Unit 5. Study of a real computer: Atmega328P microcontroller

Computer Structure
E.T.S.I. Informática
Universidad de Sevilla

Jorge Juan-Chico <jjchico@dte.us.es> 2021

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Some images and other objects are used here according to their license. Follow the reference to the original work for details. Images not referenced are either original work of the author or are in the public domain.

Contents

Contents

- AVR assembly programming
  - Requierements
  - AVR instruction set
  - Toolchains, platforms, frameworks and IDE’s
  - The GNU AVR toolchain
  - C pre-processor macros
  - GNU assembler directives and expressions
  - Assembling, uploading and testing
  - AVR programming (uploading) alternatives
  - Debugging programs
  - AVR IDEs and PlatformIO
  - Assembly subroutines
  - Reading and writing program memory
  - Working with data other than 8 bits

Contents

- AVR C programming
  - C program example
  - Code optimization
  - Compiling and disassembling
  - C language functions
  - Wait: why learning assembly?
  - Where and why is assembly used?
  - AVR C calling convention
  - Calling assembly subroutines from C
  - Calling C functions from assembly.

Contents

- Interrupts
  - Interrupt vectors
- Timers
  - AVR timers
  - Timer/Counter 1
  - Example: blink with interrupts.
- Other peripherals
  - Analog inputs
  - SPI
  - TWI (I2C)

Objectives

- Understand the structure of a real computer.
- Understand typical peripherals present in real coputers.
- Understand the importance of small computers in today's computer technology (from sensor networks to the IoT).
- Learn to program a computer at the lowest level (assembly).
- Understand the low-level programming conventions.
- Learn the basic tools involved in computer programming.
- Understand the role of programming frameworks and Integrated Development Environments (IDE's).

Bibliography

- ATmega48A/PA/88A/PA/168A/PA/328/P Data Sheet – Technical information about ATmegaX8 chips. Use as reference.
- AVR Instruction Set Manual – Detailed description of AVR assembly instructions. Use as reference.
- AVR microcontroller programming with avr-libc and the GNU toolchain
  – AKA the “avr-bare” repo. Must read.
- AVR Libc Home Page and documentation – Definitions for C and assembly programming. Use as reference.
- AVR Libc source code – Use as reference.

What is a microcontroller? (or µC or MCU)

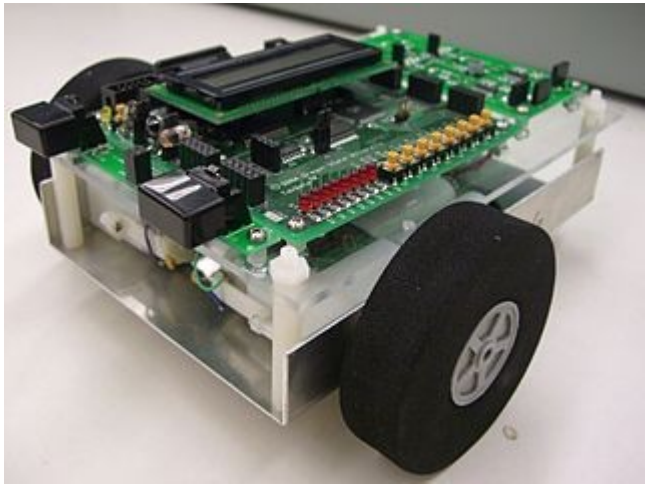
- It is a System on Chip that includes a simple CPU core, a program and data memory, and a variety of input/output peripherals that make them suitable to be applied to a wide range of software-controlled applications.
- Typical peripherals:
  - Generic input and output digital ports (GPIO).
  - Analog inputs and/or outputs.
  - Serial interfaces (SPI, USART, etc.).
  - Timers.
  - Clock generators.

MCU → MicroController Unit

What is a microcontroller?

- Typical characteristics
  - Low cost.
  - Robust.
  - Small size.
  - Low power consumption.
  - Low processing power.
  - Harvard architecture.
  - Non-volatile program memory (EEPROM/Flash).
- Typical applications.
  - Embedded systems in general.
  - Sensor networks.
  - Robotics.
  - Control systems: small appliances, automotive, remotes, heating, etc.

Typical applications



Microcontrollers drive the IoT revolution.

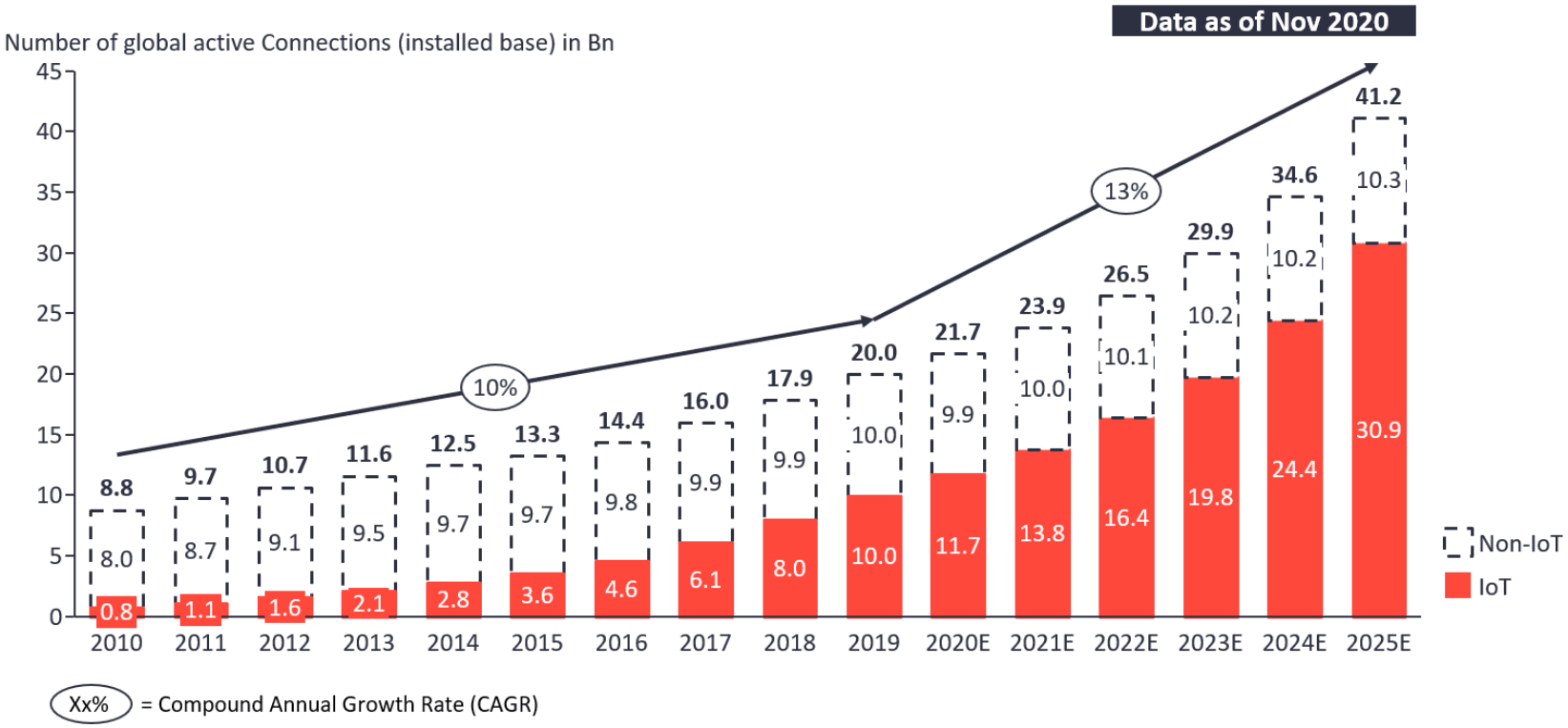
IoT devices forecast

IOT ANALYTICS

Insights that empower you to understand IoT markets

Total number of device connections (incl. Non-IoT)

20.0Bn in 2019– expected to grow 13% to 41.2Bn in 2025



Note: Non-IoT includes all mobile phones, tablets, PCs, laptops, and fixed line phones. IoT includes all consumer and B2B devices connected – see IoT break-down for further details

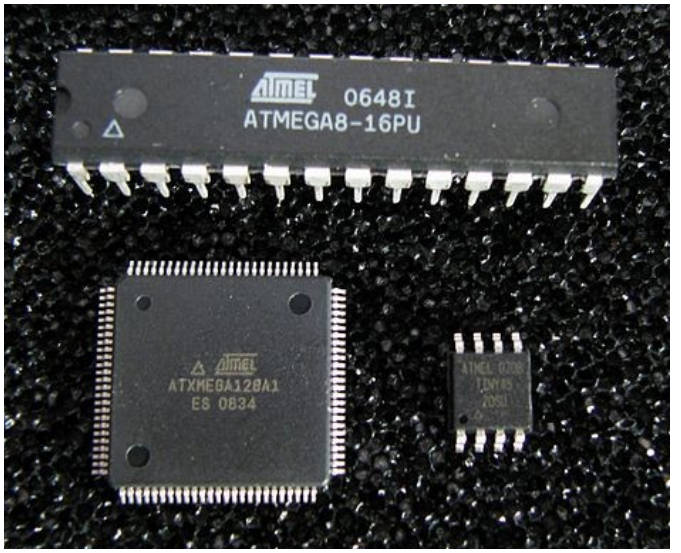
Source(s): IoT Analytics - Cellular IoT & LPWA Connectivity Market Tracker 2010-25

https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/

The Atmel AVR microcontroller family

- Created by Norwegian students Alf-Egil Bogen and Vegard Wollan.
- Developed by Atmel since 1997, sold to Microchip in 2016
- One of the most successful microcontrollers ever.
- Supported by Free Software tools and an open community (AVRFreaks).
- Main series
  - ATtiny: low-end devices with fewer pins and peripherals.
  - **ATmega**: mid-range devices with complete set of peripherals.
  - ATxmega: high-end devices with increased performance and set of peripherals.

History of the AVR



The ATmega328/P and fiends (48, 88, and 168)

- 8-bit RISC architecture.
- Clock frequency up to 20MHz @4.5-5.5V
- Up to 20MIPS
  - Two stages pipeline.
  - Many instructions execute in 1 clock cycle.
- Flash program memory.
- SRAM and EEPROM data memory.

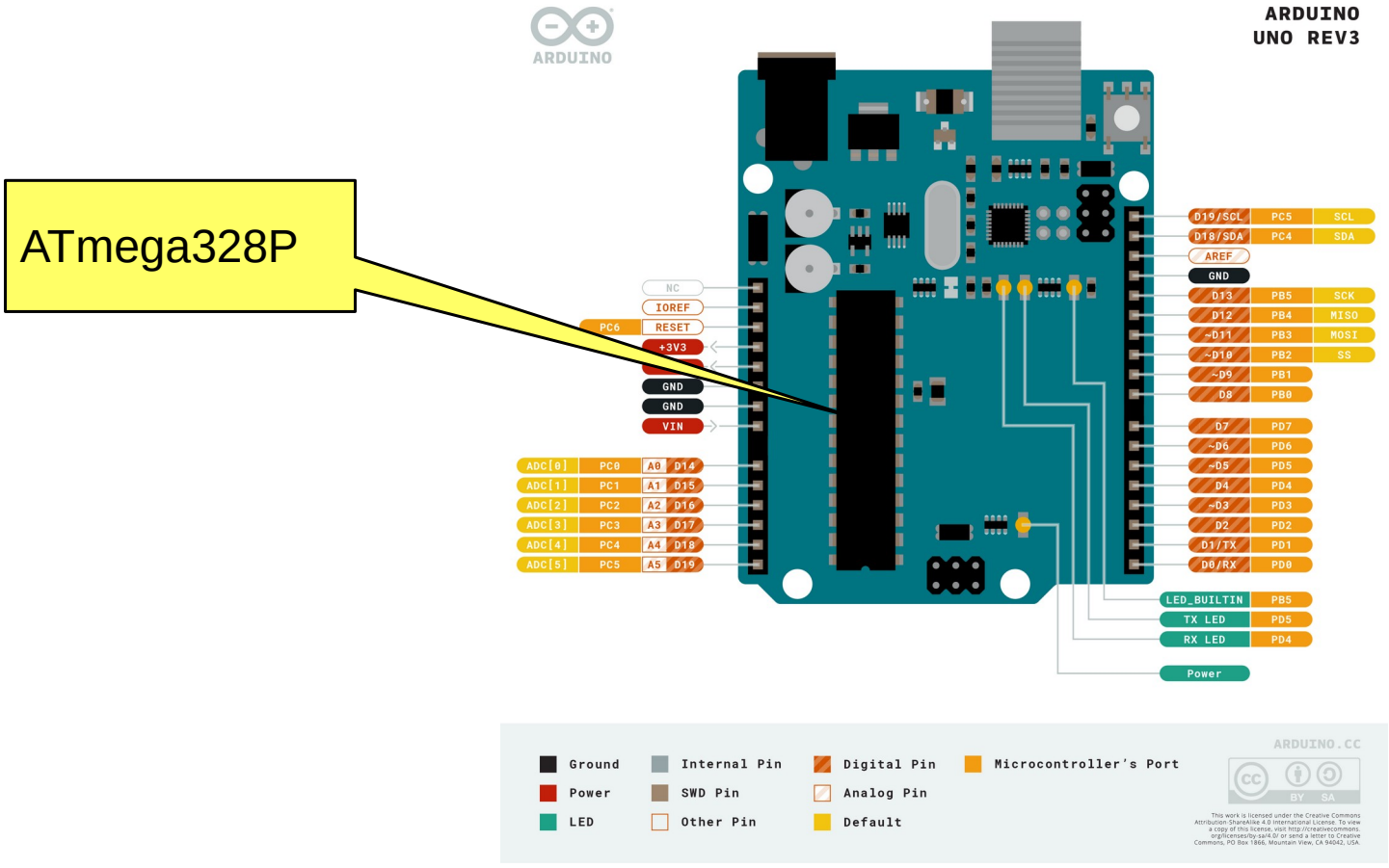
ATmegaX8 data sheet

653 pages.
Not to read from tip to toe.
Use it as reference.

