$ at which current begins to flow is called the "threshold voltage", $V_{th}$. Typically, $V_{th} \simeq 0.2\, V_{DD}$, where $V_{DD}$ is the supply voltage.

Following is a typical set of characteristic curves for a small NMOS transistor:

CMOS3 NMOS transistor DC characteristics

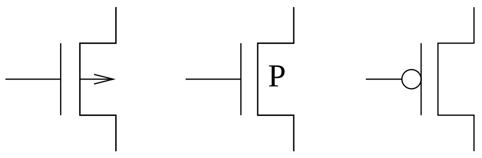


This is a P-channel enhancement mode MOSFET.
Here a negative charge on the gate induces a positive charge in the N-doped material immediately under the gate (*i.e.* in the “channel”). When more holes (positive charge carriers) are attracted into this region than there are electrons (negative charge carriers), the channel acts like a P-type material, and current can flow between the *source* and *drain*.



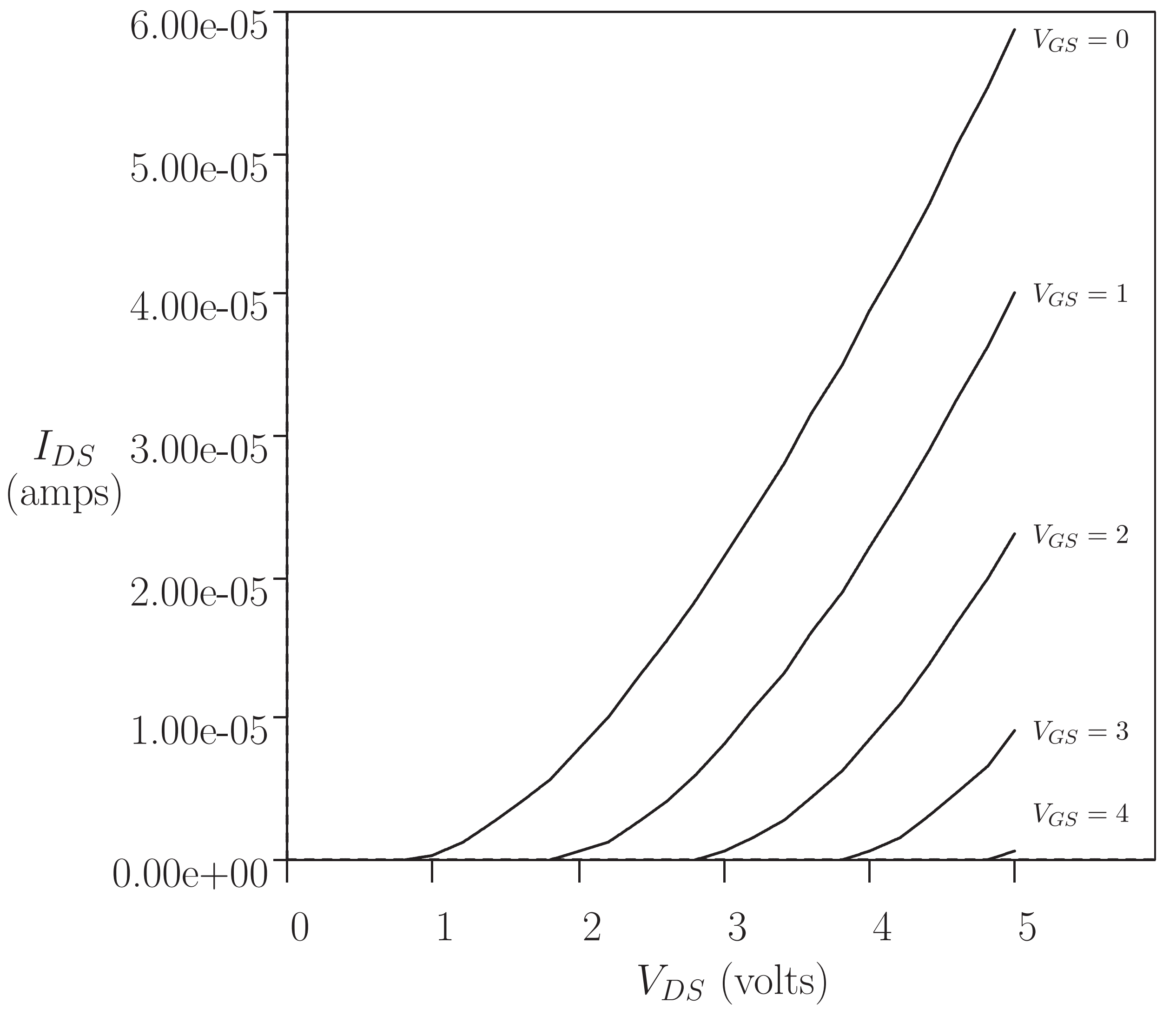
Usually, for PMOS devices, the substrate is “biased” at the supply voltage, so that any voltage below the supply voltage is negative with respect to the body of the transistor.

PMOS circuit symbols



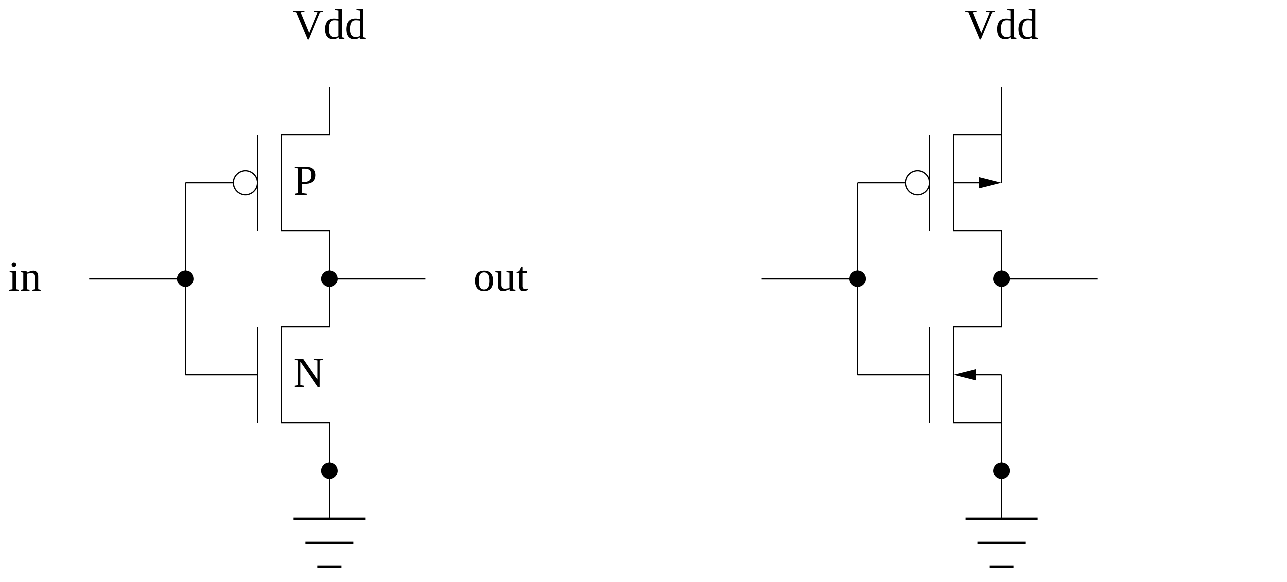
Following is a typical set of characteristic curves for a small PMOS transistor, with the substrate biased at 5V DC:

CMOS3 PMOS transistor DC characteristics - substrate biased $V_{DD}$



Most digital logic devices today are made using both NMOS and PMOS enhancement mode transistors (CMOS, or "complementary" MOS logic).
MOSFET devices act like voltage controlled switches. In these devices, the current is carried by only one type of charge carrier; either electrons of holes.
Because holes diffuse more slowly than electrons, typically PMOS devices switch slightly slower than NMOS devices.
MOSFET devices typically use very little current to operate, and this current only flows when the device is changing state. Basically, the gate is a capacitor, and the only sufficient current is required to charge the capacitor. When it is charged, no further current is needed.
As switches, MOS devices can have very low "on" resistance, and very high "off" resistance, making them very close to ideal switches.

A pair of complementary (PMOS and NMOS) transistors are often used to provide a connection to either power power or ground, depending on whether the input is 0 or 1:

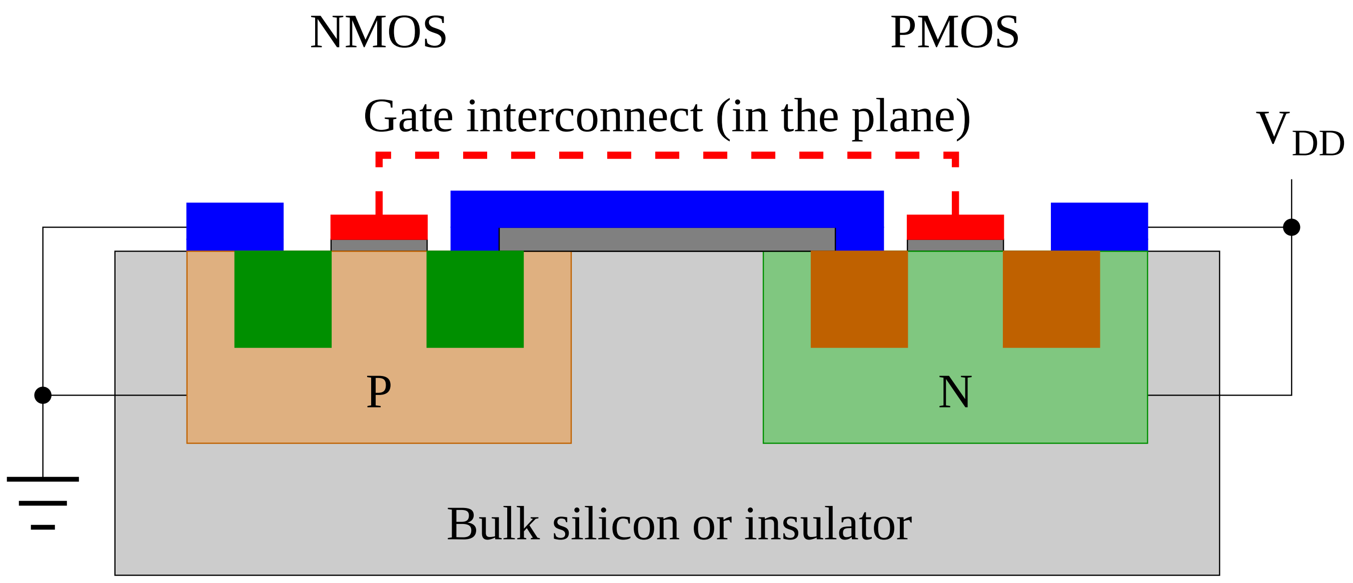


(Showing substrate connections)

Two such devices can be used to provide bi-directional control to, say, a motor.
Many MOSFET motor drivers have a similar output stage.

Most motor drivers are still made using bipolar transistors, including their output stages, however.

Following is a typical layout for a CMOS output stage, as shown on the previous page:



Note that the N-doped body for the PMOS transistor is connected to the supply voltage, while the P-doped body for the NMOS transistor is connected to ground.

Modern CMOS circuits consist of literally billions of similar structures.

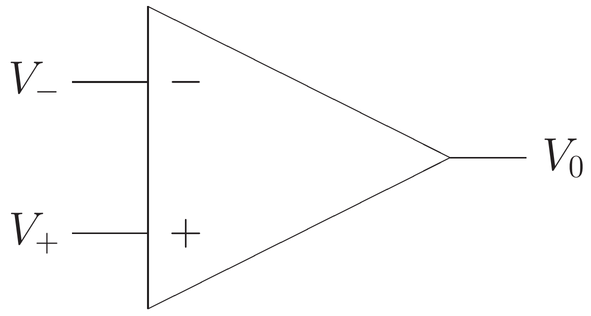
## The operational amplifier

Up to now, we have been primarily concerned with digital devices — devices having two states only.
Now we will talk about some ways in which a (continuous) electrical signal can be modified, or operated upon.
We will primarily consider a single device, called an operational amplifier, or *op amp*.
We will discuss mainly *ideal* op amps, occasionally noting where the fact that a circuit must be implemented with non-ideal devices may cause some problems.
An operational amplifier is a device with 2 input terminals and one output terminal (of course, it also has terminals for power, usually requiring both a positive and negative power supply, and ground), in which the voltage $V_0$ between the output terminal and ground is related to the voltage difference between the two input terminals, designated as $+$ and $-$, whose voltages are $V_+$ and $V_-$ respectively, relative to ground, by the equation

$$V_0 = A(V_+ - V_-)$$

where A is the amplification (often called the open loop amplification) for the op amp. For an ideal op amp, A is infinite.
The $+$ input is usually called the non-inverting input, and the $-$ input is called the inverting input.
(Note that when $V_+ = 0$, then $V_0 = -AV_-$).

The op amp is called a *linear* device, because the output voltage is linearly related to the input voltage.
Ideal op amps require zero input current for $V_+$ and $V_-$.
The circuit symbol for an op amp is shown here:



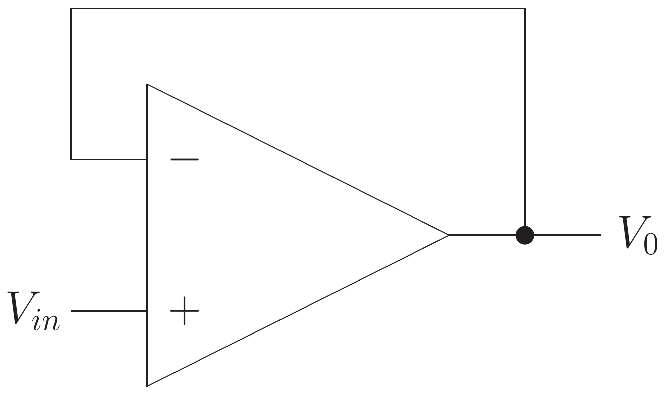
Op amps are currently available as a single integrated circuit (abbreviated IC) package, with up to 4 op amps to a single IC, and cost approximately the same as simple digital logic devices.
In practice, the output of an op amp cannot exceed the voltage of its power supply; typically 5–30V.
Since the difference between the inputs can be either positive or negative, op amps typically have both positive and negative power supplies.
In special cases, we can use certain op amps with a single (positive) voltage supply.