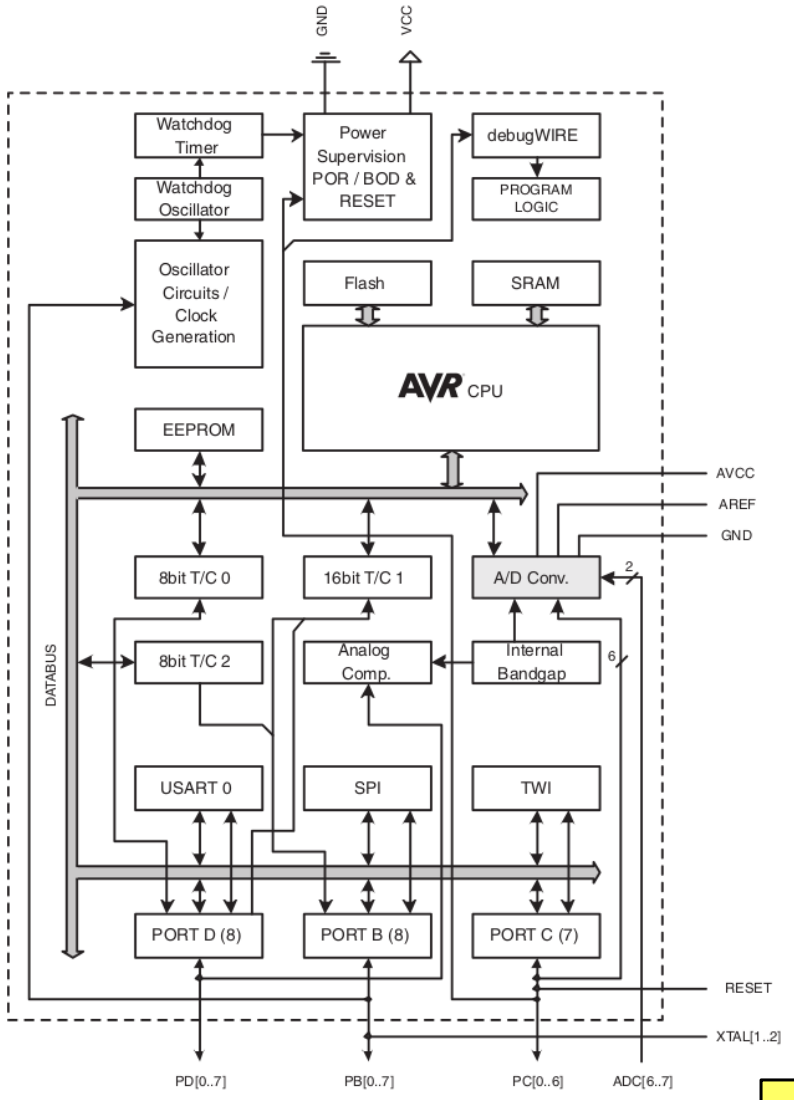
Device	Flash	EEPROM	RAM	Int. vector size
ATmega48A/PA	4KB	256B	512B	1 inst./vector
ATmega88A/PA	8KB	512B	1KB	1 inst./vector
ATmega168A/PA	16KB	512B	1KB	2 inst./vector
ATmega328/P	32KB	1KB	2KB	2 inst./vector

ATmega328P and the Arduino UNO

- The core of the Arduino UNO board.
- Arduino is Framework to build control systems easily (more of this later in the unit).



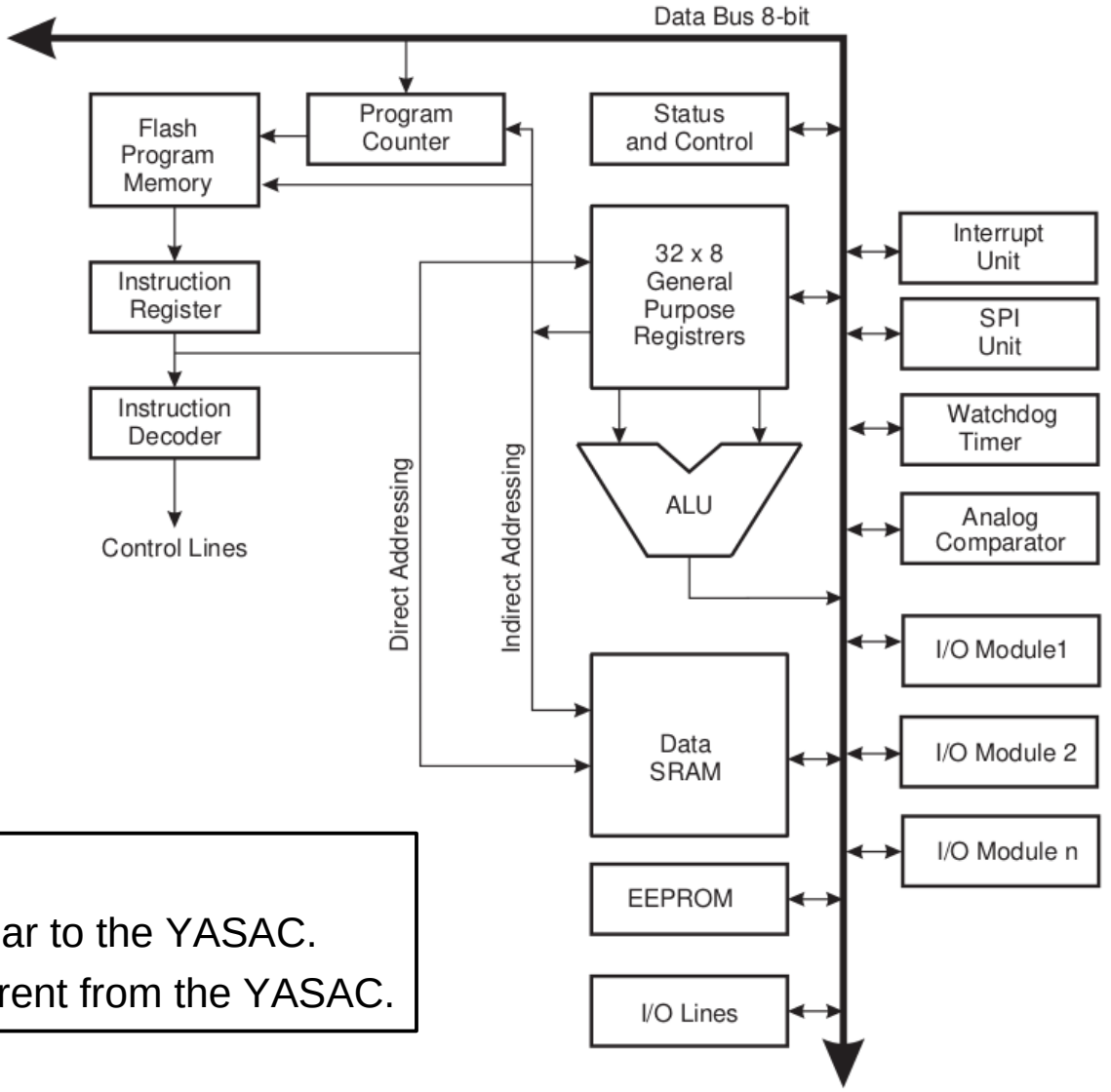
ATmegaX8 block diagram



- AVR CPU core.
- Supporting circuits
  - Clock generator.
  - Watchdog timer.
  - Debug unit.
  - Power management unit.
- Peripherals
  - PORTB, PORTC, PORTD: digital input/output ports.
  - SPI, USART0, TWI: serial interfaces.
  - T/C 0, 1, 2: timer/counters.
  - A/D: analog to digital converters.

Which of those did we have in the YASAC system?

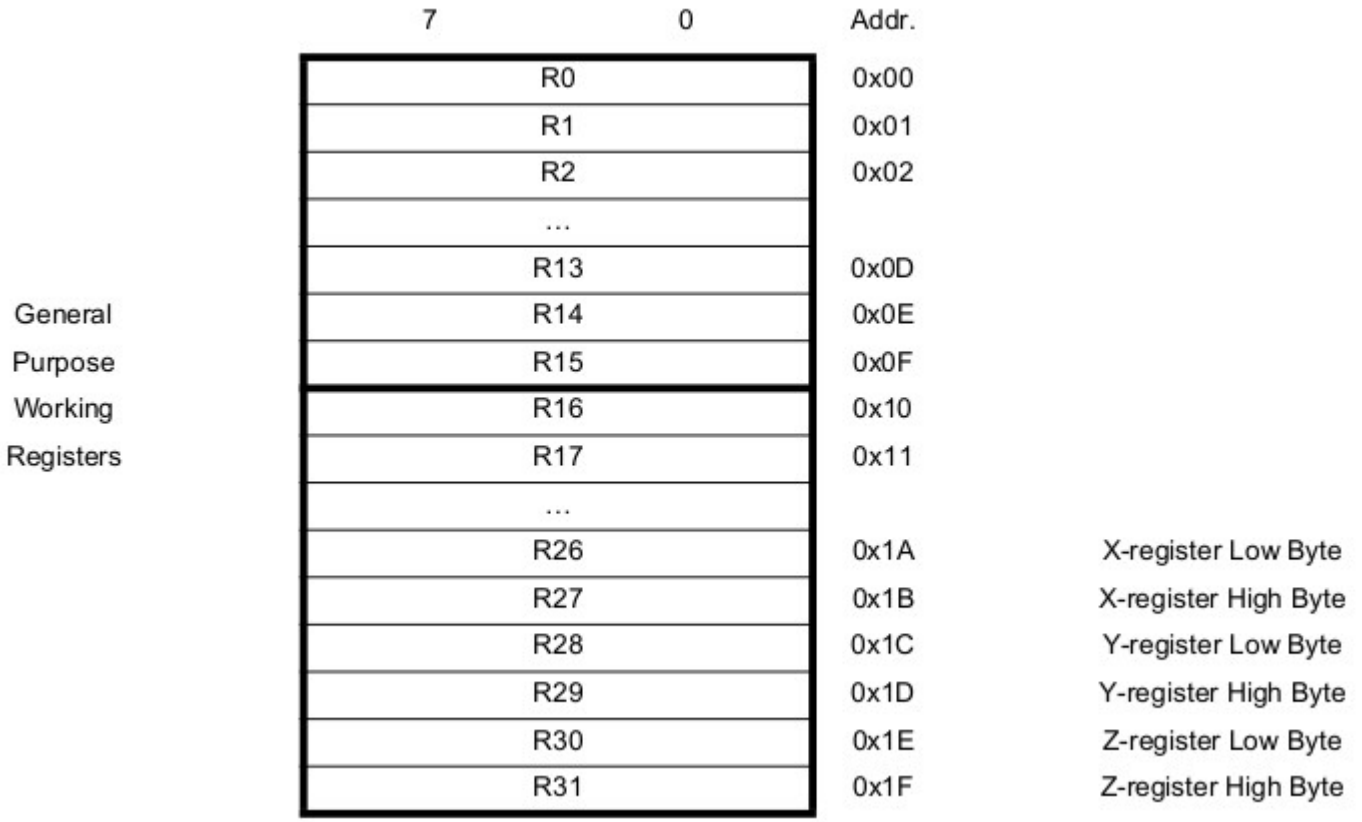
The AVR CPU core



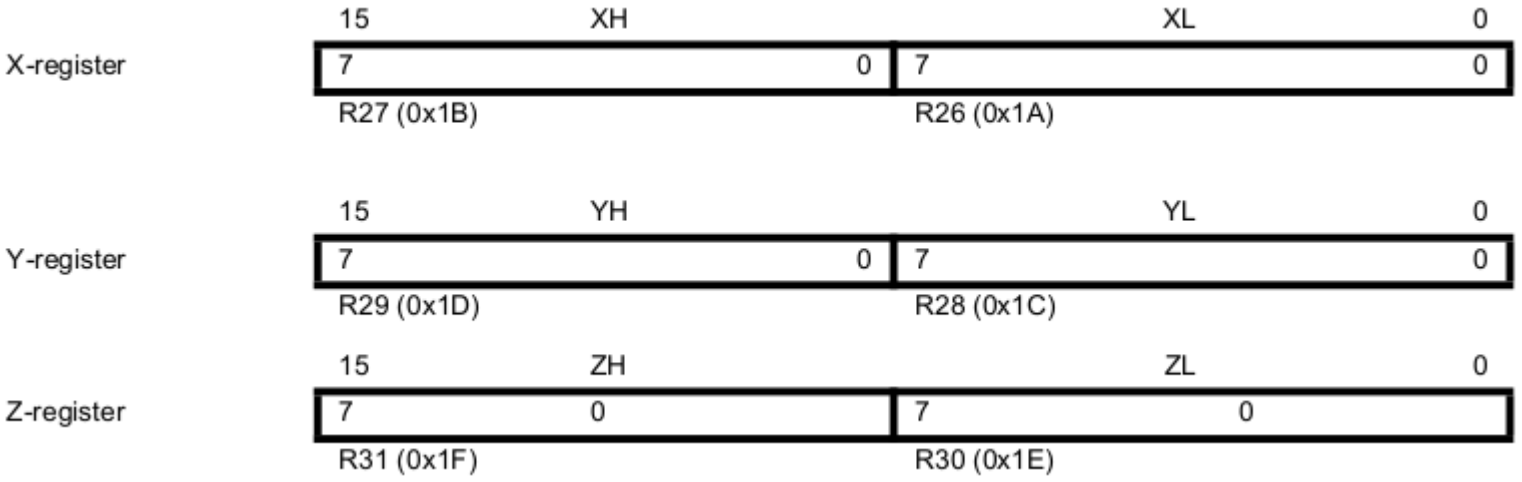
Quick exercise:

1) Find 3 things that are similar to the YASAC.
2) Find 3 things that are different from the YASAC.

General purpose registers



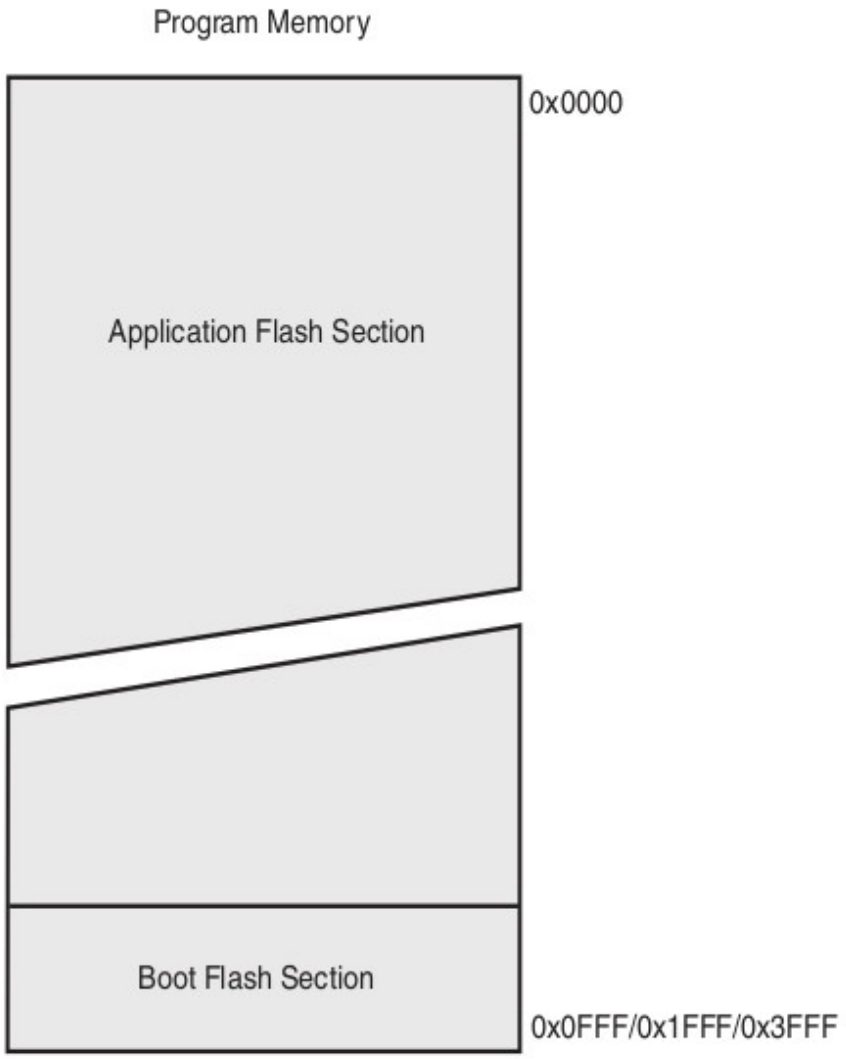
Pointer registers



Stack and stack pointer

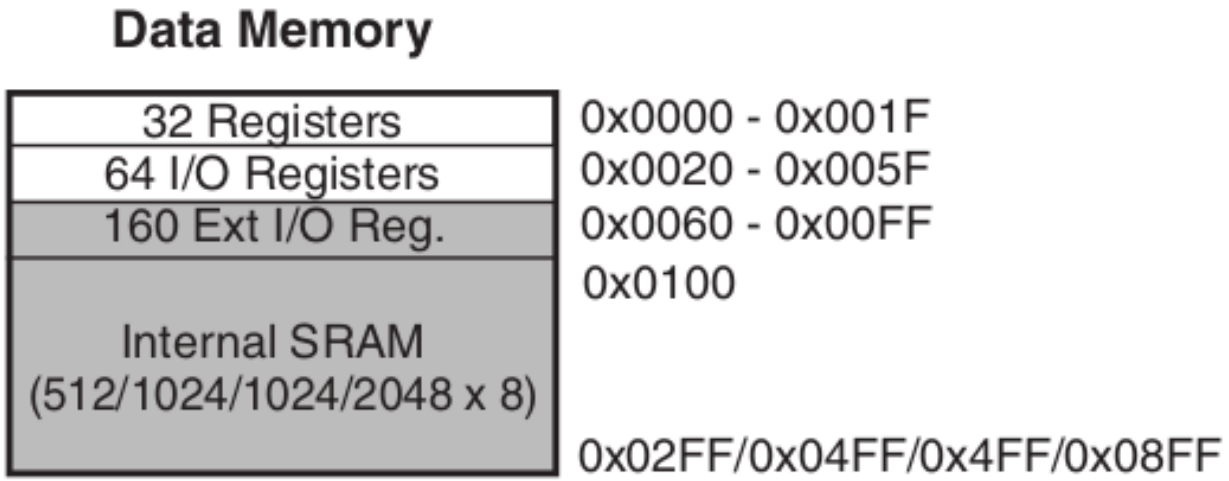
Bit	15	14	13	12	11	10	9	8	
0x3E (0x5E)	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
0x3D (0x5D)	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	
	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	

Program memory



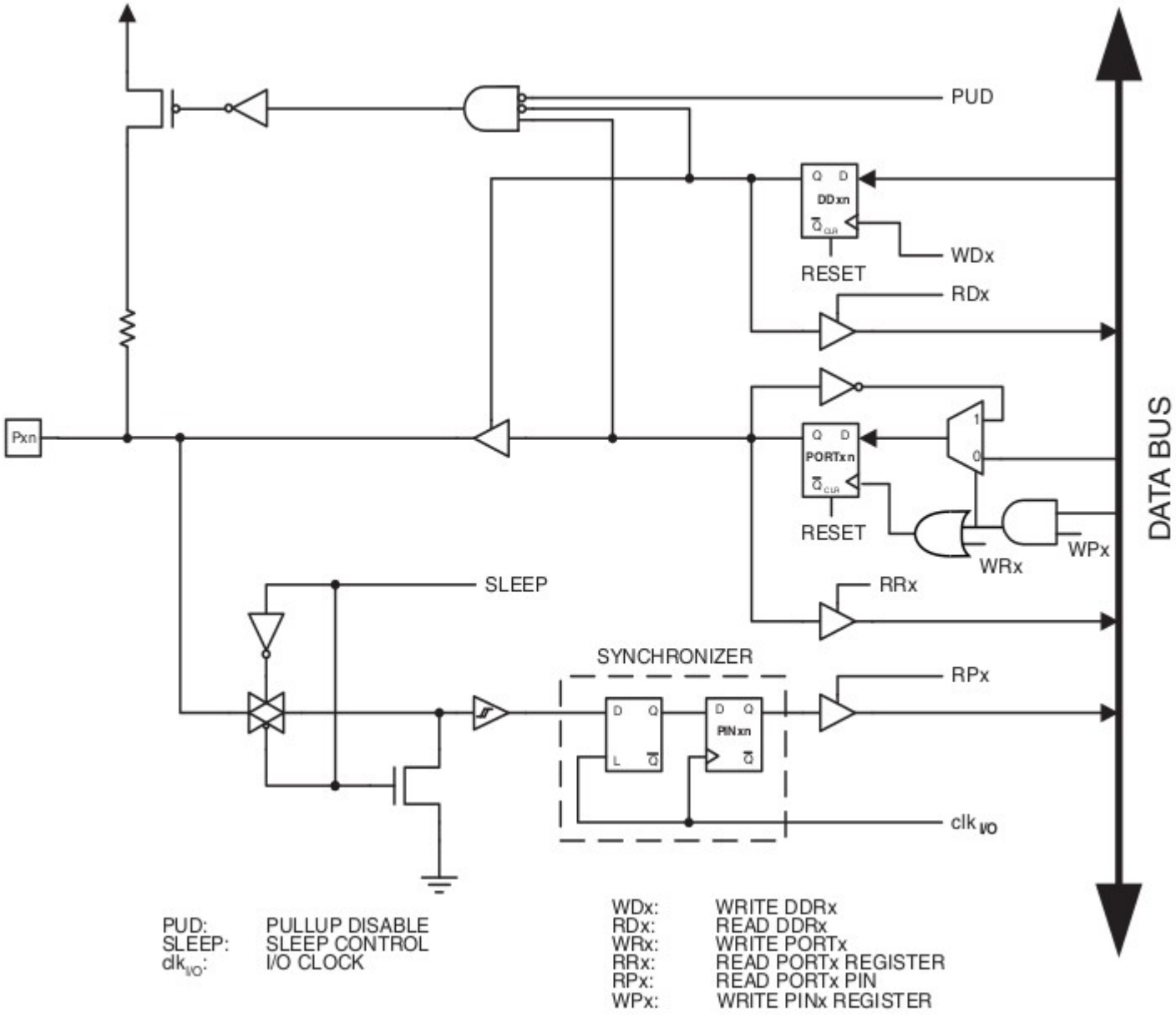
- 16-bit wide.
- All instruction codes are 16 bit.
  - Some instructions use an additional 16-bit immediate value in the next position.
- Also byte-addressable by program memory load and store instructions (will see later).
- ATmega328P memory is:
  - 16k x 16 (word addressable)
  - 32k x 8 (byte addressable)

SRAM data memory



- Byte addressable.
- All general-purpose registers and I/O registers are mapped to data memory.
  - First 64 I/O registers can also be accessed by specific input/output instructions (faster).
- Real usable data memory starts at 0x100
  - ATmega328P data memory goes from 0x100 to 0x8FF (2kB).

Digital I/O ports



Digital I/O ports

DDxn	PORTxn	PUD (in MCUCR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

Writing a port takes one cycle after the “out” instruction:
wait a cycle (nop) before reading (in) from the port.

Digital I/O ports

PORTB – The Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

DDRB – The Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PINB – The Port B Input Pins Address(1)

Bit	7	6	5	4	3	2	1	0	
0x03 (0x23)	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Note: 1. Writing to the pin register provides toggle functionality for IO

Digital I/O ports

PORTC

PORTC – The Port C Data Register

Bit	7	6	5	4	3	2	1	0	
0x08 (0x28)	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	PORTC
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

DDRC – The Port C Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x07 (0x27)	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	DDRC
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

PINC – The Port C Input Pins Address(1)

Bit	7	6	5	4	3	2	1	0	
0x06 (0x26)	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	PINC
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

PORTD

PORTD – The Port D Data Register

Bit	7	6	5	4	3	2	1	0	
0x0B (0x2B)	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	PORTD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

DDRD – The Port D Data Direction Register

Bit	7	6	5	4	3	2	1	0	
0x0A (0x2A)	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	DDRD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

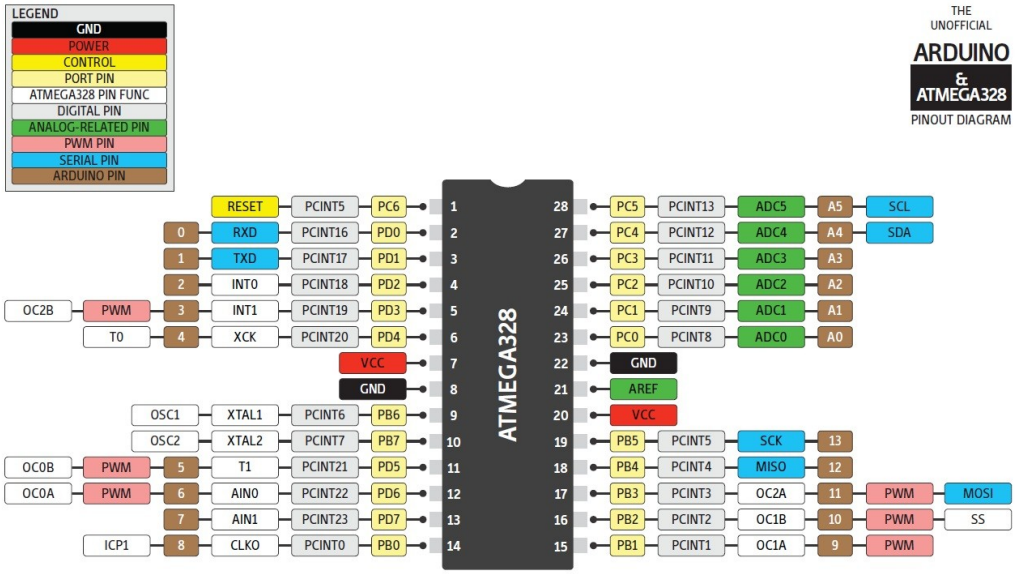
PIND – The Port D Input Pins Address(1)

Bit	7	6	5	4	3	2	1	0	
0x09 (0x29)	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	PIND
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

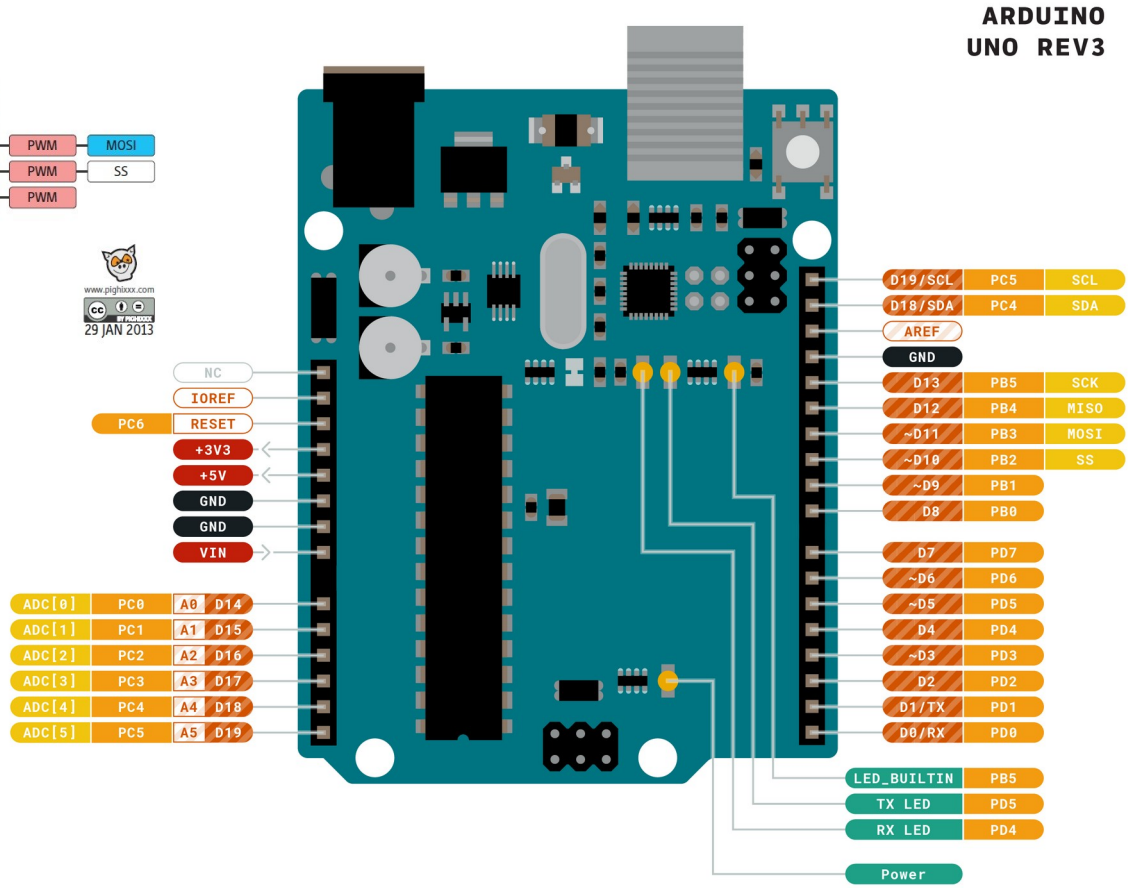
Development boards

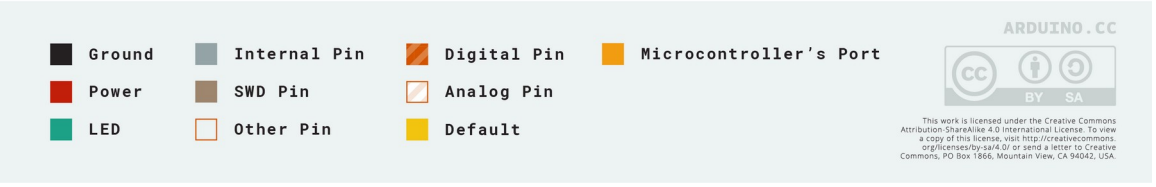
- Microcontrollers are autonomous
  - Only need a supply voltage and a connection to a programmer to be used.
- Development boards simplify doing projects with microcontrollers by including additional circuitry and devices:
  - Supply voltage connections and regulation.
  - Easy to use connection pins.
  - Easy to use programming pins.
  - Simple peripherals: LED's, switches, push buttons, etc.
  - Programming interface (e.g. USB): no need of an external programmer.
  - BUT: not all pins of the MCU may be available, some MCU resources may be used by the board's hardware or software, etc.
- Arduino boards are the most popular boards built around the AVR platform.