Clearly, an op amp is seldom used as an amplifier, directly, because of its nearly infinite gain. About the only direct application would be as an analog comparator.

## Feedback in an op amp

Typically, "feedback" is used to accurately control (reduce) the amplification, called the gain, of the op amp, and to give it other desirable features as well.

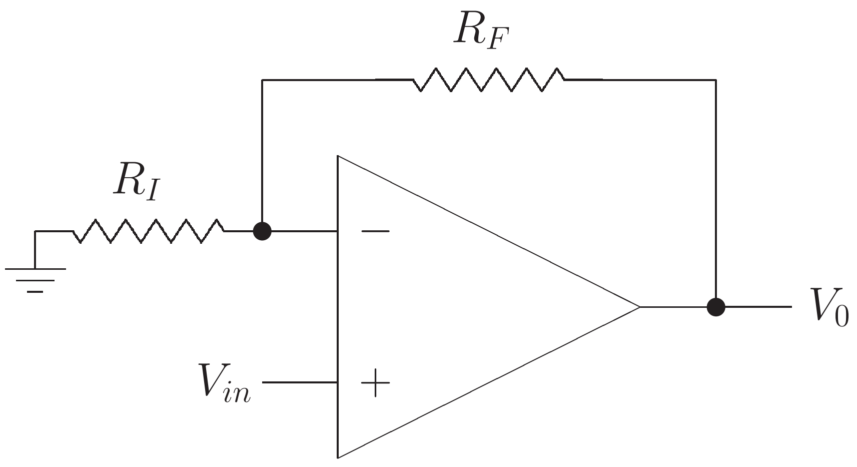
Perhaps the simplest example of feedback is shown in the following circuit, called a "voltage follower."
Here, if $A \to \infty$ , $V_{out} = V_+$.
This is an example of a "non-inverting amplifier" — the output has the same sign as the input.



$$\begin{aligned} V_0 &= A(V_+ - V_-) \\ \text{but } V_- = V_0, \text{ so} \qquad V_0 &= A(V_+ - V_0) \\ V_0 &= \frac{A}{1+A} V_+ \\ &= V_+ \qquad\qquad \text{if } A \to \infty \end{aligned}$$

This is really a specific example of the more general "non-inverting amplifier" shown in the next figure.

## The non-inverting amplifier

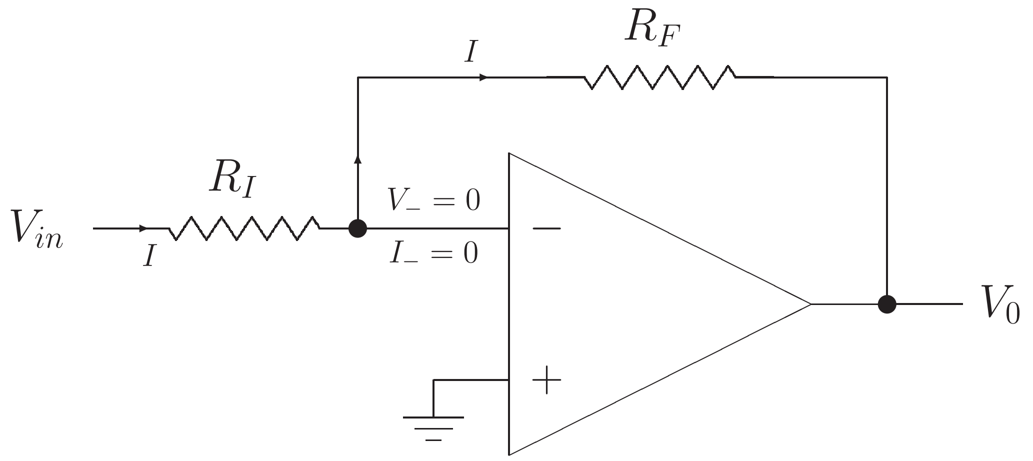


$$\begin{aligned} \text{here} \qquad V_0 &= A(V_+ - V_-) \\ \text{and} \qquad V_- &= \frac{R_I}{R_I + R_F} V_0 \\ \text{so} \qquad V_0 &= \frac{A}{1 + A(R_I/(R_I + R_F))} V_{in} \\ \text{if} \qquad A \to \infty, \qquad V_0 &= \frac{R_I + R_F}{R_I} V_{in} \\ &= (1 + R_F/R_I) V_{in} \end{aligned}$$

which is independent of A — the resistors in the feedback path determine the gain of the amplifier circuit.
We can have any gain we wish, simply by choosing appropriate values of $R_F$ and $R_I$.

## The inverting amplifier

The following diagram shows a second feedback configuration for an op amp called the "inverting amplifier."
[Note that in this amplifier the non-inverting (+) input is grounded. This is for convenience in discussion only, and is not required.]



Here, since no current flows into the inverting input; i.e., $I_- = 0$, then the same current, I, must flow through both $R_I$ and $R_F$. Therefore:

$$
\begin{aligned}
I &= \frac{V_{in} - V_-}{R_I} = \frac{V_- - V_0}{R_F} && \text{(by Ohm's law)}\\
V_- - V_0 &= \frac{R_F}{R_I}(V_{in} - V_-) &&\\
\text{but} \qquad V_0 &= A(V_+ - V_-) = -AV_- && (\text{since} \quad V_+ = 0)\\
\text{so} \qquad V_- &= -V_0/A &&\\
\text{and} \qquad V_0 &= -\frac{V_0}{A} - \frac{R_F}{R_I}(V_{in} + V_0/A) &&\\
\text{as} \quad A \to \infty, \qquad V_0 &= -\frac{R_F}{R_I}V_{in} &&
\end{aligned}
$$

Note that the amplification, or gain, is again determined only by the characteristics of the feedback loop, not the amplifier, provided that A is sufficiently large (and the ratio $R_F/R_I$ is not chosen to be too large).

Note also, that in this example, $V_-$ is 0. In this case, the inverting input is said to be a "virtual ground". This, of course, is what one would expect of any differential amplifier with *negative* feedback — the output $V_0$ is proportional to the difference between the two inputs, and has the opposite sign. Since this output forms part of the input as well, it would always tend to *decrease* the difference between the two inputs.

One difference between the non-inverting and inverting configurations is rather important.
The non-inverting amplifier has an infinite input impedance.
(The input is connected to $V_+$ directly, which has infinite resistance.)

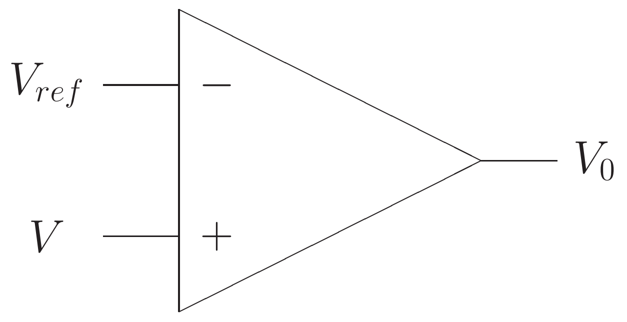
The inverting amplifier has an input impedance $R_I$.
In this sense, the inverting amplifier produces an output voltage proportional to the input current, with R as the constant of proportionality.

The inverting amplifier configuration is used much more often than the non-inverting configuration, but for single voltage supply applications the non-inverting configuration is very useful.

## OP-amp circuits

There are several op amp circuits which, although quite simple, are extremely useful. The simplest of these is the comparator circuit, shown following.

Here, an unknown voltage, V, is compared to a reference voltage, $V_{ref}$. If $V > V_{ref}$, then the output is $+\infty$ (the actual output will be the maximum voltage the op amp can produce). If $V < V_{ref}$, then the output is $-\infty$ (the actual output will be the minimum voltage the op amp can produce).

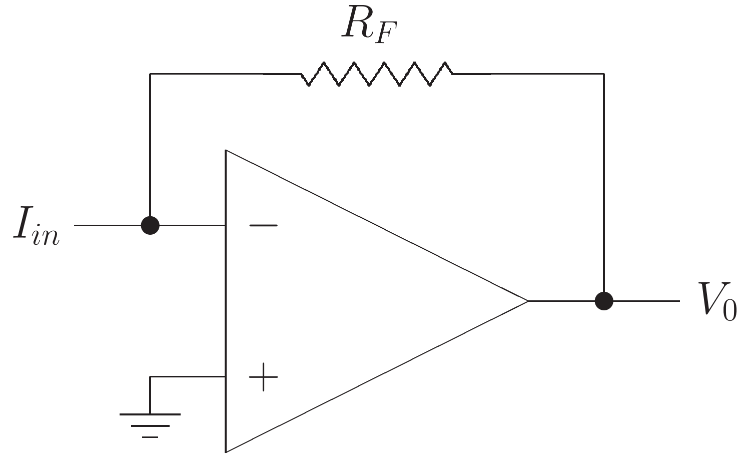


In these cases, the output from the op amp *saturates* — it is set firmly to its maximum value.

In practice, an op-amp may take a long time to recover from a saturated output, so if response time is important, using a device designed specifically as a comparator is best.

Such devices are available commercially, and are characterized by their response time.

A current-to-voltage converter can be constructed as follows:



This is essentially an inverting amplifier, but with no input resistor; the input current $I_{in}$ is applied directly to the inverting $(-)$ input of the op amp. The input voltage is then given by

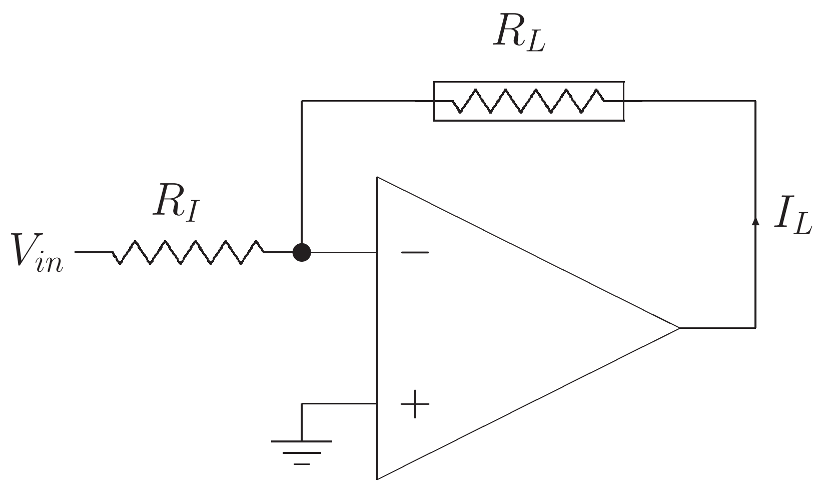
$$V_0 = -I_{in}R_F$$

This circuit is often used to convert current output from devices like photodiodes or photomultipliers to a voltage to be measured using, say, an analog-to-digital converter.

Of course, a simple resistor has the same property $(V = IR)$, but the output of the op amp can be used to power another device.

Recall that the AVR ADC was designed to be used with a low impedance ($<$ 10K ohms) so the op amp circuit is often preferable.

A voltage-to-current converter can be constructed as follows:



where $R_L$ is the resistance of the load, or device which receives the constant current.

This device, called a "transconductance amplifier" is again a simple inverting amplifier configuration.

Here, since no current flows into the inverting input of the op amp,

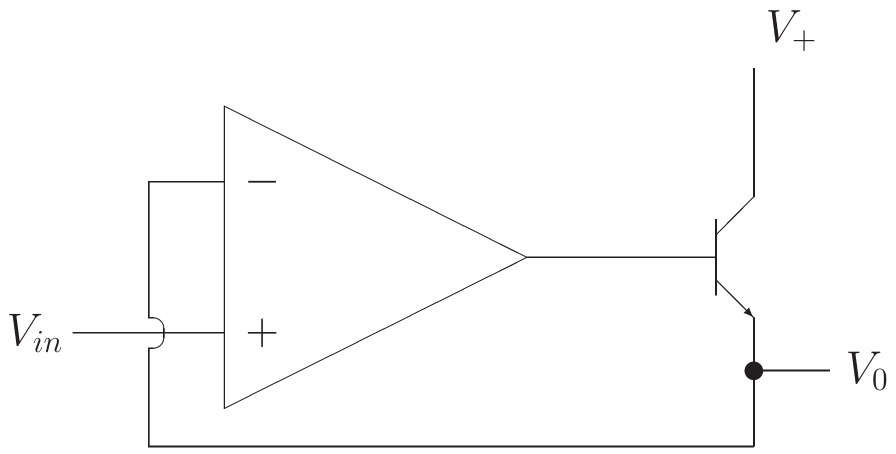
$$\begin{aligned} V_I &= I_L R_I \\ I_L &= V_I / R_I \end{aligned}$$

independent of the value of the load resistance, $R_L$.

This circuit can be used with a *fixed* input voltage, $V_{in}$, to provide a constant current source.

In fact, constant current sources are available commercially, so it is not common to construct one using an op amp circuit.

Virtually any circuit an be used in the feedback loop of an op amp. In fact, a simple way to provide a high power output from an op amp circuit is to have a power transistor in the feedback loop, as follows:



This example shows a unity gain (voltage follower) op amp configuration, but any of the amplifier configurations will also work. ($V_+$ is the voltage supply for the transistor; it need not be the same as for the op amp, but they must share a common ground reference.)

Switching higher voltages and currents does require care, however.

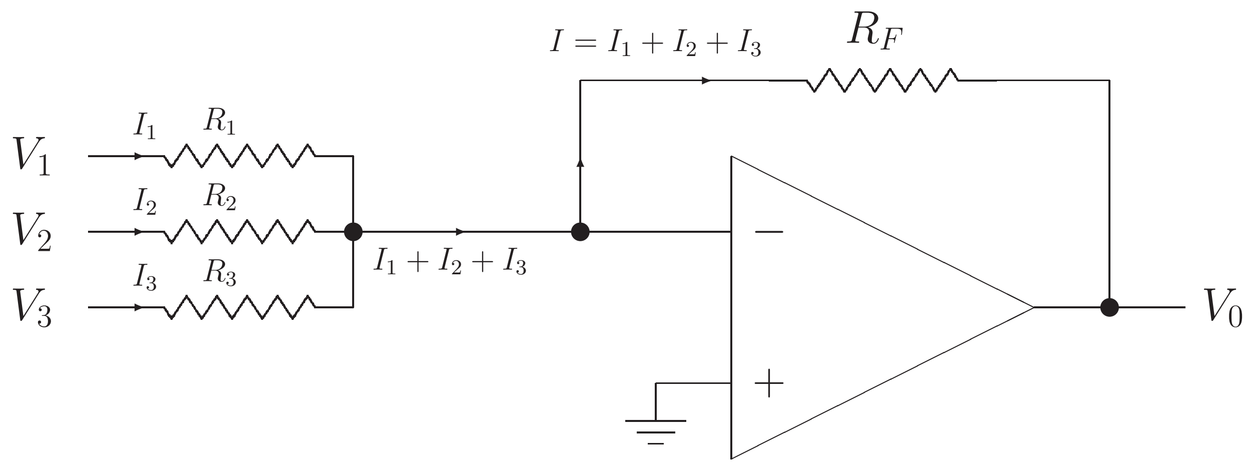
## Mathematical operations using operational amplifiers

The operational amplifier can be used to perform arithmetic operations on the input voltage waveforms. In fact, this is what gives the device its name.

It is immediately obvious from the preceding equations that an inverter (i.e., a sign changer), or a scale changer, (i.e. a constant multiplier) can be easily constructed.

Also, a level shifter (i.e., addition of a constant) can be performed, and if complex impedances are used for $R_I$ and $R_F$, a phase shifter can be constructed.

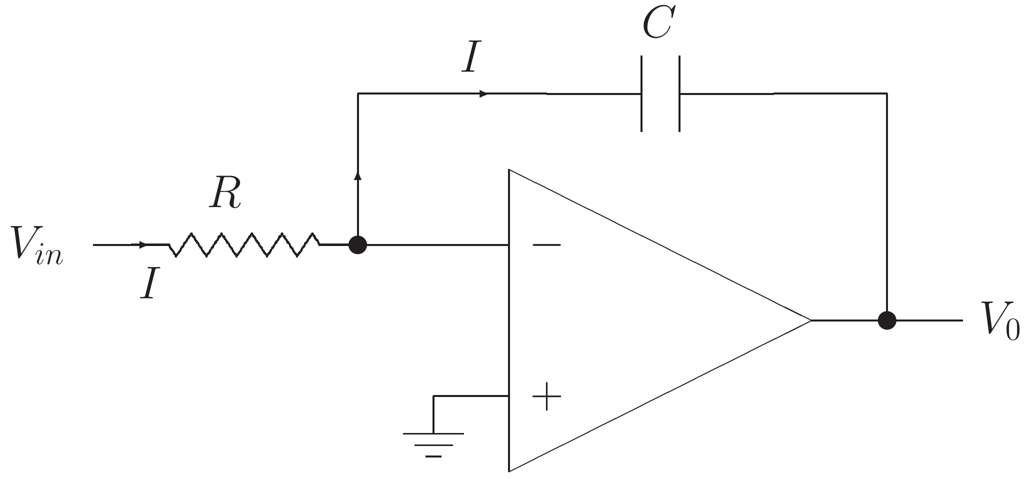
A circuit which adds together several input voltages can be constructed quite easily, for either of the op amp configurations. The case of the inverting amplifier is easiest:



$$\begin{aligned} I &= I_1 + I_2 + I_3 \\ &= V_1/R_1 + V_2/R_2 + V_3/R_3 \\ &= -V_0/R_F \\ \text{if} \quad R_1 &= R_2 = R_3 = R_F, \\ \text{then} \quad V_0 &= -(V_1 + V_2 + V_3) \end{aligned}$$

In general, the output $V_0$ is the weighted sum of $V_1$, $V_2$, $V_3$ .... where the weights are $R_F/R_1$, $R_F/R_2$, etc. Subtraction can be performed by inverting the required input. Any desired phase shift can be accomplished by adding reactive components to $R_1$, $R_2$, etc. Note that the input signals $V_1$, $V_2$, etc. can be time varying signals. They must only satisfy the (non-ideal) criteria that they are within the (frequency) bandwidth of the op amp, and that the rate of change of output voltage not exceed the slew rate of the op amp.

Other operations can be performed as well, including integration and differentiation. In fact, it is in the solution of differential equations that operational amplifiers found their greatest traditional use, in analog computers.
The following circuit can perform integration; $V_0 = -1/RC \int V_{in} dt$

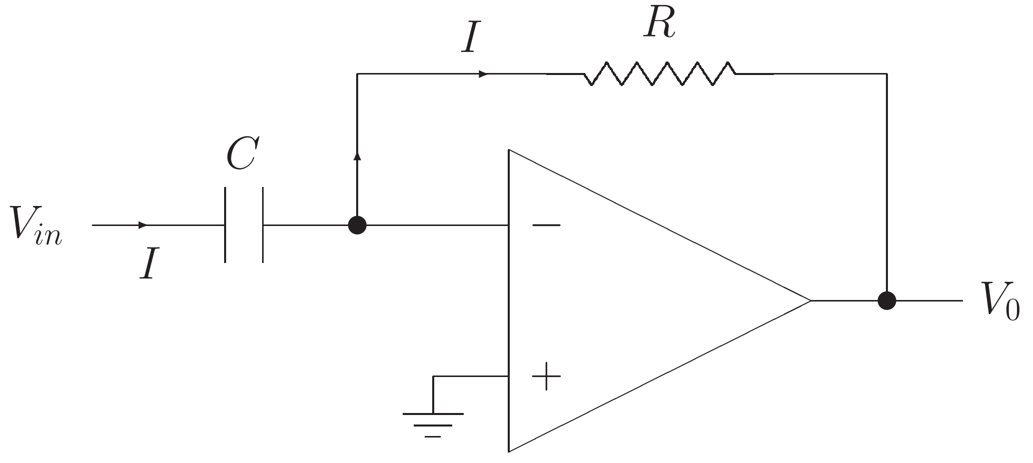


For a capacitor, $Q = CV$ where C = capacitance, Q = charge, V = voltage across capacitor. The current $I = dQ/dt$ = rate of flow of charge. As before,

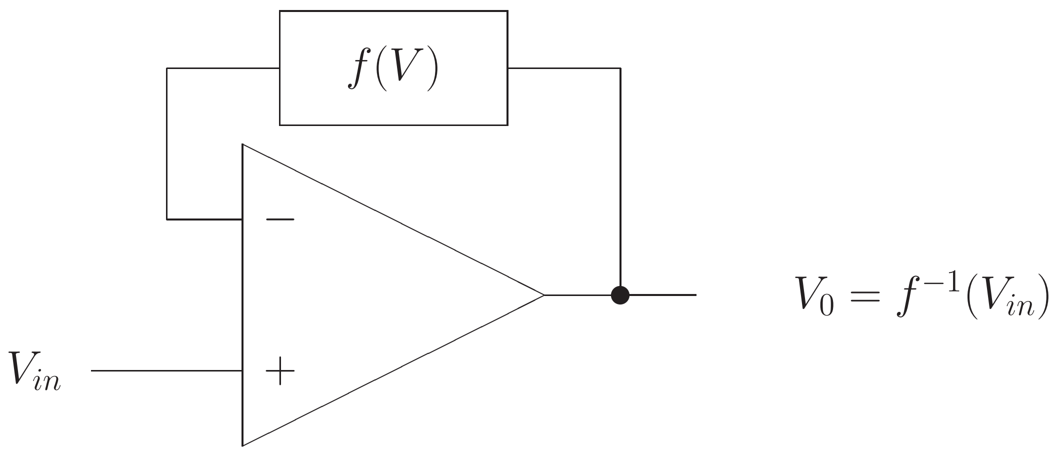
$$\begin{aligned} I &= V_{in}/R = dQ/dt = \frac{d}{dt}(-CV_0) = -C\frac{dV_0}{dt} \\ dV_0/dt &= -1/C \times V_{in}/R \\ V_0 &= -1/RC \int V_{in} dt \end{aligned}$$

If we interchange R and C in the previous circuit, we have a device which can perform differentiation, as follows:

$$\begin{aligned} I &= dQ/dt = \frac{d}{dt}(-CV_{in}) = -V_0/R \\ V_0 &= -RC\,\frac{dV_{in}}{dt} \end{aligned}$$



A most useful function of op amps is that if a device is available to perform any mathematical function on an input signal, then the inverse of this operation can be performed by placing the device which performs this operation in the feedback loop of a non-inverting op amp as follows:



$$\begin{aligned} V_0 &= A[V_{in} - f(V_0)] \\ AV_{in} &= V_0 + Af(V_0) \\ \text{if } A \to \infty, \text{ then} \qquad f(V_0) &= V_{in} \\ \text{or} \qquad V_0 &= f^{-1}(V_{in}) \end{aligned}$$

Analog circuits are readily available, at reasonable prices, which can perform the following functions:

| | |
|---|---|
| multiplication | division |
| exponentiation | logarithm extraction |
| square | square root |
| ideal diode | ideal comparator |
| RMS extraction | |

Other functions can be derived from these, or at least closely approximated. The IC's are available at reasonable cost (from a few cents to a few dollars, in single quantities), with accuracy of from about 1% to better than 0.1%, and bandwidth of 1 - 100 MHz. This corresponds to, in the digital world, accuracy of from 7 to 10 bits, and function evaluation times of about 0.1 $\mu$s; comparing this to a typical microprocessor, a single add operation is performed in less than 1 ns.

In general, it is possible to solve mathematical problems, with limited accuracy, in very short times using analog techniques.

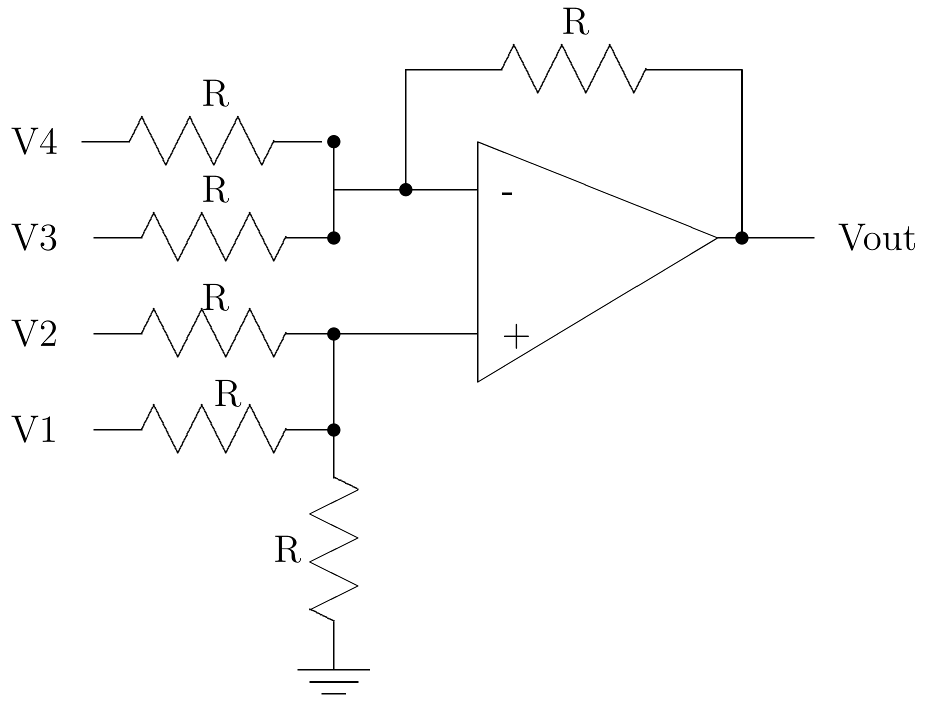
One problem with analog computers, however, is that in order to program them, the functional blocks must be physically connected together.

Today, it is possible to assemble an array of analog "building blocks", and use digitally controlled analog switches, multiplexors, etc. to, in a sense, "program" an analog computer. (Generally, though, these are used only for very specific types of functions. Analog computation, *per se*, is long obsolete.)

## Single supply op amp circuits

For single supply op amps, typically the non-inverting configuration is preferred.
In any case, the differential input between the + and − terminals should never be negative.