Arduino UNO



Arduino UNO specs





AVR programming

- AVR microcontrollers are programmed in AVR assembly. There are tools that translate assembly code to machine code (assemblers).
- Most AVR microcontrollers can also be programmed in C and C++. There are programs that translate C/C++ to machine code (compilers) and a library of standard C functions.
- All YASAC assembly instructions can be used with AVR's (we copied from them :)
  - AVR assembly has many more instructions.
  - AVR instructions have different restrictions.
  - AVR instruction format is different (not binary compatible).
  - AVR peripherals are different.

What do you need to write assembly programs (AVR or other)?

- Knowledge of the processor architecture.
- An instruction set summary (included with the data sheet).
- An instruction set manual, with the detailed instructions operation, in case the summary is not enough (see the bibliography).
- The basic structure of an assembly program (segments).
- A summary of the assembler directives.
- The pin-out and characteristics of the board you are using (if any).
- How to use the tools to produce and upload an executable program.

You do not need to know all the details from the beginning.
Start with a basic set and learn more as you need.

AVR microcontroller programming with avr-libc and the GNU toolchain

AVR instruction set overview

- Arithmetic and logic instructions.
- Branch instructions.
- Bit and bit-test instructions.
- Data transfer instructions.
- MCU control instructions.

Details are in the ATmega328P Data Sheet (instructions summary) and in the AVR Instruction Set Manual.

Arithmetic and logic instructions

- YASAC-like:
  - ADD, SUB, AND, OR, EOR.
- Immediate versions:
  - SUBI, ANDI, ORI (but not ADDI!).
  - Immediate addressing can only be used with R16 to R31!
- 16-bit instructions:
  - ADIW, SBIW.
- Multiply instructions:
  - MUL, MULS, MULSU.
- Increment and decrement instructions:
  - INC, DEC.
- Register clear and set instructions:
  - CLR, SER

Branch instructions

- YASAC-like:
  - BRBS, BRBC, and derivatives.
- Relative jump and call:
  - RJMP, RCALL.
  - Faster and uses less program memory.
- Indirect call (to address in Z):
  - ICALL.
- Skip and bit test instructions: easy decisions on single bits.
  - Compare and skip if equal: CPSE.
  - Skip if bit in register is cleared/set: SBRC/SBRS
  - Skip if bit in I/O register is cleared/set: SBIC/SBIS
- Return from interrupt:
  - RETI.

Bit and bit-test instructions

- Status register bit change:
  - BSET, BCLR and derivatives (like in YASAC).
- Logic and arithmetic shift, in addition to rotation:
  - Logic shift left/right: LSL, LSR
  - Arithmetic shift right: ASR
  - Rotate left/right: ROL, ROR (like in YASAC).
- Set and clear bits in I/O registers:
  - SBI, CBI.
  - Easy single bit I/O manipulation.
- Swap nibbles:
  - SWAP
  - Useful to handle BCD numbers.

Data transfer instructions

- YASAC-like:
  - MOV, LDI, LD, ST, LDS, STS, PUSH, POP.
  - LDI only for registers R16 to R31!
  - Register indirect addressing (LD, ST) only with pointer registers X, Y, Z.
- Post-increment and pre-decrement indirect load and stores
  - With LD and ST.
- Indirect loads and stores with displacement:
  - LDD, STD.
- Program memory load and store:
  - Only indirect through Z pointer: LPM, SPM.
  - LPM has optional post-increment.
  - Note: Z holds “byte” addresses, not instruction word addresses.
- I/O port input and output instructions:
  - IN, OUT
  - Complemented with SBIC, SBIS, CBI, SBI.

MCU control instructions

- No operation:
  - NOP
  - Useful to insert delay:
    - Finish and OUT instruction.
    - Implement delay loops
- Take the processor to bed:
  - SLEEP
  - Put the processor in an sleep (low power consumption) mode.
  - Necessary for battery-powered applications.
  - Combined with interrupts.
  - Needs previous interrupt and sleep mode configuration.

Toolchains, platforms, frameworks and IDE's

- Toolchain
  - Set of software tools that automate the process of converting source code in executable programs for some computer processor or family: compilers, assemblers, linkers, dissemblers, etc.
  - Every tool have some specific tasks and one is normally executed after another in an specific order (the "chain" in "toolchain").
- Platform
  - The toolchain extended with additional supporting tools and libraries together with the hardware it supports. E.g. "Developing for the AVR platform".
- Framework
  - Set of software and hardware (board) resources for a given platform that simplifies development by providing solutions to common problems: programming libraries, program templates, board configuration, etc. E.g. the Arduino Framework.
- Integrated Development Environment (IDE)
  - Software tool that works as user interface to the underlying framework, platform and toolchain in order to make software and/or hardware development easier.
  - Tasks: code editing, project management, tool invocation, debugger interface, etc.
  - Typically built around some advanced code editor.

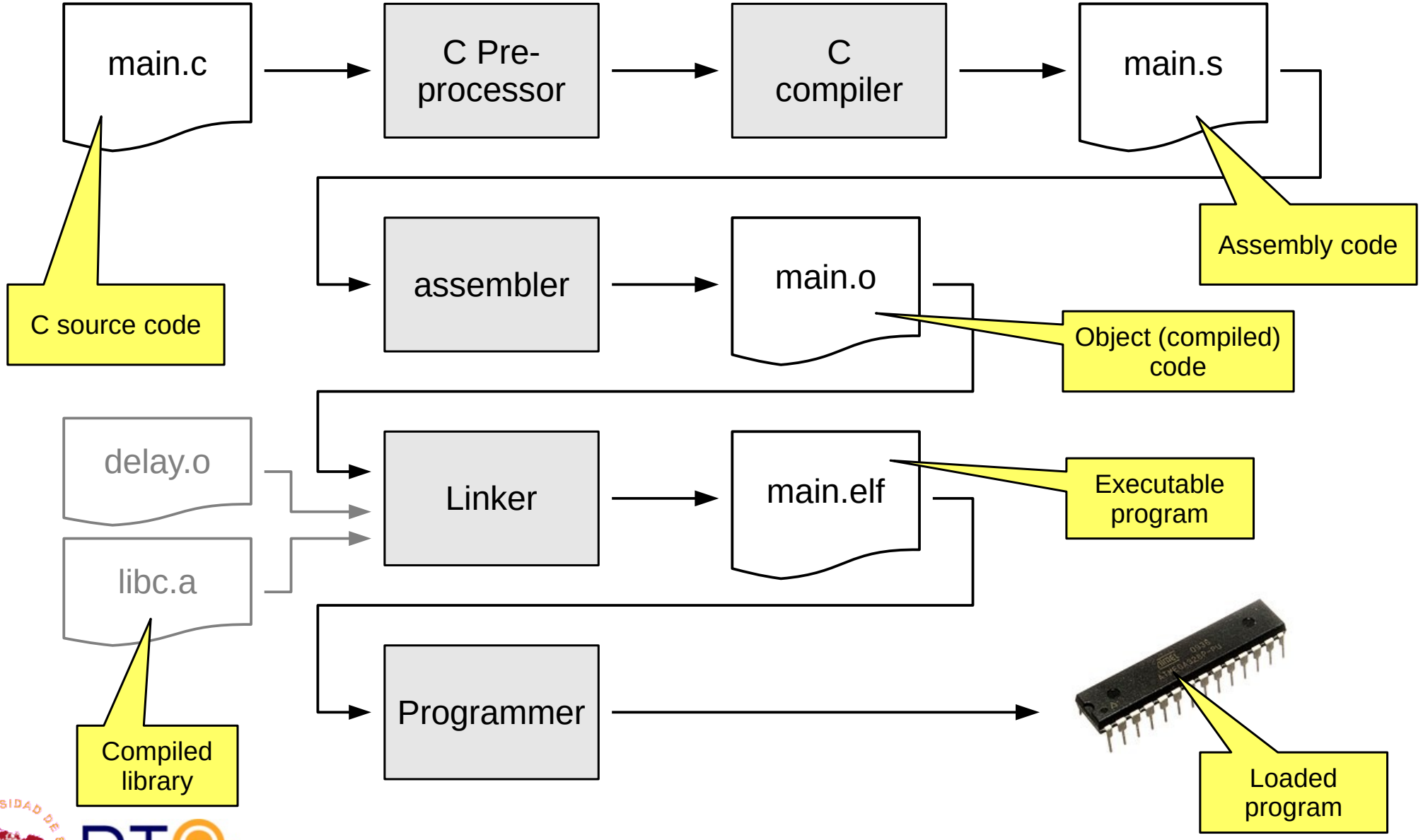
AVR toolchains

- Atmel's AVR assembler
  - Free licensed assembler.
  - Supports basic assembly programm
- AVR GNU toolchain
  - The toolchain that drives GNU/Linux and thousands of Free Software projects around the world (including Arduino).
  - Complete set of tools to program AVR MCU's in assembly, C and C++.
    - Compiler, assembler, linker, etc.
  - Complete programming platform together with external tools:
    - Programmer (avrdude), in-hardware debugger (avarice), simulators (simavr), etc.
  - Most AVR projects use this.
- MPLAB XC compiler
  - Non-free C compiler from Microchip.
  - Supports AVR MCU's since Microchip acquired Atmel.

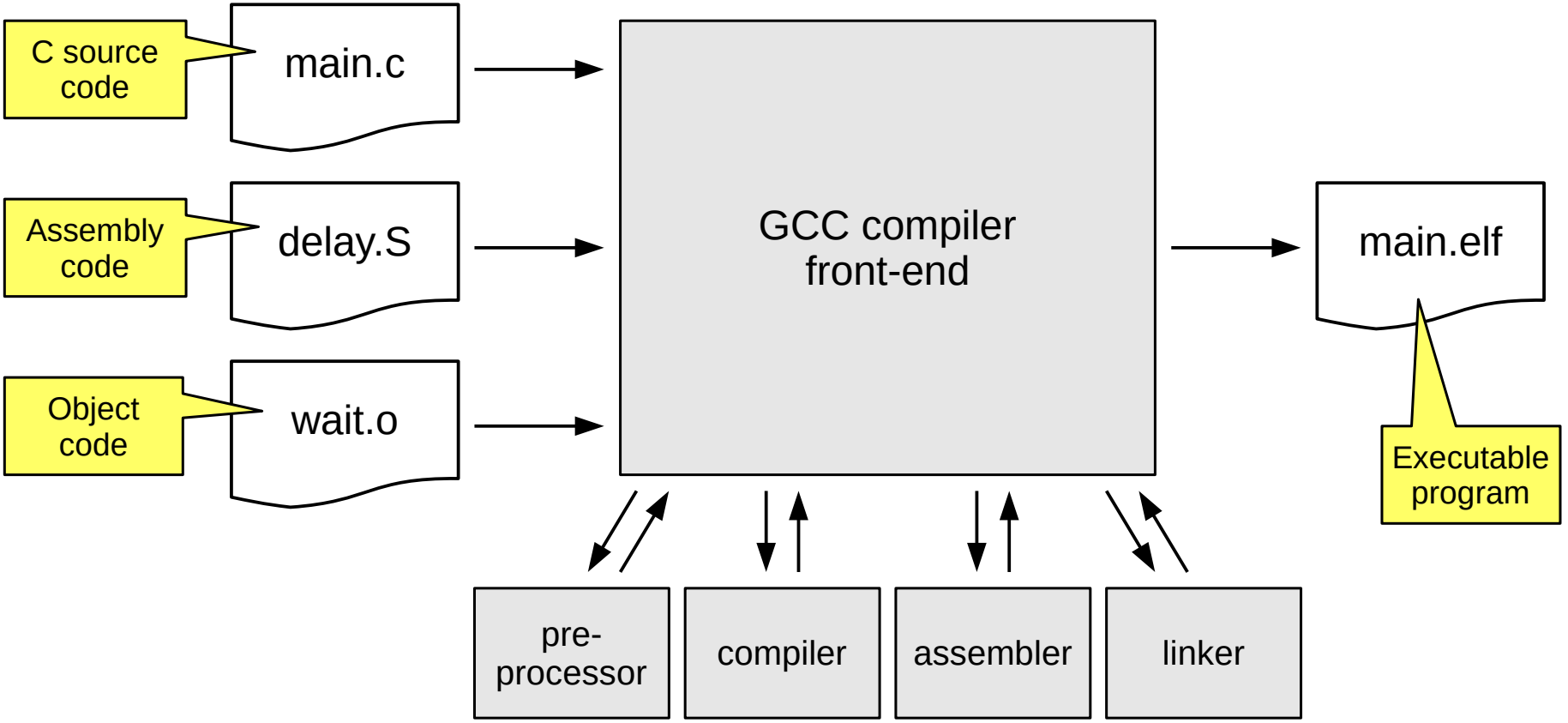
Tools for embedded systems are cross-tools: the tools run in a different architecture than the code they produce:

- Cross-compiler
- Cross-assembler
- Etc.

Typical C code toolchain



The GCC compiler front-end




```
$ avr-gcc -mmcu=atmega328p -lm -o main.elf main.c delay.S wait.o
```

Note: assembly **.S** files will be pre-processed, **.s** files will not.

C pre-processor macros

- Every assembly or C AVR source file should include the **avr/io.h** header file that defines standard pre-processor macros for AVR.
- The appropriate macros for the MCU in use will be loaded automatically (from the **-mmcu** paramter to **avr-gcc**).
- All peripheral registers are defines, like PORTB, DDRB, PINB, PB5, etc.
- Some convenient macro functions are also defined:
  - **_SRF_IO_ADDR(x)**: converts x from memory to I/O address.
    - _SRF_IO_ADDR(x) = x - 32 = x - 0x20.
  - **_BV(x)**: Bit Value for number x.
    - _BV(x) = (1 << x)

Pre-processor macros are text transformations run before actual compilation/assembly.

- Useful to define names for constants.
- Some macros may take a parameter making them look like functions.

GNU assembler directives

- Directives:
  - **.text**: assemble in code segment.
  - **.data** / **.bss**: assemble in the .data/.bss data segments. In the AVR the .data segment cannot be initialized so it is equivalent to the .bss segment (more details later on).
  - **.skip <n>**: reserves <n> bytes in memory.
  - **.lcomm <label>, <number>**: reserves (and label) <number> bytes in the .bss data memory segment (uninitialized data).
  - **.byte <b1>, <b2>, ...**: defines bytes in memory.
  - **.2byte <n1>, <n2>, …**: Defines 2 bytes data words in memory. Uses the endianess of the target architecture (little endian in the AVR). **.4byte** and **.8byte** also available. **.word** is equivalent to **.2byte** in the AVR.
  - **.align n** / **.balign n**: align code or data to multiples of *2^n* (.align) or *n* (.balign) bytes. ".align 1" is equivalent to ".balign 2".
    - Program instructions should be aligned to even addresses. Always use ".balign 2" after bytes lists to avoid misalignments.
  - **.equ** / **.set <symbol>, <expression>**: assign a symbol to an expression.
  - **.include <file>**: includes another <file>.

Assembler directives instruct the assembler on how to create object code, both program instructions and data.

GNU assembler expressions

- Constants
  - 10, -17, 0x7a, 0b01111010, “Hello!\n”, 'a', …
- Operators:
  - ~, -, +, *, /, %, <<, >>, &, |, ^, etc.
- Functions
  - **lo8(x)**: lowest significant byte of x.
  - **hi8(x)**: second lowest significant byte of x.
  - **hlo8(x)**, **hhi8(x)**: third and fourth significant bytes of x.
  - **pm_lo8(x)**, **pm_hi8(x)**: like lo8 and hi8 but returns program memory “word” addresses.
  - **exp2(x)**, **log2(x)**: base 2 exponential and logarithm.
  - **abs(x)**: absolute value.

Assembler expressions are evaluated at assembly time so they can only use constant data.

A first AVR assembly program

Example 1

a) Write an assembly program for the ATmega328P in an Arduino UNO board that reads PORTD7 (UNO D7) and write its value to PORTB5 (UNO D13 -LED-). Use load and store instructions to access port data and logic instructions to alter their bits.

b) Modify the program to use the more efficient bit and bit-branch instructions.

c) Modify the program to activates PORTB5 only when PORTD6 and PORTD7 are one (emulates an AND gate).

d) Estimate the delay from an input change to an output change if the MCU runs at 16MHz. Compare with the delay of a 74HC08 logic gate.

A first AVR assembly program

```
; Read macros with port numbers (and more)
#include <avr/io.h>

.text                       ; Program code starts here

.global main                ; "main" identifies program starting point
main:
    ldi  r16, 0b00100000    ; configure PB5 as output
    sts  DDRB, r16
    ldi  r16, 0             ; configure PORTD as input
    sts  DDRD, r16
    ldi  r16, 0b10000000    ; activate PD7 pull-up
    sts  PORTD, r16
loop:
    lds  r16, PORTB         ; read PORTB (PORTB is the output data)
    lds  r17, PIND          ; read PORTD (PIND is the external input)
    andi r17, 0b10000000    ; check bit 7
    breq is_zero
    ori  r16, 0b00100000    ; force PB5 to 1
    jmp  continue
is_zero:
    andi r16, 0b11011111    ; force PB5 to 0
continue:
    sts  PORTB, r16         ; write new value in PORTB
    jmp  loop
```

The code is similar to YASAC's, but digital ports are different here.

Load/store instructions do the job but the AVR has more powerful instructions for the task: bit set/clear instructions and bit test and skip instructions.

A first AVR assembly program

```
; Read macros with port numbers (and more)
#include <avr/io.h>

; The _SFR_IO_ADDR(x) macro converts from memory addresses
; to i/o addresses.

.text                               ; Program code is here

.global main                        ; "main" is program's starting point
main:
    sbi  _SFR_IO_ADDR(DDRB), DDB5   ; configure PB5 as output
    ldi  r16, 0                     ; configure PORTD as input
    out  _SFR_IO_ADDR(DDRD), r16
    sbi  _SFR_IO_ADDR(PORTD), PD7   ; activates PD7 pull-up
loop:
    sbis _SFR_IO_ADDR(PIND), PIND7  ; skip next instruction if PIND7 is 1
    rjmp is_zero
    sbi  _SFR_IO_ADDR(PORTB), PB5   ; set PB5 to 1
    rjmp continue
is_zero:
    cbi  _SFR_IO_ADDR(PORTB), PB5   ; set PB5 to 0
continue:
    rjmp loop
```

Note the use of the **_SFR_IO_ADDR()** macro. It converts memory addresses to I/O addresses so that the address can be used with I/O instructions.

We have also substituted **jmp** instructions by **rjmp**. **rjmp** uses less memory and is faster, but cannot jump very far away.

No. instructions	load/store	bit test
Initialization	6	4
Main loop	9	6
Program	15	10

Program size reduction: 33%

Assembling, uploading and testing

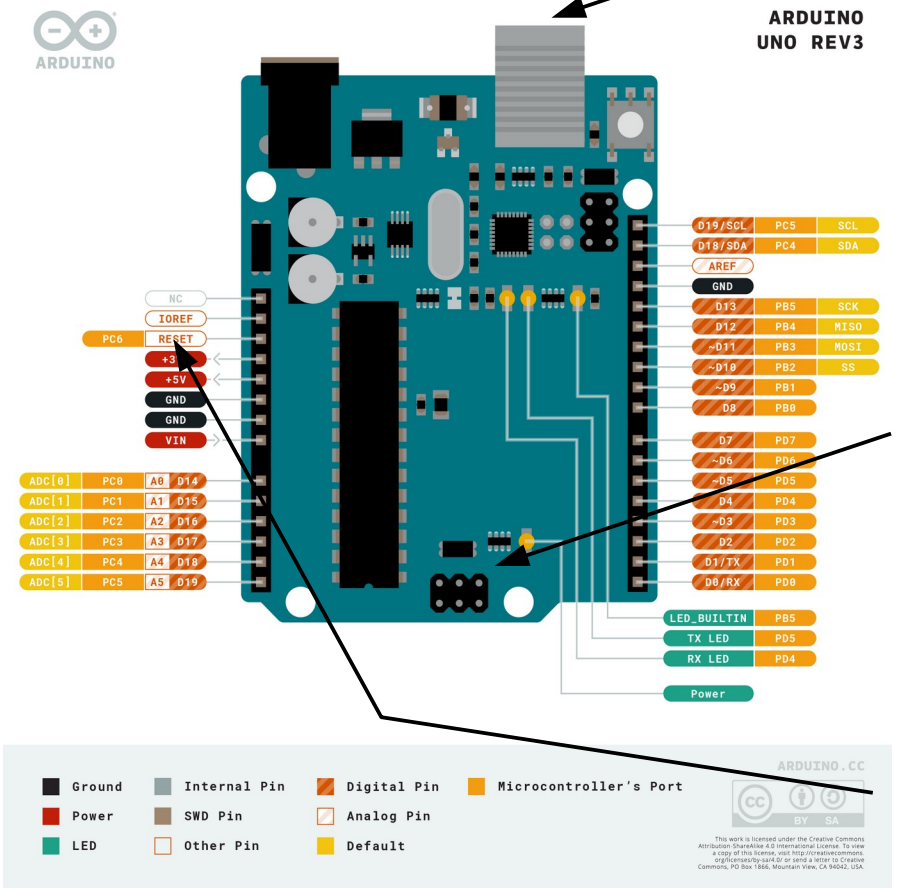
```
$ avr-gcc -mmcu=atmega328p -o main.elf main.S

$ avr-objcopy -j .text -j .data -O ihex main.elf main.hex

$ avrdude -c arduino -p atmega328p -P /dev/ttyACM0 -b 115200 \
 -D -U flash:w:main.hex:i

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.00s

avrdude: Device signature = 0x1e950f (probably m328p)
avrdude: reading input file "main.hex"
avrdude: writing flash (152 bytes):

Writing | ################################################## | 100% 0.04s

avrdude: 158 bytes of flash written
avrdude: verifying flash memory against main.hex:
avrdude: load data flash data from input file main.hex:
avrdude: input file main.hex contains 152 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.03s

avrdude: verifying ...
avrdude: 152 bytes of flash verified

avrdude: safemode: Fuses OK (E:00, H:00, L:00)

avrdude done.  Thank you.
```

Assemble and link

Convert to Intel hex format

Upload to the microcontroller

Arduino UNO and AVR MCU's programming (uploading) alternatives



USB port

The A. UNO has an USB to RS-232 serial port converter in the board. The programming software communicates with the Arduino bootloader in the MCU through the serial port and the bootloader writes the code to program memory and executes it.

Needs the Arduino bootloader pre-loaded.

ICSP (In-Chip Serial Programming)

The ICSP port is connected to the SPI interface in the ATmega328p. The programming logic in the chip can read instructions from the SPI and write it directly to program memory.

Needs an external programmer board (or another Arduino board running a programming software).

debugWIRE

Single pin protocol (connected to RESET) that allows programming and in-chip debugging.

Useful for MCU's that have a few pins only.

Needs an external control board.

See the AVR block diagram

Debugging programs

- Programmers (humans) are not perfect, thus programs have errors (bugs).
- Finding and correcting errors in programs (debugging) can be really tough.
  - Just executing and watching the program failing does not help very much.
- Debuggers can execute programs in a controlled way:
  - Single instruction execution.
  - Execution until a break point.
  - Stop the program when a condition is met.
  - Inspect registers/variables as the program runs.
  - Change the value of registers/variables as the program runs.
  - Etc.

Debugging programs

- Native vs non-native debuggers
  - Native: the debugger runs in the same computer than the program being debugged.
  - Non-native: the debugger runs in a different computer than the program being debugged.
    - The program being debugged runs in the target system.
- Non-native debuggers
  - The target system may be a real computer or a simulated one.
  - Microcontroller development uses non-native debuggers (the target system is not powerful enough to run the debugger).
  - MCU's with a debug unit can be used as target systems: in-chip debugging.
- Debug symbols
  - The compiler can include debug symbols in the object code with links to the original source code (-g option in gcc).
  - It makes debugging much more powerful and easy.

Debugging AVR programs with avr-gdb and simavr

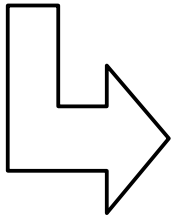
- avr-gdb is the GNU debugger (GDB) for the AVR platform.
  - GDB is one of the most powerful debuggers around.
  - It's command-line based.
  - Can be used directly in a the terminal window or through a front-end.
  - Most Integrated Development Environments (IDEs) have a front-end to GDB.
- simavr is a simulator for the AVR family.
  - Can simulate most AVR MCUs.
  - Can work as a GDB's remote target

Debugging AVR programs with avr-gdb and simavr

```
$ avr-gcc -mmcu=atmega328p -g -o main.elf main.S
```

```
$ simavr -g -m atmega328p main.elf
Loaded 152 .text at address 0x0
Loaded 0 .data
avr_gdb_init listening on port 1234
gdb_network_handler connection opened
```

```
┌──Register group: general──────────────────────────────────────────────────────
│r0             0x0      0
│r1             0x0      0
│r2             0x0      0
│r3             0x0      0
│r4             0x0      0
│r5             0x0      0
   ┌──main.S──────────────────────────────────────────────────────────────────
   │19      .global main                          ; "main" is program's start
   │20      main:
 > │21              sbi  _SFR_IO_ADDR(DDRB), DDB5   ; configure PB5 as output
   │22              ldi  r16, 0                     ; configure PORTD as input
   │23              out  _SFR_IO_ADDR(DDRD), r16
   │24              sbi  _SFR_IO_ADDR(PORTD), PD7   ; activates PD7 pull-up
   └──────────────────────────────────────────────────────────────────────────
remote Thread <main> In: main                                  L21   PC: 0x80
Single stepping until exit from function __vectors,
which has no line number information.
0x00000068 in __trampolines_start ()
(gdb) step
Single stepping until exit from function __trampolines_start,
which has no line number information.
main () at main.S:21
(gdb)
```



See the avr-bare repo for a quick introduction to avr-gdb and simavr simulation.

Integrated Development Environments for AVR microcontrollers

- PlatformIO
  - Free Software IDE integrated with VSCode editor.
  - Multi-platform.
  - Supports many embedded systems platforms, including the AVR.
  - Uses the GNU toolchain, avrdude and simavr as back-end.
  - Supports the Arduino Framework.
    - Can be used to develop Arduino projects (with Arduino libraries).
- Microchip Studio (former Atmel Studio)
  - Official IDE from the AVR foundry.
  - Supports other MCUs from Microchip (PIC, etc.).
  - Supported toolchains:
    - Basic Atmel assembler.
    - GNU toolchain.
    - Microchip's own C compiler and toolchain.
  - Based on MS Visual Studio (MS Windows only).
  - Nice peripheral interface within the debugger.

PlatformIO tips. Setup

- Create a new project using the AVR template instructions.
- In the PlatformIO tab, project taks menu, use:
  - Build, to compile/assemble your project and check erros.
  - Upload, to upload your program to the board.
  - Clean, to clean generated code.
- Debugging
  - Set option “debug.allowBreakpointsEverywhere” to TRUE in File → Preferences → Settings → …
  - In Ubuntu (as of 2021), install the libncurses5 package.
  - From the “Run and Debug” tab, use “PIO Debug” task to start debugging.