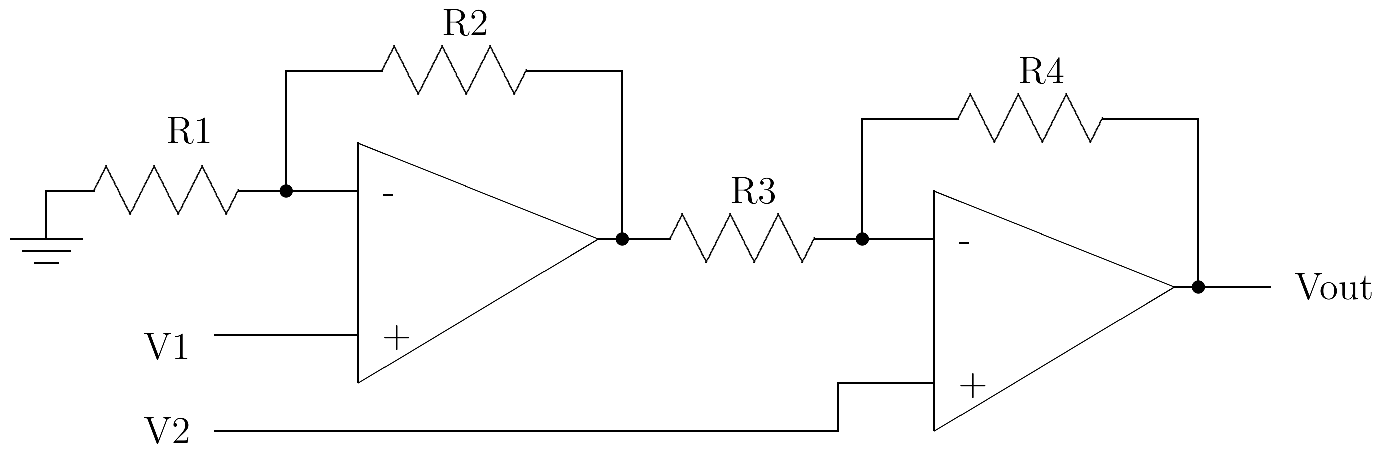
The following is a summing amplifier:



$$V_{out} = V_1 + V_2 - (V_3 + V_4)$$

Typically, R is chosen to be fairly large, say about 100K ohms.

The following is a high input impedance differential amplifier:

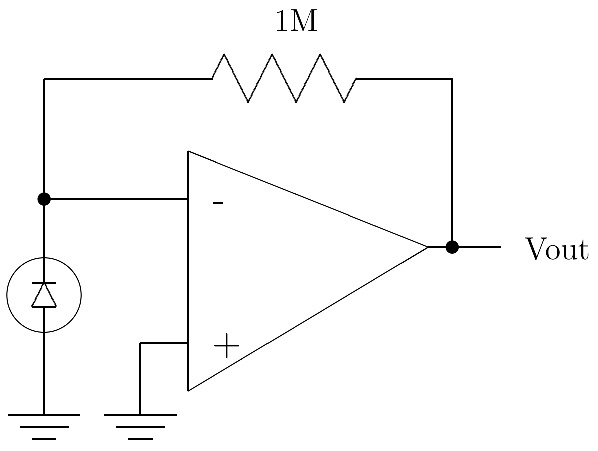


Here, $\frac{\mathrm{R}_1}{\mathrm{R}_2} = \frac{\mathrm{R}_3}{\mathrm{R}_4}$ in order to maintain the "common mode rejection ratio." (Basically, it should not preferentially amplify the noise in one stage.)

In this case, $V_{out} = (1 + \mathrm{R}_4/\mathrm{R}_3)(V2 - V1)$
(We assume $V_2 \geq V_1$).

Again, the resistor values are typically chosen to be reasonably large; in the range of 100K ohms.

The following is an interesting circuit used as an amplifier for a light sensor (photovoltaic device — outputs a voltage when light shines on it):



An interesting thing here is that the photocell has 0V across it.

This would be typical of our kind of use for an op amp — conditioning a signal so that an ADC or comparator would have a reasonable dynamic range for an input.
This typically means scaling (amplifying) a signal, and/or adding or subtracting some value from it.

## Design example

Design a circuit to enable accurate digitization of temperature (to within 0.2 degree) in the range 0 to 40 Celsius.
Use the LM335 temperature sensor, and the ADC on the ATmega1284p.

The LM335 produces an output of 10 mV/K, so at 0C the output would ideally be 2.73 V (corresponding to a temperature of 273K). At 40C the temperature would be 313K, and the output 3.13V. The difference is 0.4V, which must be measured in steps of at least 0.2C, or 2mV. The ADC can digitize to 10 bits, so its minimum resolution is $\approx 0.1\%$ of $V_{\text{ref}}$. If $V_{\text{ref}}$ is 5V, then this corresponds to 5mV.

In order to get this accuracy, we need to amplify the output of the sensor (by a factor of at least 2.5). If we do this, the output will be over 5V, which is beyond the input range of the ADC.
Since we do not need to consider temperatures below 0C (273K) we can subtract 2.73V from the output, and amplify only the difference. Therefore, we need to

- set up the temperature sensor
- subtract a fixed voltage (2.73V) from the output
- amplify the new value by at least 2.5 (say, 10)
- digitize the amplified value

We also assume that we can calibrate the whole system after it is assembled.

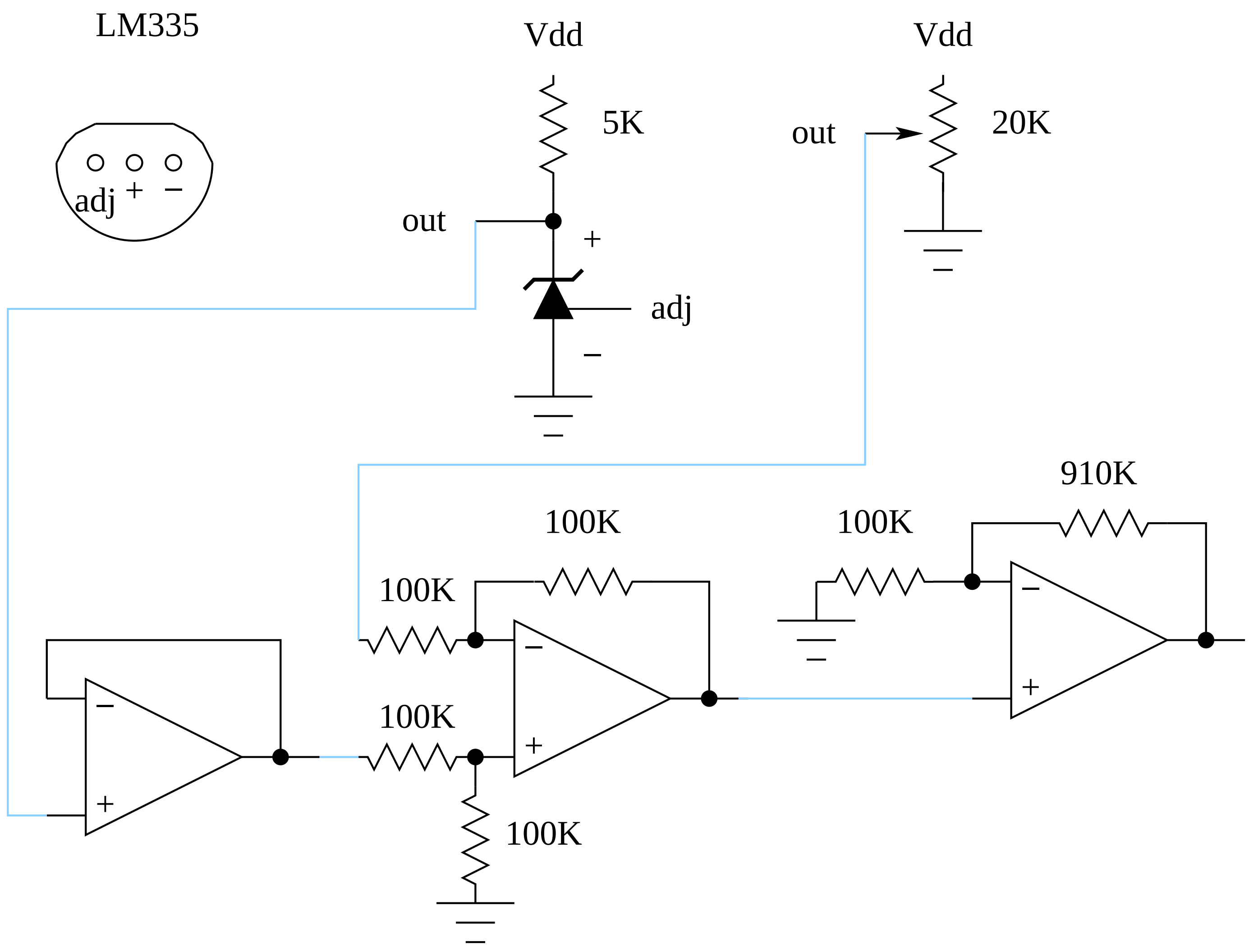
We will use a single supply op amp (the MCP6241/4) to do this "signal conditioning." (There are many more that would be effective, as well; e.g., the TLC2254, or even the LM324.)
We choose this op amp because

- it is a low power single supply device
- it has rail-to-rail output
- it uses the same power supply range as the AVR processors
- it is low cost, and I have some on hand
- it comes with 4 op amps in one 14 pin chip

Although it can all be done using one op amp, we will use three, in one package.
The first will be a unity gain configuration, to isolate the sensor from the rest of the circuitry.
The second will be a single supply summer (actually a subtracter.)
The third will be a non-inverting amplifier configuration.

One requirement is a source for the 2.73 V to be subtracted.
We can use a (resistive) voltage divider (but the voltage will then depend on the supply voltage, or a constant current source supplying a fixed resistor, or a "voltage reference" where we amply the voltage with an op amp.
We will use the voltage divider, but the other methods are preferable.

## Circuit diagram for example:



## How long will this design run if powered by battery?

We can calculate the approximate current used by the various components, assuming a 4.5 – 6 V battery, and high resistance output:

| | | |
|---|---|---|
| LM335 | $\approx$ 1 ma | I = V/R = (6V - 2.73V)/5K = 0.65 ma |
| 20K pot | $\approx$ 0.3 ma | I = 6V/20K = 0.3 ma |
| op amp | $\approx$ 1 ma | input resistance 100K (< 0.1 ma/stage) |
| total | < 2 ma | |

The capacity of some common batteries is tabulated below:

| Type | voltage | capacity (ma-H) |
|---|---|---|
| Alkaline | | |
| AAA | 1.5 | 1250 |
| AA | 1.5 | 2700 |
| C | 1.5 | 8000 |
| D | 1.5 | 12000 |
| 9V | 9 | 565 |
| Lantern | 6 | 26000 |
| Lithium | | |
| CR123 | 3 | 1500 |
| CR2032 | 3 | 220 |
| CR2477 | 3 | 1000 |

How can the battery life be extended?

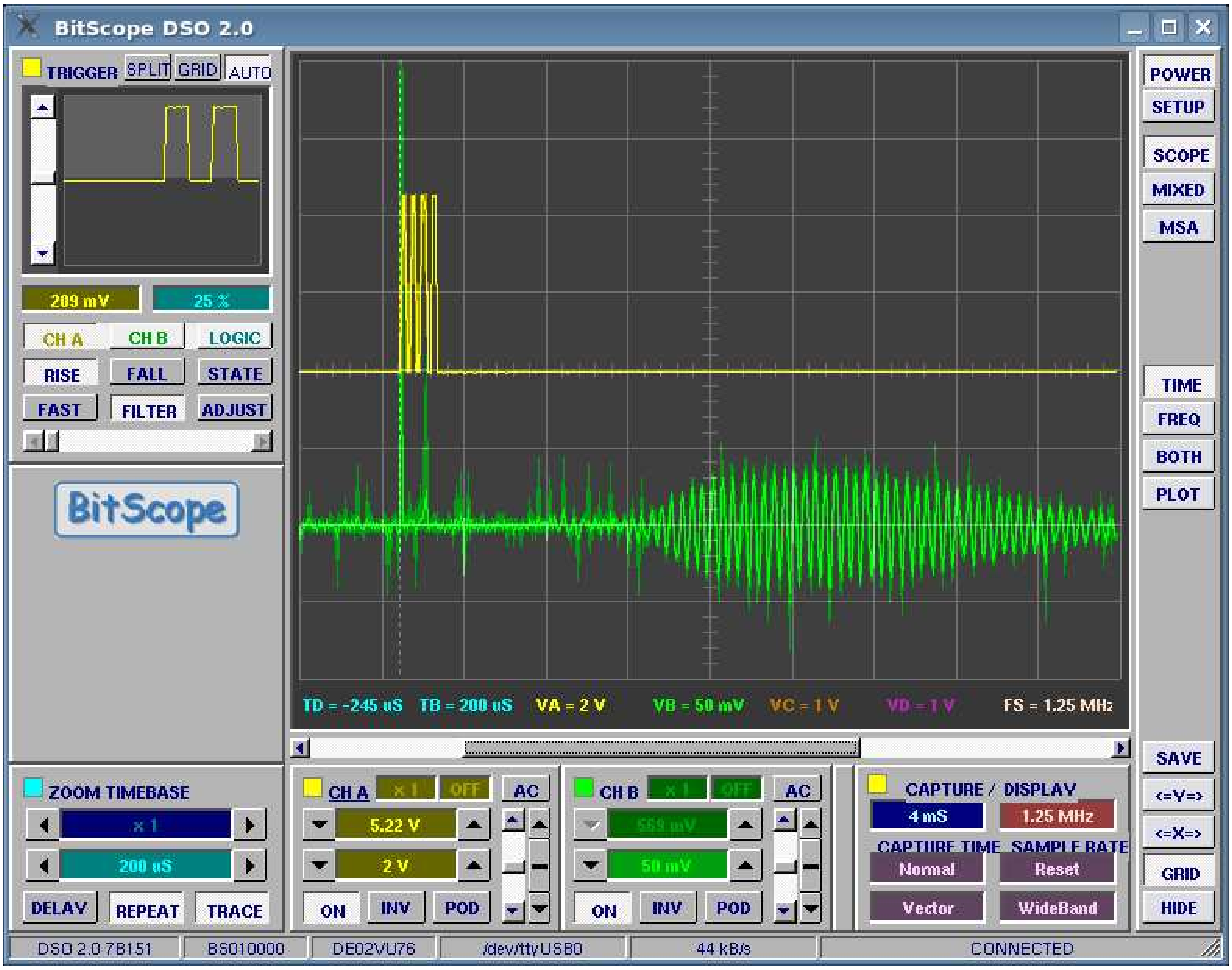
## Another design example

Given a 40 KHz. ultrasonic transmitter and receiver, design a circuit to detect when a sensor is "close" to a solid object. (I.e., when a return signal of a certain amplitude can be detected.)
The same idea can be used to measure the distance from a transducer to a solid object.