PlatformIO tips. Debug control

- Debug control: from “Run” menu or debug panel on top of the tab.
  - Right-click on code line to add/remove break points.
  - Continue (F5): run program until next break point or end of program.
  - Step over (F10): execute one instruction but “step over” subroutines: executes subroutines in one step.
  - Step into (F11): execute one instruction and “step into” subroutines: execute subroutines step by step.
  - Step out (Shift-F11): continue execution until leaving the current subroutine.
  - Stop (Shift-F5): stop the program and leave the debugging session.

PlatformIO tips. Inspecting data

- From debug console (GDB interface): use any GDB command. E.g.:
  - Print register r16: p $r16
  - Display PINB in binary format: display /t PINB
  - Print register ‘y’ in hexadecimal format: p /x 256*$r29 + $r28
  - Print 10 bytes starting at ‘data_list’ in hexadecimal format: x /10xb &data_list
- Add expression to the “watch” sidebar. Use any GDB-compatible expr. E.g.:
  - Register value: $r16
  - Register expression (e.g. ‘y’ register value): $r28+256*$r29
  - I/O register: PORTB, PIND, etc.
  - Address of symbol ‘data_list’: &data_list
  - Value at address ‘result’ (little-endian, 16-bit integer): result
  - 10 values of type ‘int’ at address ‘data_list’: (int [10])data_list
- Memory: add memory start and range to the “memory” side bar (note: data memory starts at 0x800000, data segment at 0x800100):
  - First 10 bytes of the data segment: 0x800100, 10

PlatformIO tips. Modifying data

- From debug console (GDB interface): use any GDB command. E.g.:
  - Set register r16 to 17: set $r16=17
  - Set value at memory address ‘result’ (16-bit, little endian): set result=0x3a
  - Set PIND5 to 1, others to 0: set PIND=0b00100000
- From peripherals panel
  - Select the register.
  - Right-click → Update value

A first AVR assembly program

Example 1

c) Modify the program to activate PORTB5 only when PORTD6 and PORTD7 are one (emulates an AND gate).

```
; A software AND gate
; Inputs: PD6, PD7 (active pull-up)
; Output: PB5

; Read macros with port numbers (and more)
#include <avr/io.h>

.text                                   ; Program code starts here

.global main                            ; "main" is program's starting point
main:
        sbi  _SFR_IO_ADDR(DDRB), DDB5   ; configure PB5 as output
        ldi  r16, 0                     ; configure PORTD as input
        out  _SFR_IO_ADDR(DDRD), r16
        sbi  _SFR_IO_ADDR(PORTD), PD6   ; activates PD6 and PD7 pull-up
        sbi  _SFR_IO_ADDR(PORTD), PD7
loop:
        sbis _SFR_IO_ADDR(PIND), PIND6  ; skip next instruction if PIND7 is 1
        rjmp is_zero
        sbis _SFR_IO_ADDR(PIND), PIND7  ; skip next instruction if PIND7 is 1
        rjmp is_zero
        sbi  _SFR_IO_ADDR(PORTB), PB5   ; set PB5 to 1
        rjmp continue
is_zero:
        cbi _SFR_IO_ADDR(PORTB), PB5    ; set PB5 to 0
continue:
        rjmp loop
```

Write the code and debug it in PlatformIO.

A first AVR assembly program

Example 1

d) Estimate the delay from an input change to an output change if the MCU runs at 16MHz. Compare with the delay of a 74HC08 logic gate.

- Worst case is when the input changes right after it has been checked: the whole loop have to be executed before activating the output.
- Input changes from 10 to 11 (output changes from 0 to 1)
  - rjmp (2), cbi (2), rjmp (2), sbis-true (3), sbis-true (3), sbi (2), input sync (1), output sync (1)
  - 16 cycles at 16MHz = 1 microsecond.
- Input changes from 11 to 10 (output changes from 1 to 0)
  - Fewer cycles because one sbis instruction is false (1 cycle only).
- Typical 74HC delay: 10ns = 0,01us (100 times faster).
  - NOTE: the actual gate is much faster. This delay includes the input and output buffers in the chip.

Assembly subroutines

- Programs in high-level languages use functions, objects and methods extensively. Assembly language programs use subroutines for the same purpose:
  - Code re-usability, program modularization, easier development, easier debugging, etc.
- Arguments can be passed to subroutines using registers or the stack.
  - Which option is better?
  - Which registers may be used?
- Subroutines will use registers for their calculations.
  - Will a subroutine clobber my used registers?
  - Who is responsible for saving register values, caller or callee?

Assembly subroutines

Example 2

Write a subroutine in assembly that adds up a list of 8-bit number in data memory. The subroutine should add all the numbers up to the first one that is zero.

The memory address where the list starts is passed to the subroutine in registers r25:r24.

The sum should be left in register r24 before returning from the subroutine.

Leave the subroutine in a separate file so that it can be re-used in other projects, and write a simple program to test it.

Assembly subroutines
Add list main (test) program

```
; File: main.S

#include <avr/io.h>

.data                           ; Work on the data segment (SRAM memory)
.lcomm data_list, 10            ; Save 10 bytes to store data

.text                           ; Program code starts here
.global main
main:
        ; initialize the output pin
        sbi  _SFR_IO_ADDR(DDRB), PB5    ; set PB5 as output
        cbi  _SFR_IO_ADDR(PORTB), PB5   ; set PB5 to zero

        ; load some data in the data list
        ldi  yl, lo8(data_list)         ; use Y register as pointer to the data
        ldi  yh, hi8(data_list)
        ldi  r16, 1
        st   y+, r16                    ; indirect store with post-increment
        ldi  r16, 2
        st   y+, r16
        ldi  r16, 3
        st   y+, 16
        ldi  r16, 5
        st   y+, r16
        ldi  r16, 8
        st   y+, r16
        clr  r16                        ; the last value is 0
        st   y, r16                     ; no need to post-increment this time

        ; prepare arguments and call the subroutine
        ldi  r24, lo8(data_list)
        ldi  r25, hi8(data_list)
        call add_list

        ; check the return value (should be 19) and activate PB5 if correct
        cpi  r24, 19
        brne finish                     ; not correct. finish the program
        sbi  _SFR_IO_ADDR(PORTB), PB5   ; activate PB5
finish:                                 ; just stay here for a while
        rjmp finish
```

Assembly subroutines
Add list subroutine

```
; File:         add_list.S
; Description:  Subroutine to add a list of bytes in memory up to the first
;               byte that is zero.
;               Input:  list address in r25:r24.
;               Return: list sum in r24
; Author:       Jorge Juan-Chico
; Date:         2021-05-27

; Read macros with port numbers (and more)
#include <avr/io.h>

.text                           ; this goes to the code segment

.global add_list                ; must be global to bee seen from other files
add_list:
        ; use Z as pointer and r24 as accumulator
        mov  zl, r24
        mov  zh, r25
        clr  r24

        ; main loop: read data from the list, return if zero or accumulate and
        ; read the next value.
loop:
        ld   r0, z+
        tst  r0                  ; test r0 (equivalent to and r0,r0)
        breq finish
        add  r24, r0
        rjmp loop
finish:
        ret
```

Quick exercise

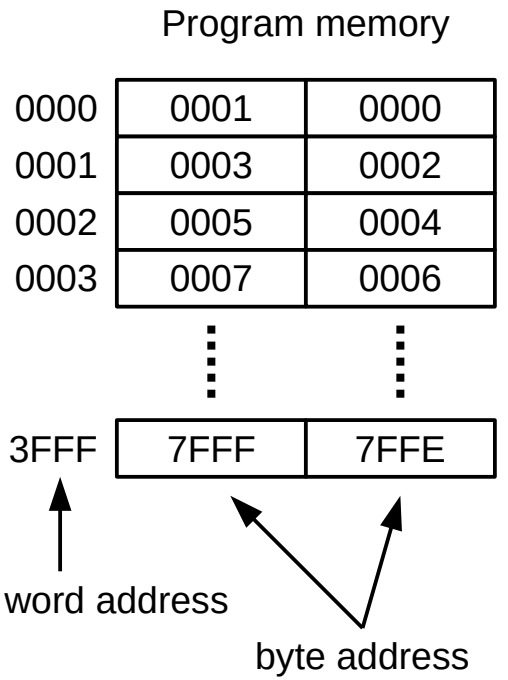
Take a look at the main loop. Can it be improved to run faster?

AVR handling of static data

- Regular programs use different types of data:
  - Constant data: unmodifiable data that stays during the whole the program lifetime.
    - Stored with program code in the ".text" segment or another read-only segment.
  - Static data: modifiable data that stays during all the program.
    - Stored in the ".data" segment if it initialized.
    - Stored in the ".bss" segment (from "block starting symbol") if it is not initialized.
  - Dynamic data: created and destroyed as the program runs.
- AVR programmers cannot initialize RAM memory so data in the .data segment cannot be initialized.
  - Constant data can be stored in the .text segment, together with the program code.
    - Use ".balign 2" after defining bytes to avoid code misalignment.
  - Initialized static data can be emulated by copying constant data from program memory (.text) to data memory (.data) at the beginning of the program.
    - GCC does it automatically for C programs.

Reading and writing program memory

- AVR MCUs can load and store data from/to program memory.
- LPM (Load Program Memory)
  - Load bytes from program memory.
  - Uses program memory “byte address” instead of “word address”.
  - Uses the Z register as pointer.
- SPM (Store Program Memory)
  - Not supported by all AVR MCUs.
  - Flash memory blocks need to be erased before writing.
  - Flash blocks have to be written at once (no byte or word write).
  - Mostly useful in bootloader code.



LPM		Load Program Memory	R0 ← (Z)	None	3
LPM	Rd, Z	Load Program Memory	Rd ← (Z)	None	3
LPM	Rd, Z+	Load Program Memory and Post-Inc	Rd ← (Z), Z ← Z+1	None	3
SPM		Store Program Memory	(Z) ← R1:R0	None	-

Branch instructions use word addresses. GNU tools use byte addresses for display, like in **avr-gdb** or **avr-objdump**.

Reading and writing program memory

Example 3

a) Write a test program for the add list subroutine of example 2 that creates the list in data memory by copying a constant list stored in program memory.

b) Write a similar subroutine to the one in example 2 but operating on constant data stored in program memory instead of data memory. Modify the test program to test it.

Reading and writing program memory

Example 3

a) Write a test program for the add list subroutine of example 2 that creates the list in data memory by copying a constant list stored in program memory.

```
; Input:        none.
; Output:       Activates PB5 if the subroutine is correct.

; Read macros with port numbers (and more)
#include <avr/io.h>

.data                           ; Work on the data segment (SRAM memory)
.lcomm data_list, 10

.text                           ; Program code starts here
data_list_pm:
        .byte 1, 2, 3, 5, 8, 0
        .balign 2

.global main
main:
        ; initialize the output pin
        sbi  _SFR_IO_ADDR(DDRB), PB5    ; set PB5 as output
        cbi  _SFR_IO_ADDR(PORTB), PB5   ; set PB5 to zero

        ; copy data list from program to data memory
        ldi  zl, lo8(data_list_pm)      ; setup pointers
        ldi  zh, hi8(data_list_pm)
        ldi  yl, lo8(data_list)
        ldi  yh, hi8(data_list)
loop:
        lpm  r0, z+                     ; << NOTE: this is "lpm" not "ld"
        st   y+, r0
        tst  r0
        brne loop

        ; prepare arguments and call the add_list subroutine
        ldi  r24, lo8(data_list)
        ldi  r25, hi8(data_list)
        call add_list

        ; check the return value (should be 19) and activate PB5 if correct
        cpi  r24, 19
        brne finish                     ; not correct. finish the program
        sbi  _SFR_IO_ADDR(PORTB), PB5   ; activate PB5
finish:                                 ; just stay here for a while
        rjmp finish
```

Reading and writing program memory

Example 3

b) Write a similar subroutine to the one in example 2 but operating on constant data stored in program memory instead of data memory. Modify the test program to test it.

```
; Subroutine:   add_list_pm
; File:         add_list_pm.S
; Description:  Subroutine to add a list of bytes in program memory.
; Author:       Jorge Juan-Chico
; Date:         2021-05-27
;
; Arguments:    r25:r24 - address of the list (byte address in program mem.)
; Return:       r24     - result of the sum.
; Description:  Add all number up to the first zero byte.

; Read macros with port numbers (and more)
#include <avr/io.h>

.text                       ; this goes to the code segment

.global add_list_pm         ; must be global to bee seen from other files
add_list_pm:
        ; use Z as pointer and r24 as accumulator
        mov  zl, r24
        mov  zh, r25
        clr  r24

        ; main loop: read data from the list, return if zero
        ; or accumulate and read the next value.
loop:
        lpm  r0, z+
        tst  r0             ; test r0 (equivalent to and r0,r0)
        breq finish
        add  r24, r0
        rjmp loop
finish:
        ret
```

Reading and writing program memory

Example 3

b) Write a similar subroutine to the one in example 2 but operating on constant data stored in program memory instead of data memory. Modify the test program to test it.

```
; Input:        none.
; Output:       Activates PB5 if the subroutine is correct.

; Read macros with port numbers (and more)
#include <avr/io.h>

.text                           ; Program code starts here
data_list_pm:
        .byte 1, 2, 3, 5, 8, 0
        .balign 2

.global main
main:
        ; initialize the output pin
        sbi  _SFR_IO_ADDR(DDRB), PB5    ; set PB5 as output
        cbi  _SFR_IO_ADDR(PORTB), PB5   ; set PB5 to zero

        ; prepare arguments and call the add_list subroutine
        ldi  r24, lo8(data_list_pm)
        ldi  r25, hi8(data_list_pm)
        call add_list_pm

        ; check the return value (should be 19) and activate PB5 if correct
        cpi  r24, 19
        brne finish                     ; not correct. finish the program
        sbi  _SFR_IO_ADDR(PORTB), PB5   ; activate PB5
finish:                                 ; just stay here for a while
        rjmp finish
```

Working with data other than 8 bits

- The AVR core is an 8-bit processor, but includes some instructions to make it easier to operate with 16-bit data (words) or even wider or narrower numbers.
- Instruction that operate with words:
  - ADIW, SBIW: add/subtract a constant to a word (pair of registers).
  - MUL, MULS, MULSU: multiply instructions write the result to R1:R0.
  - MOVW: copy register word.
  - SPM: store data words in program memory (R1:R0).
- Instructions that support multi-byte operations:
  - ADC, SBC, SBCI: addition and subtraction with carry.
  - CPC: compare with carry.
- Instructions that operate on 4-bit data (nibbles).
  - SWAP: swap nibbles in a register. Useful with BCD numbers.

Working with data other than 8 bits

Example 4 (16-bit version of example 2)

Write a subroutine in assembly that adds up a list of 8-bit number in data memory. The subroutine should add all the numbers up to the first one that is zero and return the result as a 16-bit word.

The memory address where the list starts is passed to the subroutine in registers r25:r24.

The sum should be left in register r25:r24 before returning from the subroutine.

Leave the subroutine in a separate file so that it can be re-used in other projects, and write a simple program to test it.

Working with data other than 8 bits

```
; Arguments:    r25:r24 - address of the list.
; Return:       r25:r24 - result of the sum (16 bits).
; Description:  Add all numbers (bytes) up to the first zero byte.

; Read macros with port numbers (and more)
#include <avr/io.h>

.text                       ; this goes to the code segment

.global add_list_w          ; must be global to bee seen from other files
add_list_w:
        ; use Z as pointer and r25:r24 as accumulator
        mov  zl, r24
        mov  zh, r25
        clr  r25
        clr  r24

        ; main loop: read data from the list, return if zero
        ; or accumulate and read the next value.
loop:
        ld   r0, z+
        tst  r0             ; test r0 (equivalent to and r0,r0)
        breq finish
        add  r24, r0
        adc  r25, r1        ; add carry to r25. r1=0 under GNU toolchain
        rjmp loop
finish:
        ret
```

Subroutine

Add the LSB first and the MSB next with the carry bit.

LSB: Least Significant Byte.
MSB: Most Significant Byte.

Working with data other than 8 bits

Test program

```
; Input:        none.
; Output:       Activates PB5 if the subroutine works correctly.
; Description:  Adds the prime numbers that are lower than 256.

; Read macros with port numbers (and more)
#include <avr/io.h>

; pre-calculated result
.equ RESULT, 6081

.data                ; Work on the data segment (SRAM memory)
.lcomm data_list, 100

.text                ; Program code starts here

; Static data is at the end of the text segment.
; Putting too much data before "main" may prevent
; the initialization code to "rjmp" to "main".

.global main
main:
    ; initialize the output pin
    sbi  _SFR_IO_ADDR(DDRB), PB5    ; set PB5 as output
    cbi  _SFR_IO_ADDR(PORTB), PB5   ; set PB5 to zero

    ; copy data list from program to data memory
    ldi  zl, lo8(data_list_pm)      ; setup pointers
    ldi  zh, hi8(data_list_pm)
    ldi  yl, lo8(data_list)
    ldi  yh, hi8(data_list)
loop:
    lpm  r0, z+
    st   y+, r0
    tst  r0
    brne loop
```

```
    ; prepare arguments and call the add_list subroutine
    ldi  r24, lo8(data_list)
    ldi  r25, hi8(data_list)
    call add_list_w

    ; check the return value and finish if not correct
    cpi  r24, lo8(RESULT)         ; compare and skip if equal
    brne finish
    cpi  r25, hi8(RESULT)
    brne finish
    sbi  _SFR_IO_ADDR(PORTB), PB5 ; activate PB5
finish:                           ; just stay here for a while
    rjmp finish

; sequence of prime numbers that can be represented with 8 bits.
data_list_pm:
    .byte 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, \
          43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, \
          101, 103, 107, 109, 113, 127, 131, 137, 139, 149, \
          151, 157, 163, 167, 173, 179, 181, 191, 193, 197, \
          199, 211, 223, 227, 229, 233, 239, 241, 251, 0
    .balign 2
```

Controlling time

- Controlling time is important in many MCU applications:
  - Wait an amount of time before taking an action.
    - E.g. triggering an alarm only when the sensor is active for at least 3 seconds.
  - Do something regularly with a fixed time lapse.
    - E.g. blinking a LED or incrementing a counter.
  - Waiting some time after a button press to avoid bounces.
- Two ways to control time:
  - Waiting loops
    - Execute a loop a number of times.
    - The processor does not stop (bad for energy consumption).
    - Elapsed time can be calculated counting the number of cycles and using the processor's clock frequency.
  - Using a timer peripheral and possible interrupts (will see later).

Controlling time

Example 5

Write a subroutine in assembly that waits for a number of milliseconds. The number of milliseconds to wait is passed in registers r25:r24. Returns r25:r24=0.

Using the subroutine, write an assembly program that makes an LED to blink once every second: it is 500ms on and 500ms off. The LED is connected to pin PB5.

Subroutine

```
; Define the board's CPU frequency in Hz. Depends on the board and chip
; configuration.
.equ F_CLK, 16000000

; Number of passes of the 1ms delay loop.
.equ NPASS_1MS, F_CLK/1000/4
;                     ^    ^
;                     |    |
;     change units to ms   No. of cycles of the 1ms loop (see below)

; Code segment
;
.text
;
; Subroutine: delay_ms
;
; Input: wait time in ms (r25:r24)
; Return: 0 (r25:r24 = 0)
;
.global delay_ms
delay_ms:
    ldi r27,hi8(NPASS_1MS)          ; initialize 1ms loop
    ldi r26,lo8(NPASS_1MS)
wait_1ms_loop:                      ; 1ms loop (see AVR Instruction Set Manual)
    sbiw r26,1                      ; substract 1 (2 cycles)
    brne wait_1ms_loop              ; compare (1 cycle if false, 2 if true)
    sbiw r24,1                      ; main loop check
    brne delay_ms
    ret
```

1ms loop.

Repeat r25:r24 times.

Controlling time

Test program

```
; Commutes bit 5 in PORTB (PB5) every 500ms. It makes the built-in LED to
; blink in the Arduino UNO board that uses an ATmega328p. Can be easily
; modified to us a different output bit or blinking frequency.
; Uses the delay_ms subroutine in delay_ms.S.

; AVR i/o symbols definitions. Defines port and memory names for easier
; programming. Do not write AVR programs without it
#include <avr/io.h>

; Blinking delay in ms (half blinking period).
.equ DELAY, 500

;
; Code segment
;
.text                                   ; text (code) segment starts here

;
; Main program
;
.global main
main:                                   ; our program starts here
    ; Initialization code
    ldi r16, _BV(PB5)                   ; set bit 5 in PORTB as output
    out _SFR_IO_ADDR(DDRB), r16         ; (the built-in LED in Arduino UNO is here)
    cbi _SFR_IO_ADDR(PORTB), PB5        ; clear PB5
    ldi r16, lo8(DELAY)                 ; preload delay
    ldi r17, hi8(DELAY)

    ; Main loop
loop:
    movw r24, r16                       ; argument to the delay_ms subroutine
    call delay_ms                       ; wait
    sbi _SFR_IO_ADDR(PINB), PB5         ; toggle the LED
    rjmp loop                           ; repeat forever
```

In output mode, setting PINB toggles the corresponding pin.

AVR C/C++ programming

- Most AVR microcontrollers can be programmed in C/C++ (like the Atmega series).
  - The C compiler needs a minimum set of characteristics: data memory, stack, etc.
- The AVR C library provides lots of useful functions
  - Strings, floating point maths, serial port printing, etc.
- Supported by the GNU compiler and other GNU tools
  - Free software
- A major reason for the success of the AVR family.
- Foundation of the Arduino framework.

C program example

Example 6

a) Write a C program for the ATmega328P in an Arduino UNO board that reads PORTD7 (UNO D7) and write its value to PORTB5 (UNO D13 -LED-).

b) Debug the program using a debugger and simulator. Test the program in the board if you have one.

C program example

```c
// Trivial AVR C program
// Read bit 7 of PORTD and write bit 5 of PORTB

// Read macros with port numbers (and more)
#include <avr/io.h>

void main (void)
{
      unsigned int data;

      DDRB = _BV(PB5);          // configure PB5 as output
      DDRD = 0;                 // configure PORTD as input
      PORTD = _BV(PD7);         // activate PD7 pull-up

      while (1) {
          data = PIND & _BV(7);    // read P
          if (data)
              PORTB |= _BV(5);     // force
          else
              PORTB &= ~_BV(5);    // force
      }
}
```

Same as:
DDRB = 0b00100000

Same as:
PORTB = PORTB & 0b11011111

Our assembly version (example 1)

```
; Read macros with port numbers (and more)
#include <avr/io.h>

; The _SFR_IO_ADDR(x) macro converts from memory addresses
; to i/o addresses.

.text                                ; Program code starts here

.global main                         ; "main" is program's starting point
main:
    sbi  _SFR_IO_ADDR(DDRB), DDB5    ; configure PB5 as output
    ldi  r16, 0                      ; configure PORTD as input
    out  _SFR_IO_ADDR(DDRD), r16
    sbi  _SFR_IO_ADDR(PORTD), PD7    ; activates PD7 pull-up
loop:
    sbis _SFR_IO_ADDR(PIND), PIND7   ; skip next instruction if PIND7 is 1
    rjmp is_zero
    sbi  _SFR_IO_ADDR(PORTB), PB5    ; set PB5 to 1
    rjmp continue
is_zero:
    cbi  _SFR_IO_ADDR(PORTB), PB5    ; set PB5 to 0
continue:
    rjmp loop
```

C compiler code optimization

- The C compiler can optimize the resulting assembly code for:
  - Code size, speed or both.
- Compiler optimization may:
  - Change order in which instructions are executed.
  - Alter the assembly instructions used to implement the C code.
  - Remove portions of code that the compiler thinks are useless.
- More optimization means more compilation time and memory usage.
- Typical optimization options (full list here)
  - -O<level>: <level> goes from 0 (no optimization) to 3 (maximum optimization). Default is no optimization (-O0).
  - -Os: optimize for size. Like -O2 but disables any optimization that may increase object code size.
  - -Og: optimize for debugging. Like -O1 but disables any optimization that may affect debugging.

Compiling only and disassembling

- To see what the compiler have done we have two options:
  - Disassembling the object code in the executable program.
  - Telling the compiler to just compile to assembly (no assembling or linking).
- Why disassembling?
  - Low level debugging, inspecting malicious code, etc.
  - Reverse engineering.
- Why compile only?
  - Debug the compiler.
  - Decide about optimization levels.
  - Learn how a compiler works.

Compiling only and disassembling

Example 7

a) Compile the program in example 5 to assembly language and compare the generated assembly code with that of example 1. Try different options for compiler optimization:

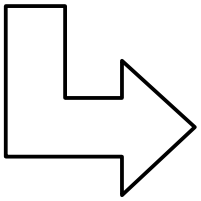
- -Os, -Og, etc.

b) Imagine you do not have the original C code anymore. You can still list the assembly code by disassembling the executable file. Disassemble the code compiled with size optimization “-Os” and compare with the assembly code generated in (a). Pay attention to the extra code introduced by the linker.

Compile without optimization (-O0)

```
avr-gcc -mmcu=atmega328p -S -O0 -o main_Os.s main.c
```

```
// Trivial AVR C program
// Read bit 7 of PORTD and
// write bit 5 of PORTB

// Read macros with port numbers
#include <avr/io.h>

void main (void)
{
      unsigned int data;

      DDRB = _BV(PB5);
      DDRD = 0;
      PORTD = _BV(PD7);

      while (1) {
          data = PIND & _BV(7);
          if (data)
              PORTB |= _BV(5);
          else
              PORTB &= ~_BV(5);
      }
}
```

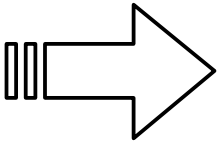


```
      .file "main.c"
__SP_H__ = 0x3e
__SP_L__ = 0x3d
__SREG__ = 0x3f
__tmp_reg__ = 0
__zero_reg__ = 1
      .text
.global     main
      .type main, @function
main:
      push r28
      push r29
      rcall .
      in r28,__SP_L__
      in r29,__SP_H__
/* prologue: function */
/* frame size = 2 */
/* stack size = 4 */
.L__stack_usage = 4
      ldi r24,lo8(36)
      ldi r25,0
      ldi r18,lo8(32)
      movw r30,r24
      st Z,r18
      ldi r24,lo8(42)
      ldi r25,0
      movw r30,r24
      st Z,__zero_reg__
      ldi r24,lo8(43)
      ldi r25,0
      ldi r18,lo8(-128)
      movw r30,r24
      st Z,r18
```

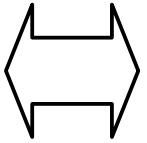
```
.L4:
      ldi r24,lo8(41)
      ldi r25,0
      movw r30,r24
      ld r24,Z
      mov r24,r24
      ldi r25,0
      andi r24,128
      clr r25
      std Y+2,r25
      std Y+1,r24
      ldd r24,Y+1
      ldd r25,Y+2
      or r24,r25
      breq .L2
      ldi r24,lo8(37)
      ldi r25,0
      ldi r18,lo8(37)
      ldi r19,0
      movw r30,r18
      ld r18,Z
      ori r18,lo8(32)
      movw r30,r24
      st Z,r18
      rjmp .L4
.L2:
      ldi r24,lo8(37)
      ldi r25,0
      ldi r18,lo8(37)
      ldi r19,0
      movw r30,r18
      ld r18,Z
      andi r18,lo8(-33)
      movw r30,r24
      st Z,r18
      rjmp .L4
      .size main, .-main
      .ident      "GCC: (GNU) 5.4.0"
```

Compile with size optimization (-Os)

```
avr-gcc -mmcu=atmega328p -S -Os -o main_Os.s main.c
```



```
        .file "main.c"
__SP_H__ = 0x3e
__SP_L__ = 0x3d
__SREG__ = 0x3f
__tmp_reg__ = 0
__zero_reg__ = 1
        .section    .text.startup,"ax",@progbits
.global     main
        .type main, @function
main:
/* prologue: function */
/* frame size = 0 */
/* stack size = 0 */
.L__stack_usage = 0
        ldi r24,lo8(32)
        out 0x4,r24
        out 0xa,__zero_reg__
        ldi r24,lo8(-128)
        out 0xb,r24
.L2:
        sbis 0x9,7
        rjmp .L3
        sbi 0x5,5
        rjmp .L2
.L3:
        cbi 0x5,5
        rjmp .L2
        .size main, .-main
        .ident      "GCC: (GNU) 5.4.0"
```



```
; Read macros with port numbers
#include <avr/io.h>

.text

.global main
main:
    sbi  _SFR_IO_ADDR(DDRB), DDB5
    ldi  r16, 0
    out  _SFR_IO_ADDR(DDRD), r16
    sbi  _SFR_IO_ADDR(PORTD), PD7
loop:
    sbis _SFR_IO_ADDR(PIND), PIND7
    rjmp is_zero
    sbi  _SFR_IO_ADDR(PORTB), PB5
    rjmp continue
is_zero:
    cbi  _SFR_IO_ADDR(PORTB), PB5
continue:
    rjmp loop
```