Following is an oscilloscope trace of the output and received signal from a transmitter/receiver pair.
The yellow trace is the 40 KHz. input applied to the ultrasonic transmitter. The green trace is what is picked up by the receiver, with a reflector about 8–10 cm. away from the transducers.

Note the difference in the scales — about a factor of 100.

We therefore need to amply the returned signal by a factor of about 100.



209 mV 25 %

TD = -245 uS TB = 200 uS VA = 2 V VB = 50 mV VC = 1 V VD = 1 V FS = 1.25 MHz

ZOOM TIMEBASE × 1, 200 uS

CH A × 1 OFF AC 5.22 V 2 V

CH B × 1 OFF AC 569 mV 50 mV

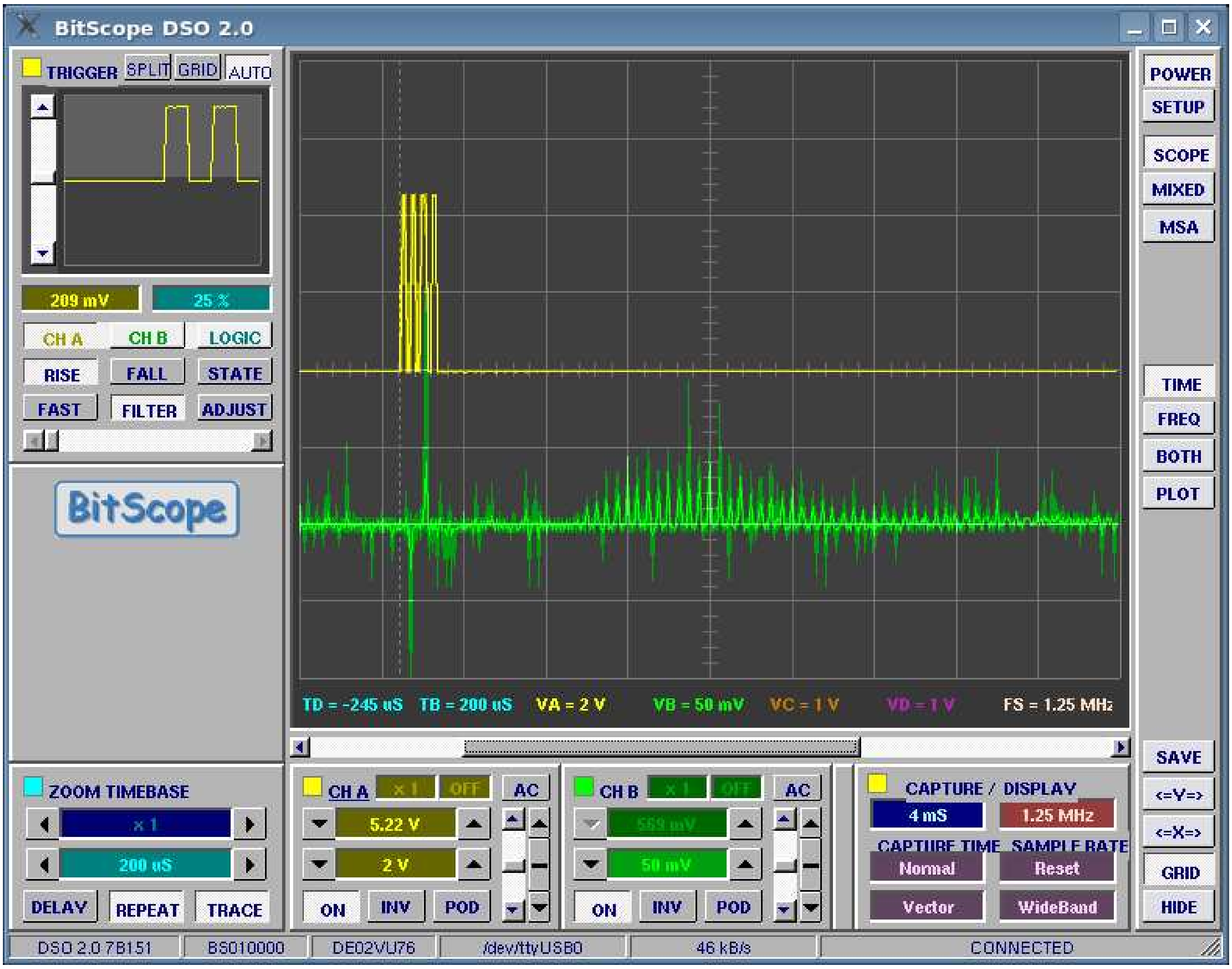
CAPTURE TIME 4 mS SAMPLE RATE 1.25 MHz

DSO 2.0 7B151 | BS010000 | DE02VU76 | /dev/ttyUSB0 | 44 kB/s | CONNECTED

Again, we will use a unity gain op amp configuration as the first stage.

An added benefit this time is that, since we are using a single supply op amp, the negative portion of the returned signal will be eliminated.

Following is the output from this stage:



209 mV   25 %

TD = -245 uS   TB = 200 uS   VA = 2 V   VB = 50 mV   VC = 1 V   VD = 1 V   FS = 1.25 MHz

ZOOM TIMEBASE: x 1, 200 uS

CH A: x 1, OFF, AC, 5.22 V, 2 V

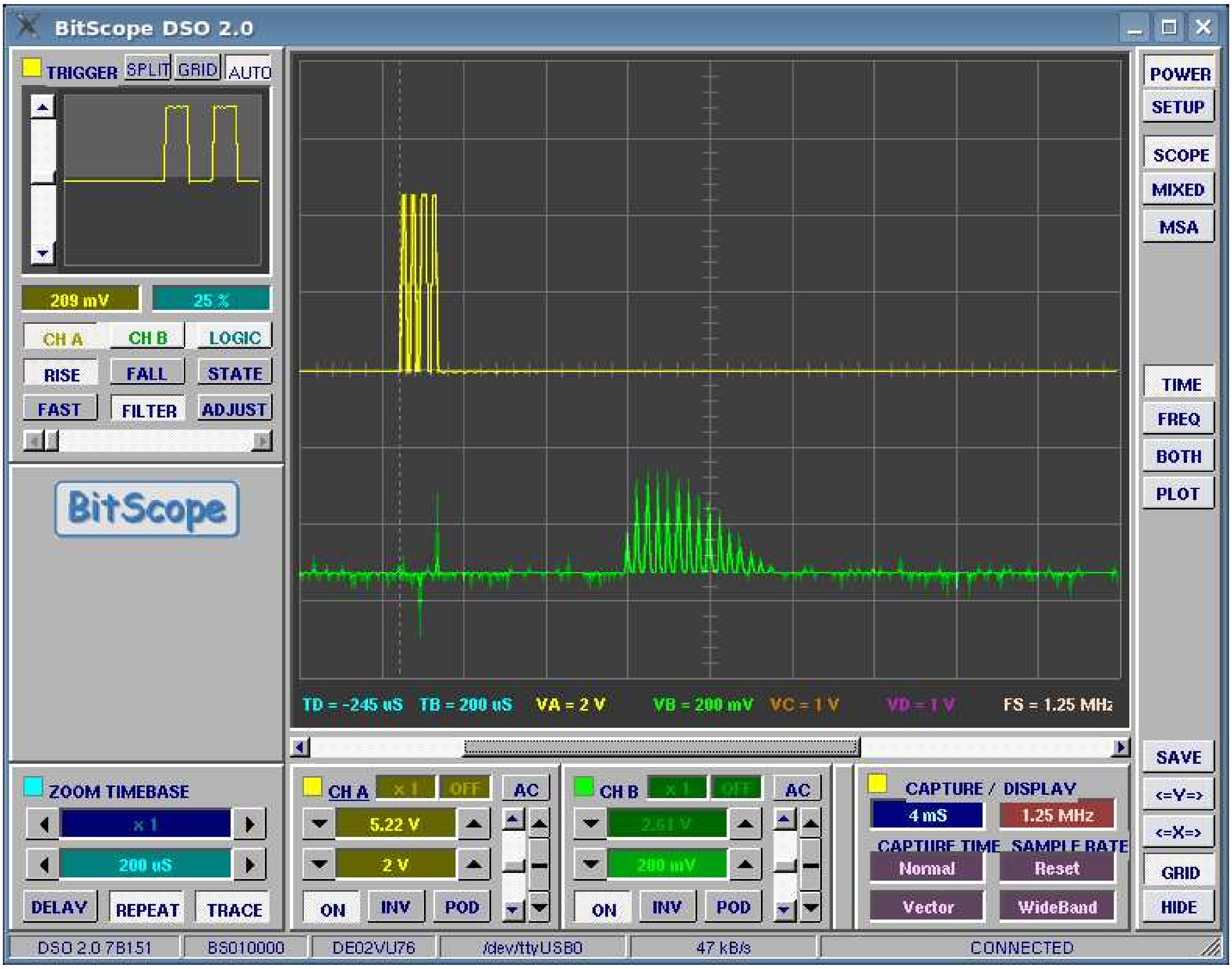
CH B: x 1, OFF, AC, 559 mV, 50 mV

CAPTURE / DISPLAY: 4 mS, 1.25 MHz

DSO 2.0 7B151 | BS010000 | DE02VU76 | /dev/ttyUSB0 | 46 kB/s | CONNECTED

We now need to amplify this signal by a factor of approximately 100. Since the gain-bandwidth product for this amplifier (MCP6244) is about 550 KHz, and we are amplifying a 40 KHz signal, we can achieve a gain of about 10 in one stage. Therefore, we need at least two such stages.

Following is the output from the first amplifier stage:



209 mV   25 %

ZOOM TIMEBASE x 1   200 uS

CH A x 1 OFF AC   5.22 V   2 V

CH B x 1 OFF AC   2.61 V   200 mV

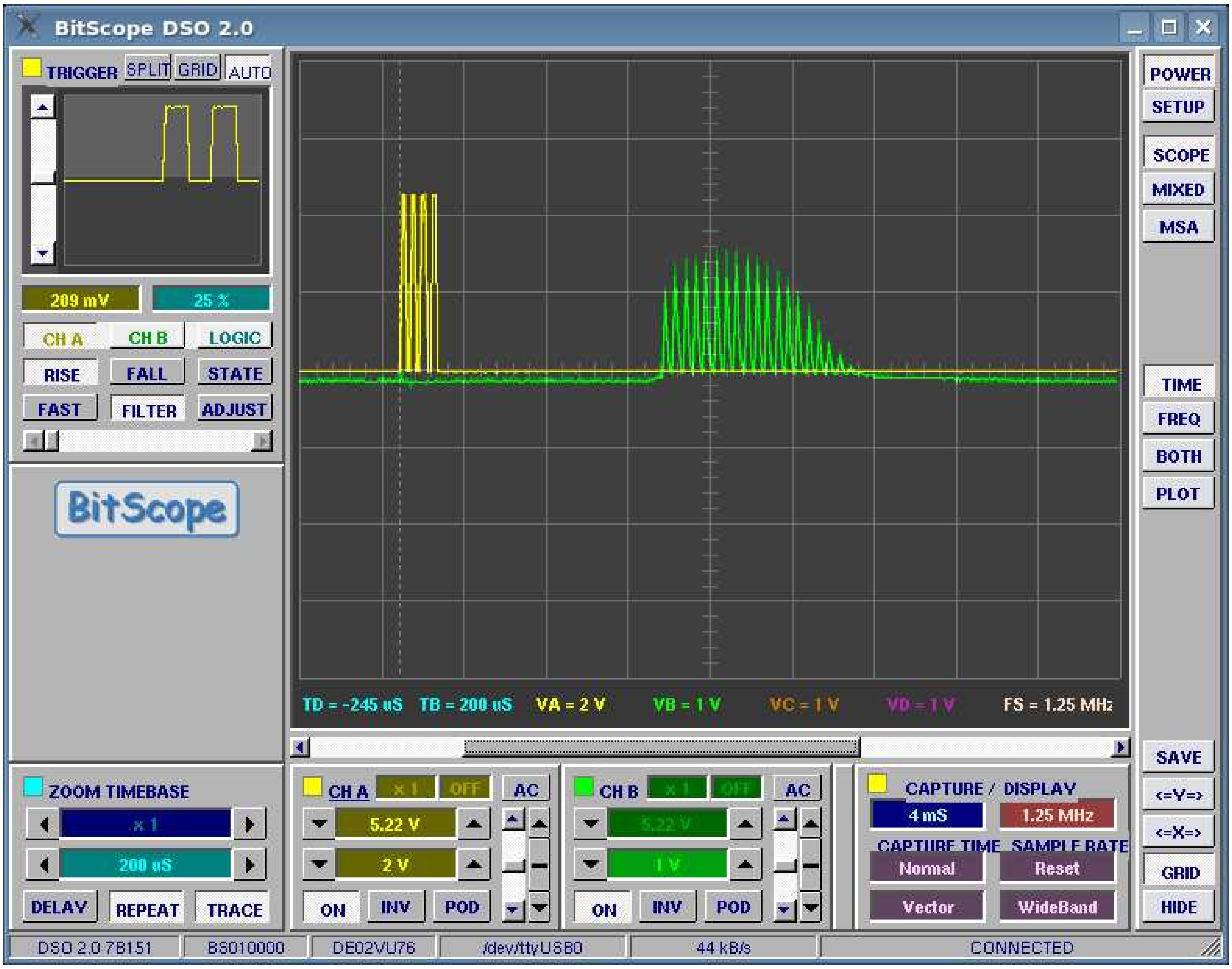
CAPTURE / DISPLAY   4 mS   1.25 MHz

TD = -245 uS   TB = 200 uS   VA = 2 V   VB = 200 mV   VC = 1 V   VD = 1 V   FS = 1.25 MHz

DSO 2.0 7B151 | BS010000 | DE02VU76 | /dev/ttyUSB0 | 47 kB/s | CONNECTED

Note the oscilloscope gain settings — the signal is amplified by a factor of about 10.

Following is the output from the second amplifier stage:



209 mV 25 %

TD = -245 uS TB = 200 uS VA = 2 V VB = 1 V VC = 1 V VD = 1 V FS = 1.25 MHz

ZOOM TIMEBASE x 1, 200 uS

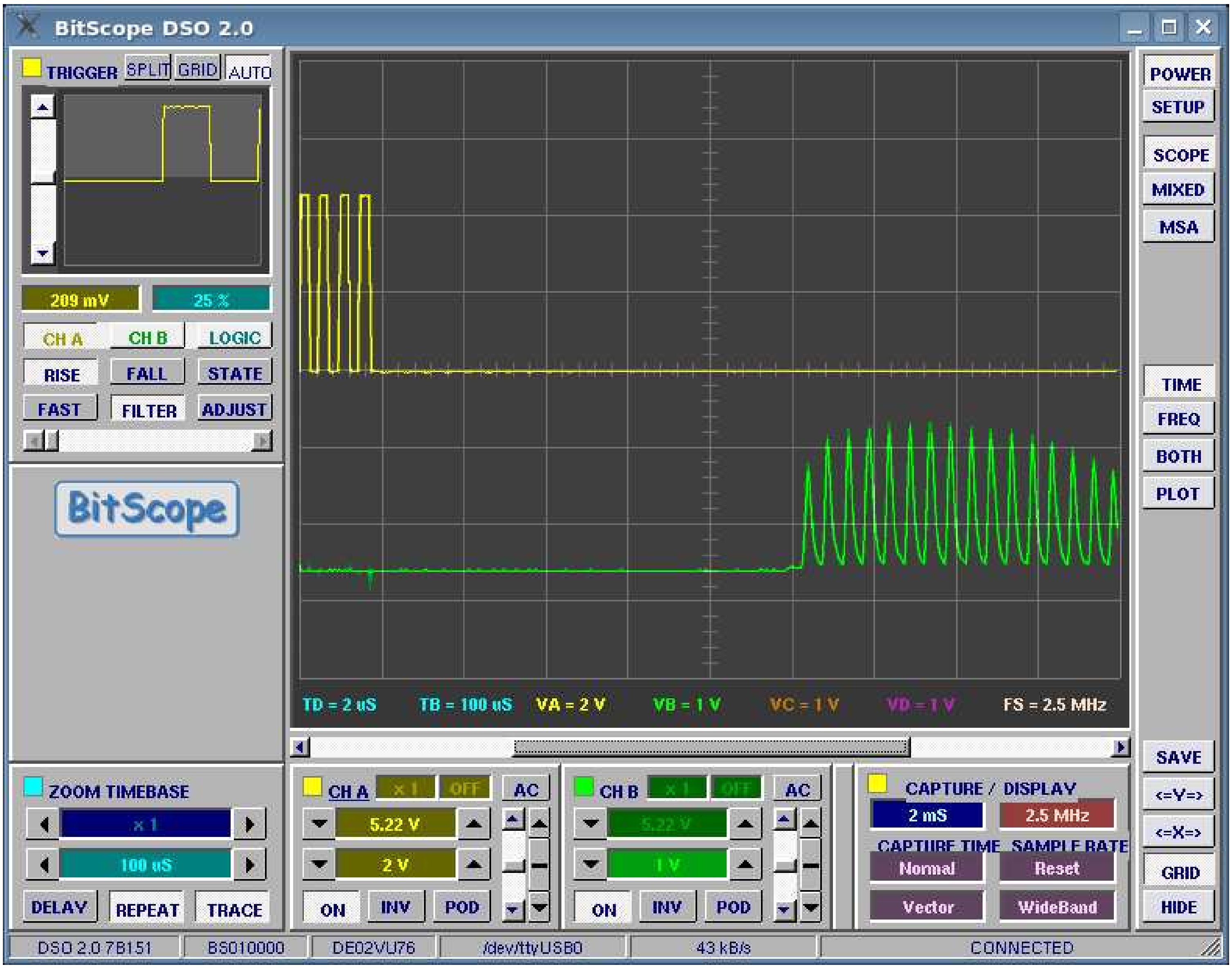
CH A x 1 OFF AC 5.22 V 2 V

CH B x 1 OFF AC 5.22 V 1 V

CAPTURE TIME 4 mS SAMPLE RATE 1.25 MHz

DSO 2.0 7B151 | BS010000 | DE02VU76 | /dev/ttyUSB0 | 44 kB/s | CONNECTED

Following is the output from the second amplifier stage, again, with the time scale expanded. Note the structure in the returned signal:


BitScope DSO 2.0

TRIGGER 209 mV 25 %

ZOOM TIMEBASE x 1 100 uS

TD = 2 uS TB = 100 uS VA = 2 V VB = 1 V VC = 1 V VD = 1 V FS = 2.5 MHz

CH A x 1 OFF AC 5.22 V 2 V

CH B x 1 OFF AC 5.22 V 1 V

CAPTURE / DISPLAY 2 mS 2.5 MHz CAPTURE TIME SAMPLE RATE Normal Reset Vector WideBand

DSO 2.0 7B151 | BS010000 | DE02VU76 | /dev/ttyUSB0 | 43 kB/s | CONNECTED

The position of the returned signal (time between the output and return pulse) varies with the distance to the reflector. A more powerful ultrasonic burst can be used to measure distances up to several meters with these transducers.

## Sensors and transducers

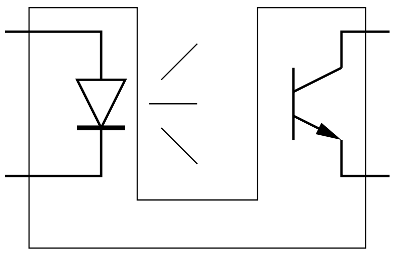
A *transducer* is a device that converts one type of energy to another. For example, a speaker converts electrical energy to sound.
The term transducer is also used for a device that converts one type of signal to another.
A *sensor* is transducer that allows some effect to be measured or sensed.
For example, a mercury thermometer is a sensor, where the output is easily read by a human. Such sensors — litmus paper for measuring pH is another example, are said to be *direct reading.*
More typically, though, when we call something a sensor, it is a device that converts something into a form that can be measured automatically, by electronic instruments, such as a voltage or current, or by direct digital methods (e.g., counting).
Many things can be sensed in different ways. For example, the speed of a bicycle can be obtained by measuring the time required for a revolution of the wheel. Knowing the circumference of the wheel, and the time for a full revolution, the speed can easily be calculated. The wheel could also be connected to a small electrical generator, and the output current measured, also giving an output related to the speed of revolution of the wheel.
Typically, we would call the first type of sensor — counting revolutions per unit time— a digital sensor, and the second — generating a voltage or current proportional to the speed – an analog sensor.

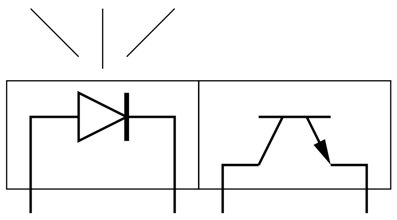
## Digital sensors

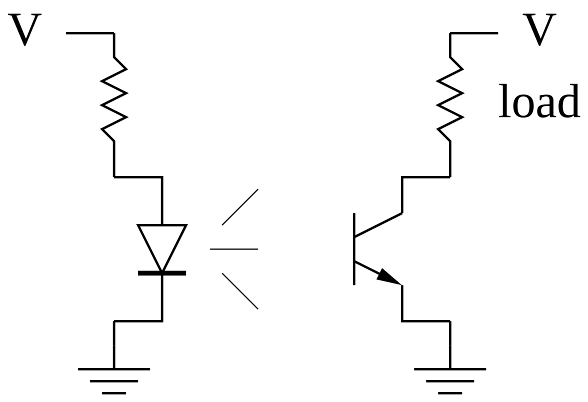
Digital sensors typically produce discrete outputs that can be counted (e.g., revolutions of a bicycle wheel) or outputs that can be timed, using a digital counter.

Often digital sensors include analog components. For example, a counter can have a photosensor that generates an electrical pulse whenever a reflecting surface is near the sensor. If the time between those pulses is related to the parameter to be measured, then the pulses can be used to build a digital sensor.

Typical optosensors use LEDs and phototransistors:


slotted


reflective



optocoupler

Often the analog component of a digital sensor is more elaborate than a simple switch. In the optical sensors, the emitter is usually am infrared (IR) LED, and the sensor is a phototransistor. The phototransistor can be used as a simple switch, turning a logic gate on or off, or as part of an amplifier circuit.

Other electrical parameters can also be used in digital sensors. Consider a capacitor that is allowed to charge to some voltage, through a fixed resistor. The time to charge the capacitor (up to some fixed voltage, say) is related to the voltage applied to the capacitor. An analog comparator could be used to generate an output when the target voltage was reached, and the charging time (measured with a digital clock) is a measure of the applied voltage.

If either the resistance or the capacitance changes, then the charging time also changes. This technique (or a variant of it, where the resistor and capacitor are in the feedback loop of an oscillator) is often used to measure changes in resistance or capacitance.

Devices which vary resistance and/or capacitance can be constructed which measure many physical parameters — pressure, temperature, sound, humidity, force, light intensity, radiation, and many others.

## The 555 timer

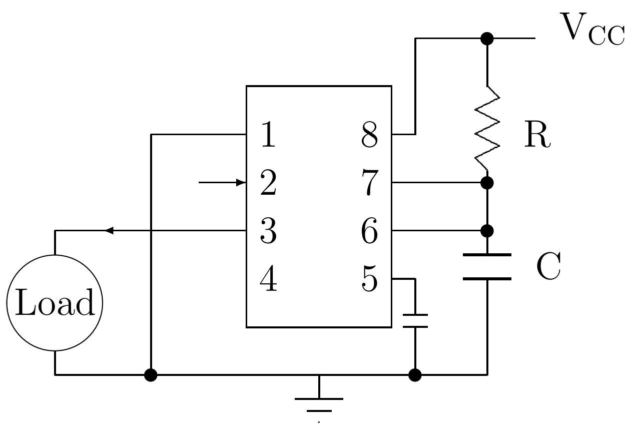
This “timer” is one of the first commercially popular analog integrated circuits, first produced by Signetics Corp. in 1971. It is still used today — in 2003, over 1 billion 555 timers were produced.
It is a simple circuit, basically consisting of two voltage comparators set to 1/3 and 2/3 of the supply voltage, a control flip-flop, and a power output stage.
The basic device has 8 pins, as follows:

555 timer

| | | | |
|---|---|---|---|
| Gnd | 1 | 8 | Vcc |
| Trigger | 2 | 7 | Discharge |
| Output | 3 | 6 | Threshold |
| Reset | 4 | 5 | Control |

The Trigger and Threshold inputs (2 and 6) are inputs to the upper and lower comparators, respectively.
The Discharge output is used to discharge the timing capacitor.
The Reset input brings the output low.
In typical operation, a pulse starts when the Trigger input is brought low, and its duration is controlled by a RC network.
When configured to generate a single pulse, the configuration is called “monostable” or a “one-shot.”

A typical "one-shot" configuration is the following:

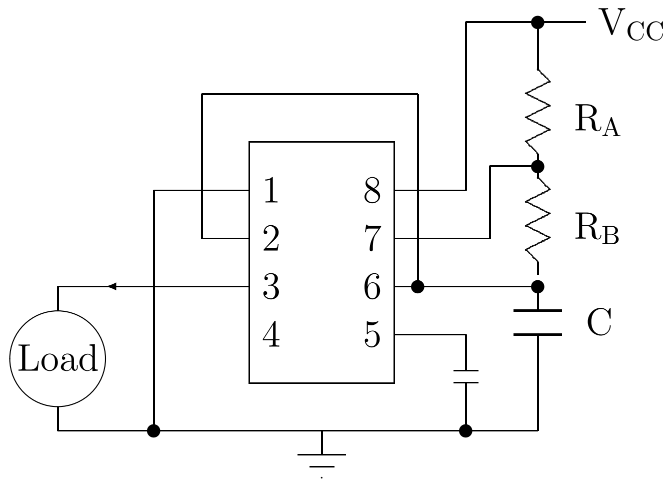


The duration of the output pulse is 1.1 × RC.
(The small capacitor is a decoupling capacitor, typically $0.01\mu F$.)

This device is capable of producing quite a range of pulses; e.g. if R is 1 M ohm and C is 1 $\mu$F, the pulse has a duration of about 1.1 seconds.
If R is 1K ohms, and C is 1 nF, then the pulse has a duration of 1 $\mu$s.

The device can also be set to trigger itself, by connecting pins 2 and 6, and adding another resistor. In this configuration, it is said to be "free running" or "astable."

Perhaps the most common configuration is the free running configuration:



In this mode, the capacitor charges and discharges between 1/3 and 2/3 of Vcc.

The charge time is $t_c = 0.693(R_A + R_B)C$

The discharge time is $t_d = 0.693(R_B)C$

The total period is $t_d = 0.693(R_A + 2R_B)C$

The frequency is $\frac{1.44}{(R_A+2R_B)C}$

The "duty cycle" D is $R_B/(R_A + 2R_B)$

Because of the linear relationship between the period and the resistance and capacitance, this kind of circuit (producing an output which can easily be counted) is a common component of a digital sensor.
Consider the following types of devices, all of which vary depending on some physical effect:

thermistor — resistance varies with temperature

photoresistor — resistance varies with light intensity

capacitive diaphragm — varies with pressure (sound as well)

capacitive channel — varies with flowing material

## Analog sensors

Not all of the things we might wish to sense have a convenient way to be converted to the digital domain. In other cases, it may be possible to increase the accuracy in the analog domain.

Consider two common parameters — temperature and pressure.

We can use a thermistor in the RC feedback of a timer to get a digital value, bit there are other devices with strong temperature effects as well. For example, the current flowing through a transistor junction is a function of temperature (this is what is used in the LM335, in fact).

The thermocouple effect, which produces a small voltage difference between the "hot" and cold" ends of a connection between two wires is often used to measure high temperatures (1000 C or so).

A fixed container of gaseous material could be used, and the pressure inside the container could be measured, also giving a measure of temperature.
Pressure can be measured digitally using capacitance — changing the separation of two conducting plates.
It can also be measured by other effects, including the piezoelectric effect, where a small voltage is generated due to forces acting on a crystal.
This effect can be used to measure large forces or pressures, and is commonly used in high pressure sensors.

Generally, analog sensors are devices in which the interaction between the sensor and the sensors energy domain produce an output directly in the electrical domain.
For example, piezoelectric sensors — output is voltage
photovoltaic sensors — output is current.
Sometimes, sensors are constructed by adding a material which interacts with a particular entity to be sensed. For example, some gas sensors have thin layers of material which interact (reversibly) with the entity being sensed, and change some property of the sensing element. (E.g., a gas sensor.)
Other sensors act by their effect on the material to be sensed. Smoke detectors use ionizing radiation from a small radioactive source to preferentially ionize combustion products, causing a small current to flow in the ionization chamber.

Many sensors are now constructed using micro electro-mechanical system (MEMS) technology. These devices are typically minature mechanical devices fabricated using integrated circuit technology. Perhaps the simplest of these is the accelerometer. It is essentially a miniature suspended cantilever beam with some type of (electronic) deflection sensing and circuitry.
These are used in the Nintendo WII controller.
Miniature cameras are used in a modern optical mouse.

## INTERFACING TO THE REAL WORLD - DAC's and ADC's

In the past few years, since digital devices have become computationally very fast, many functions previously performed by analog devices are now done digitally. This has allowed the implementation of much more accurate (and more complex) signal processing.
Generally, however, the signal must be digitized from an analog form before it is digitally processed, and then reconverted to analog form after the computation.
Therefore, devices to do these conversions (digital-to analog converters — DACs, and analog-to-digital converters — ADCs) have become important circuit elements.

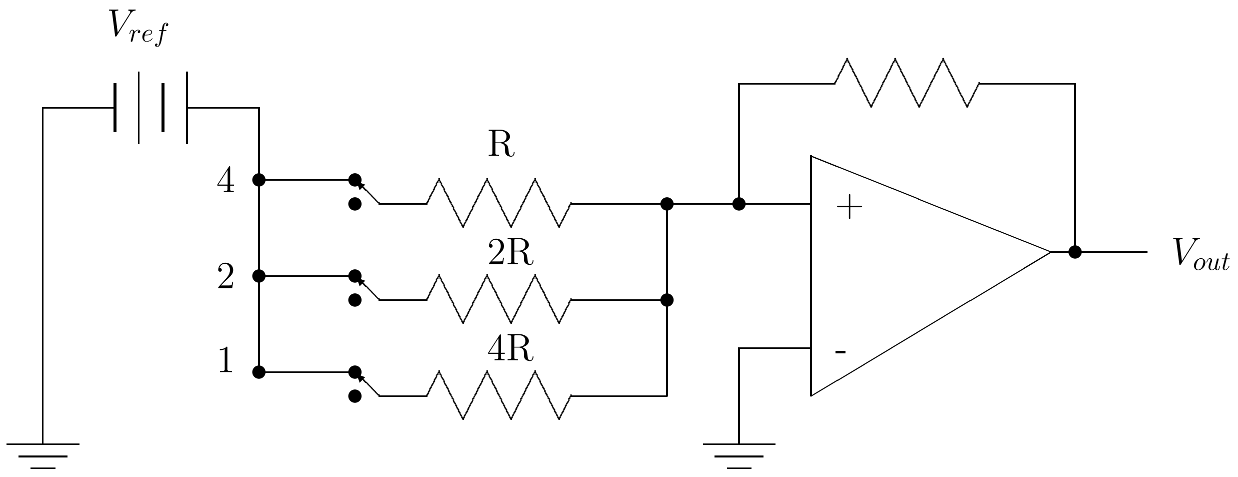
We will look at how these devices are implemented.

## Digital to analog converters

D-A converters are generally constructed in one of 3 ways: (all examples will show a 3-bit DAC, having 8 possible outputs, 0-7, with output corresponding to digital input 5 (binary 101).

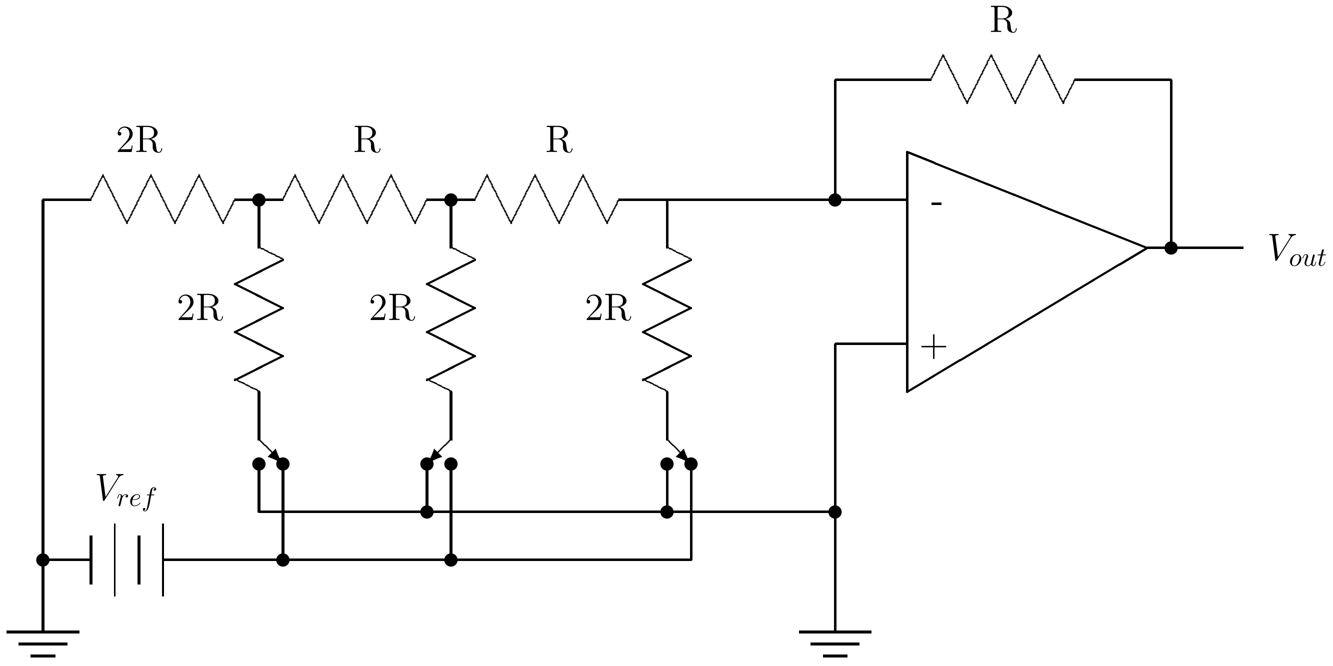
## Binary weighted summer

The first method employs a technique we have seen already — binary weighted resistors as input resistors to a summing op amp. In this case, each resistor must be a factor of two greater than the previous. It is rather difficult to make resistors which vary over more than about an order of magnitude with great accuracy on an IC. However, this method requires the least number of resistors, requiring only N resistors for an N bit D–A converter. A circuit diagram for a typical DAC using this technique is shown in the following diagram:



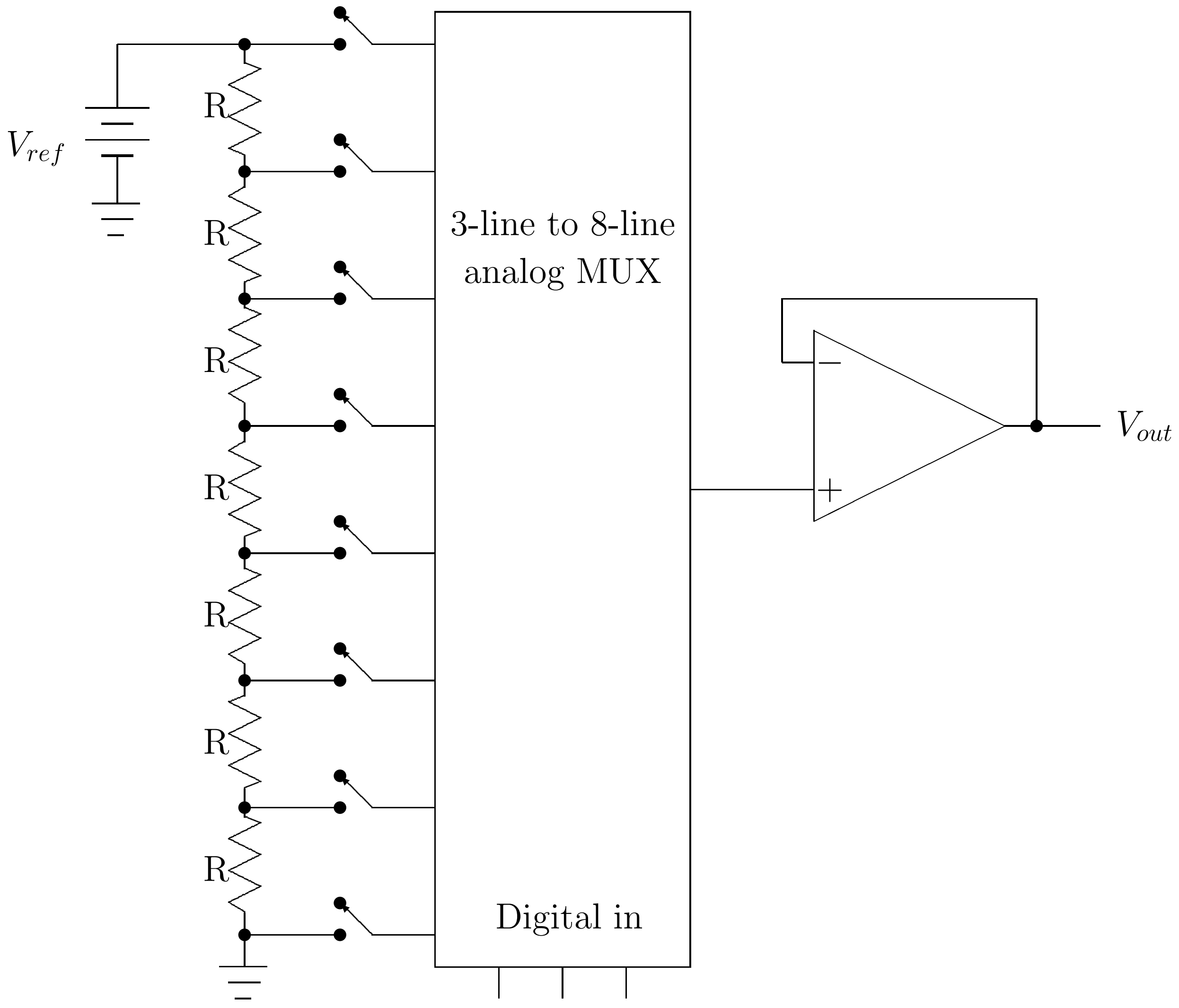
## The R – 2R ladder

This circuit requires only two values of resistance, with value R and 2R, as shown in the following diagram.
It requires 2N resistors, where N is the number of bits in the D–A converter. The resistors must be quite accurate, however. The operation of this D–A converter is more difficult to understand than the previous, but it can be understood readily by noting that at each intersection of 3 resistors the current is divided into two equal parts. This is perhaps most easily seen if the centre leg of the ladder is connected to the reference, and all others are connected to ground; it can then readily be seen that at each junction the current is divided into two.



## $2^N$ **resistor string**

The third method employs a string of $2^N$ identical resistors, where N is the number of bits in the D to A converter, as shown in the following diagram.
Here the number of resistors is large, but they need not be extremely accurate. This device has the further advantage that the output is certain to be monotonic.



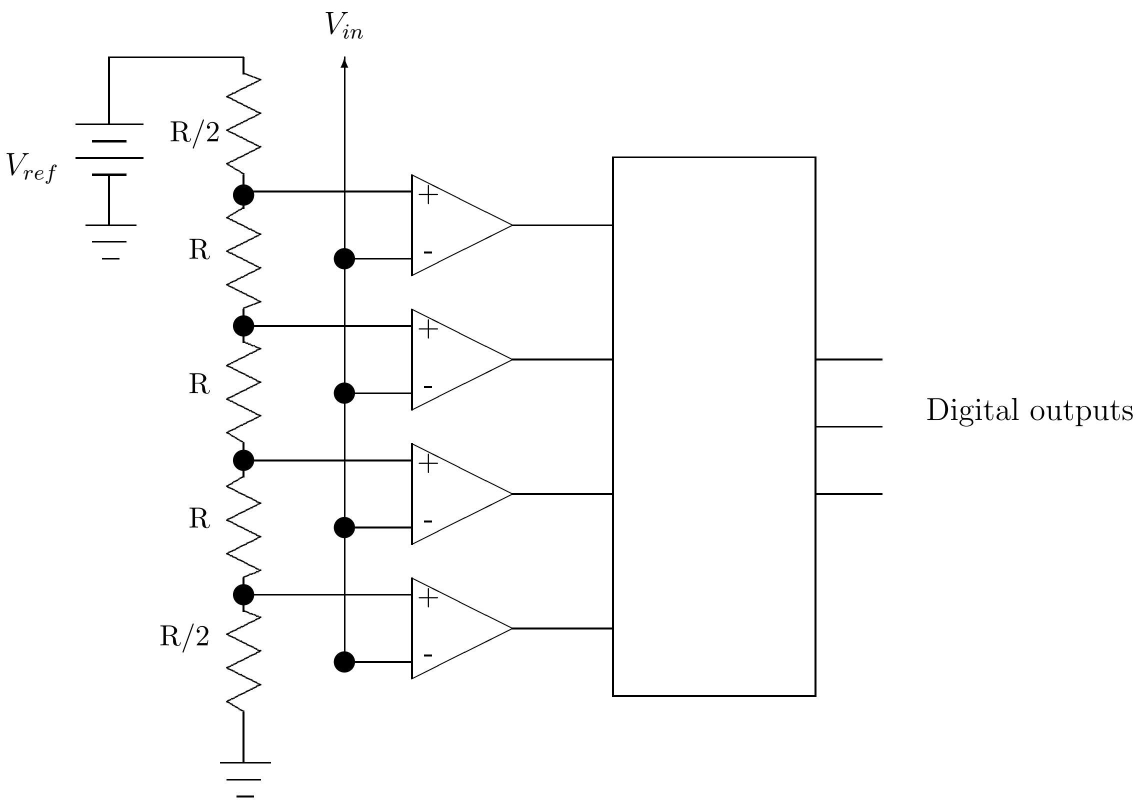
DAC's have either an internal (fixed) reference voltage $V_{ref}$, or allow an external reference to be used. If an external reference is used, the DAC is called a multiplying DAC, or MDAC. Its output is the product of $V_{ref}$ and the binary code which is input.

Note that in the last example, if the analog MUX is bi-directional, the MDAC can be used as a programmable resistor.
In fact, this circuit is commercially available. In fact, it is often packaged as a programmable device with a serial input.

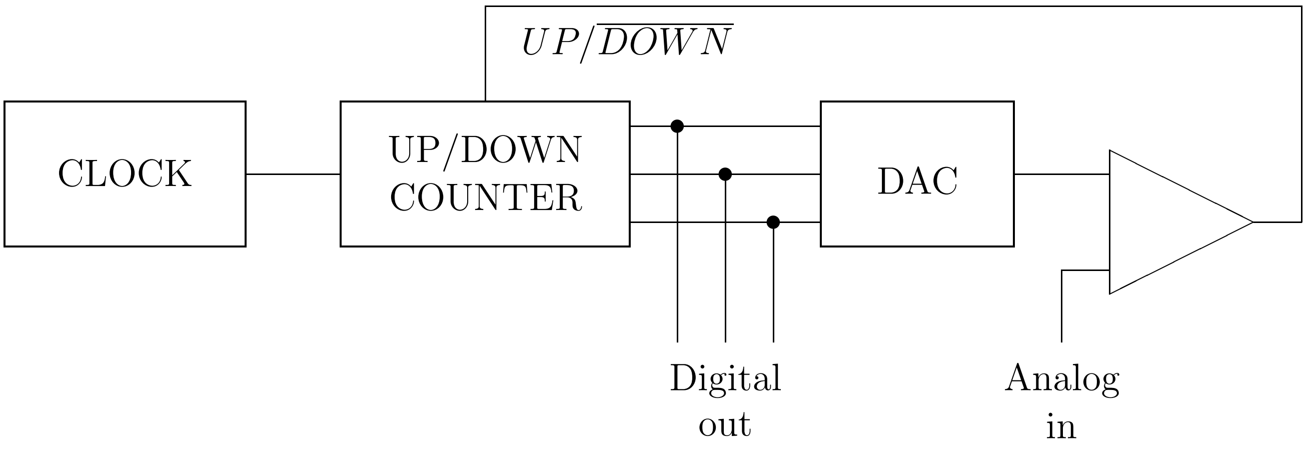
## Analog to digital converters

There are 3 main methods.

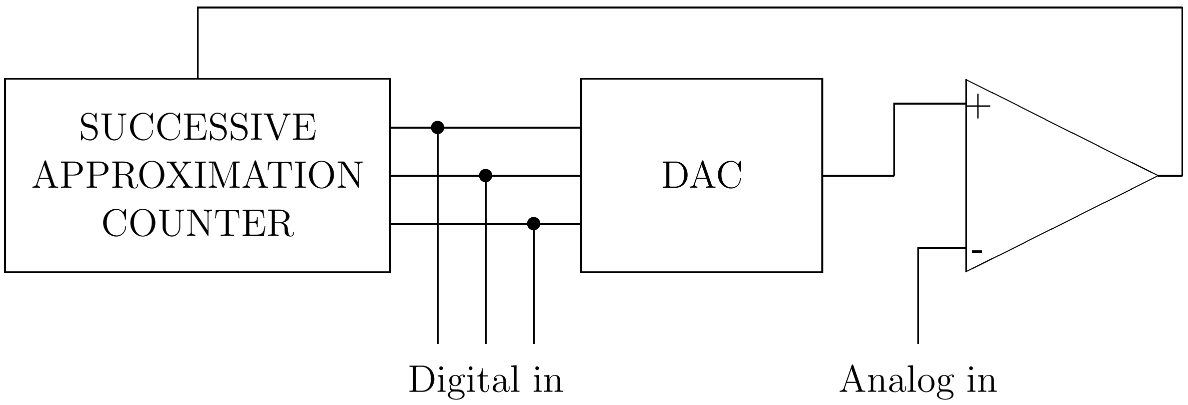
1. Flash conversion — extremely fast. Uses a set of $2^n$ resistors and $2^n$ comparators.



2. Using a DAC and a counter, and a single comparator.



3. Using a DAC and a successive approximation counter.

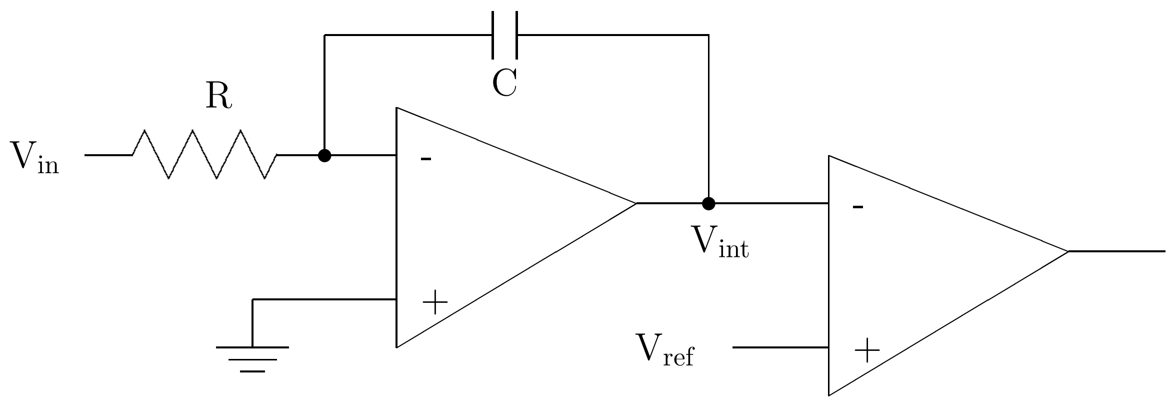


try 100, then either 010 or 100, *etc. i.e.*, perform a binary search for the correct number.

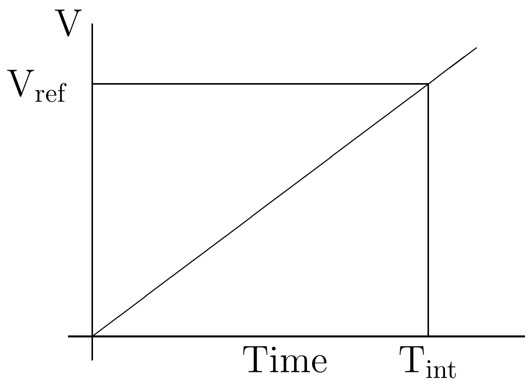
4. Integrating ADCs — again, there are two main methods.

## Single slope ADCs

A very simple, accurate, and effective ADC can be constructed using an integrator and timer. Basically, the input is integrated, and the time required to charge the integrating capacitor to a reference voltage is measured.



For a steady input voltage, the voltage from the integrator (Vint) increases linearly:

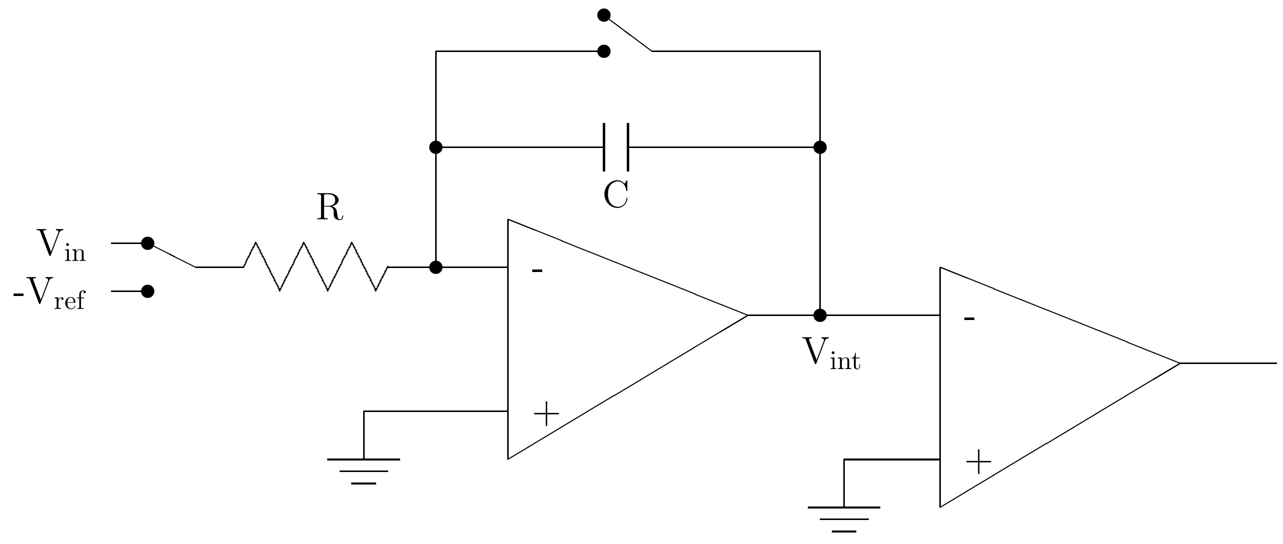


This works reasonably well for slowly varying signals, but its accuracy depends on the tolerances of R and C.

Also, the time for a conversion depends on the input voltage. In particular, for Vin = 0, the time is infinite.

## Dual slope ADCs

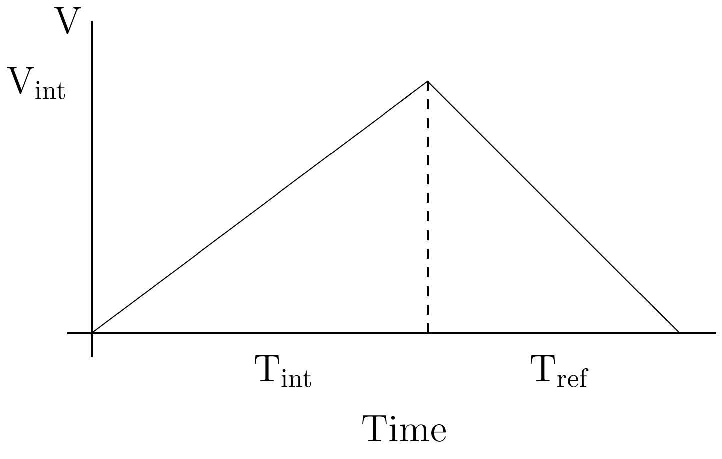
It is possible to eliminate the drawbacks of the previous ADC, by measuring both the charge and discharge time of the integrating capacitor. Effectively, a negative reference voltage is integrated with the positive unknown integral, and the relative times for the positive and negative integration give the required voltage.



This is a three stage process:

1. The capacitor is discharged through switch S.
2. $V_{in}$ is integrated for some fixed time, $T_{int}$.
3. $V_{ref}$ is integrated until the output is 0, in time $T_{ref}$.

The ratio $V_{in}/V_{ref}$ is the ratio of the integration time $T_{int}$ to $T_{ref}$.



Here, $V_{in} = V_{ref} \times T_{int}/T_{ref}$.

Recall that $T_{int}$ is fixed (constant), so the output is a function of $T_{ref}$ only.

This effectively eliminates the effect of variations in R and C, and also solves the problem of a variable integration time.

Integrating ADCs have another interesting property — they tend to average out noise. In particular, periodic noise (e.g., 60 cycle noise from external power) can be averaged out by choosing the $T_{int}$ as some multiple of a 60 cycle period.

Integrating ADCs can have more than 16 bit accuracy. In fact, they are used in many digital voltmeters because of their accuracy and noise reducing properties.
They are not very fast, however.