The compiler is good at optimizing code

Most code is compiled with optimization

Disassembling

```
avr-objdump -dr main.elf > main.dis
```

```
      .file "main.c"
__SP_H__ = 0x3e
__SP_L__ = 0x3d
__SREG__ = 0x3f
__tmp_reg__ = 0
__zero_reg__ = 1
    .section .text.startup,"ax",@progbits
.global main
    .type main, @function
main:
/* prologue: function */
/* frame size = 0 */
/* stack size = 0 */
.L__stack_usage = 0
      ldi r24,lo8(32)
      out 0x4,r24
      out 0xa,__zero_reg__
      ldi r24,lo8(-128)
      out 0xb,r24
.L2:
      sbis 0x9,7
      rjmp .L3
      sbi 0x5,5
      rjmp .L2
.L3:
      cbi 0x5,5
      rjmp .L2
      .size main, .-main
      .ident      "GCC: (GNU) 5.4.0"
```

Linker

```
main.elf:     file format elf32-avr

Disassembly of section .text:

00000000 <__vectors>:
   0: 0c 94 34 00       jmp   0x68  ; 0x68 <__ctors_end>
   4: 0c 94 3e 00       jmp   0x7c  ; 0x7c <__bad_interrupt>
   8: 0c 94 3e 00       jmp   0x7c  ; 0x7c <__bad_interrupt>
[...]
  64: 0c 94 3e 00       jmp   0x7c  ; 0x7c <__bad_interrupt>

00000068 <__ctors_end>:
  68: 11 24             eor   r1, r1
  6a: 1f be             out   0x3f, r1    ; 63
  6c: cf ef             ldi   r28, 0xFF   ; 255
  6e: d8 e0             ldi   r29, 0x08   ; 8
  70: de bf             out   0x3e, r29   ; 62
  72: cd bf             out   0x3d, r28   ; 61
  74: 0e 94 40 00       call  0x80  ; 0x80 <main>
  78: 0c 94 4b 00       jmp   0x96  ; 0x96 <_exit>

0000007c <__bad_interrupt>:
  7c: 0c 94 00 00       jmp   0     ; 0x0 <__vectors>

00000080 <main>:
  80: 80 e2             ldi   r24, 0x20   ; 32
  82: 84 b9             out   0x04, r24   ; 4
  84: 1a b8             out   0x0a, r1    ; 10
  86: 80 e8             ldi   r24, 0x80   ; 128
  88: 8b b9             out   0x0b, r24   ; 11
  8a: 4f 9b             sbis  0x09, 7     ; 9
  8c: 02 c0             rjmp  .+4         ; 0x92 <main+0x12>
  8e: 2d 9a             sbi   0x05, 5     ; 5
  90: fc cf             rjmp  .-8         ; 0x8a <main+0xa>
  92: 2d 98             cbi   0x05, 5     ; 5
  94: fa cf             rjmp  .-12        ; 0x8a <main+0xa>

00000096 <_exit>:
  96: f8 94             cli

00000098 <__stop_program>:
  98: ff cf             rjmp  .-2         ; 0x98 <__stop_program>
```

Our code is here.

Structure of an executable program (generated by the GNU toolchain)

```
main.elf:     file format elf32-avr

Disassembly of section .text:

00000000 <__vectors>:
   0: 0c 94 34 00       jmp   0x68  ; 0x68 <__ctors_end>
   4: 0c 94 3e 00       jmp   0x7c  ; 0x7c <__bad_interrupt>
   8: 0c 94 3e 00       jmp   0x7c  ; 0x7c <__bad_interrupt>
[...]
  64: 0c 94 3e 00       jmp   0x7c  ; 0x7c <__bad_interrupt>

00000068 <__ctors_end>:
  68: 11 24             eor   r1, r1
  6a: 1f be             out   0x3f, r1    ; 63
  6c: cf ef             ldi   r28, 0xFF   ; 255
  6e: d8 e0             ldi   r29, 0x08   ; 8
  70: de bf             out   0x3e, r29   ; 62
  72: cd bf             out   0x3d, r28   ; 61
  74: 0e 94 40 00       call  0x80  ; 0x80 <main>
  78: 0c 94 4b 00       jmp   0x96  ; 0x96 <_exit>

0000007c <__bad_interrupt>:
  7c: 0c 94 00 00       jmp   0     ; 0x0 <__vectors>

00000080 <main>:
  80: 80 e2             ldi   r24, 0x20   ; 32
  82: 84 b9             out   0x04, r24   ; 4
  84: 1a b8             out   0x0a, r1    ; 10
  86: 80 e8             ldi   r24, 0x80   ; 128
  88: 8b b9             out   0x0b, r24   ; 11
  8a: 4f 9b             sbis  0x09, 7     ; 9
  8c: 02 c0             rjmp  .+4         ; 0x92 <main+0x12>
  8e: 2d 9a             sbi   0x05, 5     ; 5
  90: fc cf             rjmp  .-8         ; 0x8a <main+0xa>
  92: 2d 98             cbi   0x05, 5     ; 5
  94: fa cf             rjmp  .-12        ; 0x8a <main+0xa>

00000096 <_exit>:
  96: f8 94             cli

00000098 <__stop_program>:
  98: ff cf             rjmp  .-2         ; 0x98 <__stop_program>
```

Interrupt vectors (we'll see later on)

Initialization code:
- make r1=0
- clear the status register (0x3F)
- set the stack pointer to 0x08FF
- call the main code
- call the exit code

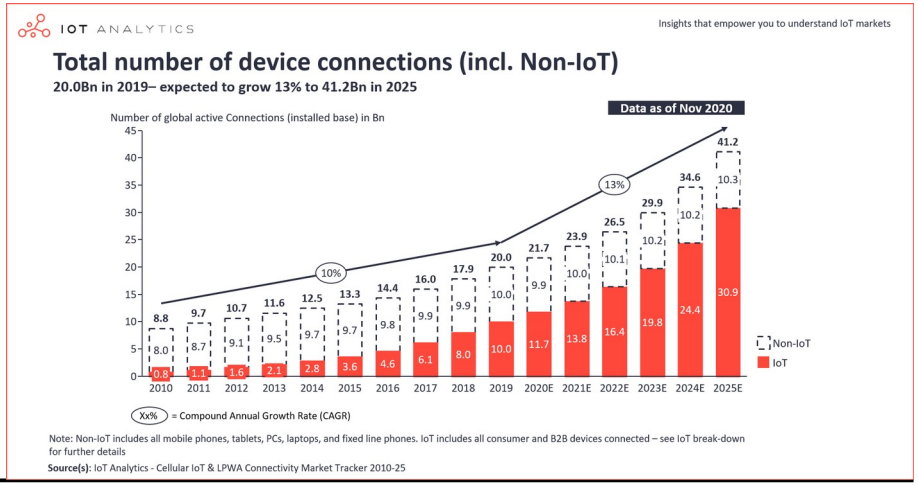
Bad interrupt handling code: just reset the processor

Main (user) code

Exit code: disable interrupts and block the processor.

Wait!
Why learning assembly?

- If we have C and C compilers, why learning assembly?
  - The academic reason: Software Engineers supposed to **fully** understand how software works.
  - The obvious reason: someone have to write the compilers.
    - Not only C, but any system-level language compiler: C++, Rust, etc.
  - The practical reason: some operating system code and C library functions are actually written in assembly (see _delay_ms() in avr-libc for example).
- Is assembly and low-level programming that useful?
  - Embedded systems and the IoT.
  - Operating systems.
  - Device drivers.
  - Game engines.
  - It's fun!



Where and why is assembly used?

- Only assembly.
  - In embedded systems that cannot be programmed otherwise.
  - Some very simple AVR MCUs, for example.
  - E.g.: tiny embedded systems, toaster, battery controller, light controller, etc.
- High-level programs like C and C++ calling assembly subroutines.
  - Maximum performance is needed.
  - Complete control of the hardware is needed.
  - Critical parts where exact timing is needed.
  - E.g.: compilers, operating systems, real-time code, device drivers, graphics engines, etc.

C language functions

Example 8

a) Write a C language function that adds up a list of 8-bit number in data memory. The function should add all the numbers up to the first one that is zero. The sum should be returned by the function as a 16-bit integer (type int).

b) Write a simple C program to test the function.

c) Take a look at the assembly code generated by the compiler and compare with the add list assembly version in example 4.

C language functions are the equivalent to subroutines in assembly and procedures or methods in other languages.

Functions in C may return a value of some type.

C language functions

```
// File: add_list.c
// Add bytes in a list up to the first zero

int add_list(char *data_list)
{
    int accum = 0;              // accumulator
    char val;                   // current value
    int i = 0;                  // index

    val = data_list[i];         // take first value
    while (val != 0) {          // accumulate as long as
        accum = accum + val;    // the value is not zero
        i++;
        val = data_list[i];
    }
    return accum;               // return the result
}
```

add_list function

compile with optimization

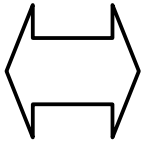
test code

```
// File: main.c
// Test add_list

#include <avr/io.h>

// Declare the add_list function
int add_list (char *);

int main(void)
{
    char data_list[] = {1, 2, 3, 5, 8, 0};  // test list to add
    int sum;                                // store the sum here
    DDRB = _BV(PB5);                        // set PB5 as output

    sum = add_list(data_list);              // sum the list

    if (sum == 19)                          // activate PB5 if the
        PORTB |= _BV(PB5);                  // sum is correct
    else
        PORTB &= ~_BV(PB5);
}
```

C language functions

```
avr-gcc -mmcu=atmega328p -Os -S -o add_list_Os.s add_list.c
```

```
      .file "add_list.c"
__SP_H__ = 0x3e
__SP_L__ = 0x3d
__SREG__ = 0x3f
__tmp_reg__ = 0
__zero_reg__ = 1
      .text
.global     add_list
      .type add_list, @function
add_list:
/* prologue: function */
/* frame size = 0 */
/* stack size = 0 */
.L__stack_usage = 0
      movw r30,r24
      ld r18,Z+
      ldi r24,0
      ldi r25,0
.L2:
      tst r18
      breq .L5
      add r24,r18
      adc r25,__zero_reg__
      sbrc r18,7
      dec r25
      ld r18,Z+
      rjmp .L2
.L5:
/* epilogue start */
      ret
      .size add_list, .-add_list
      .ident      "GCC: (GNU) 5.4.0"
```



```
; Arguments:    r25:r24 - address of the list.
; Return:       r25:r24 - result of the sum (16 bits).
; Description:  Add all bytes up to the first zero byte.

; Read macros with port numbers (and more)
#include <avr/io.h>

.text
.global add_list_w
add_list_w:
        ; use Z as pointer and r25:r24 as accumulator
        mov  zl, r24
        mov  zh, r25
        clr  r25
        clr  r24

        ; main loop
loop:
        ld   r0, z+
        tst  r0             ; test r0
        breq finish
        add  r24, r0
        adc  r25, r1        ; add carry to r25
        rjmp loop
finish:
        ret
```

The compiler did a good job!

AVR C calling convention
Application Binary Inrterface (ABI)

- Open questions when programming in assembly:
  - Which option is better for argument passing, registers or the stack?
  - Which registers may be used?
  - Will a subroutine overwrite my used registers?
  - Who is responsible for saving register values, caller or callee?
- The C compiler uses a function calling convention that answers these questions.
- The calling convention is part of the Application Binary Interface.
- It is a very good idea to use the C calling convention in our assembly programs in order to be compatible with C programs so that:
  - C programs can call our assembly subroutines.
  - Assembly programs can call C language library functions.

AVR C calling convention
AVR libc data types

- The C language does not define a fixed size for its basic data types.
- Knowing the type sizes is necessary to interact with C programs from assembly.
- Types in the AVR libc:
  - Character (`char`): 8 bits.
  - Integer (`int`): 16 bits.
  - Long integer (`long`): 32 bits.
  - Extra long integer (`long long`): 64 bits.
  - Floating point (`float` and `double`): 32 bits.
  - Pointers: 16 bits.
    - Pointers to functions contain addresses to 16-bit program memory words.

AVR C calling convention
Register usage

- r0: temporal register.
  - Can be used for intermediate calculations and discarded later.
- r1: zero register.
  - It is assume its value is always zero. Can be used temporarily but returned to zero afterwards (`clr r1`).
- r18-r25, x (r27:r26), z (r31:r30): caller-saved registers.
  - The caller is responsible of saving their values before calling a subroutines.
  - Subroutines can use these without having to restore their original values before returning.
- r2-r17, y (r29:r28): callee-saved registers.
  - The callee (subroutine being called) can use these registers but have to restore them to their initial value before returning.
  - The caller can use them without worrying about a subroutine changing their value.

AVR C calling convention
Argument passing and return value

- Function arguments are passed from left to right using registers r25 to r8 in little-endian format.
  - All arguments align to start in even registers. E.g. two 8-bit arguments will be passed in register r24 (first one) and r22 (second one).
  - Extra arguments are passed through the stack.
- Return value:
  - 8 bits (char) in r24
  - 16 bits (int) in r25:r24,
  - Up to 32 bits in r25:r22 and up to 64 bits in r25:r18.
  - 8-bit data are extended to 16 bits using zeros.
- Functions with variable number of arguments (e.g. printf)
  - All arguments are passed through the stack.
  - char arguments are extended to int.

Calling assembly subroutines from C

Example 9

a) Check if the add list subroutine in example 4 adheres to the C language calling convention. If not, modify the subroutine to make it compliant with the convention.

b) Write a simple C program to test that calling the subroutine from C actually works. You may adapt the test program in example 8.

Solution:

a) Yes, the add_list_w subroutine adheres to the C calling convention:

- Argument in r25:r24.
- Return value in r25:r24
- Use registers correctly: r0 (scratch register), r24, r25, z (r31:r30).

b) Just use add_list_w() instead of add_list().

Calling assembly subroutines from C

Example 10

We want to make an LED to blink for 0.1s every second. Write a C program for the AVR to do the task.

a) Use the _delay_ms() function from the C library to control the time.

b) Write another version that uses the delay_ms assembly subroutine of example 5 to control the time.

c) (Optional) Disassemble both versions and compare the generated code.

d) (Optional) Find the source code of the _delay_ms() function in the avr-libc distribution. Compare to the delay_ms assembly subroutine.

Calling C functions from assembly

Example 11

Write a subroutine that adds the absolute values of a list of 16-bit words in data memory. Use the abs function in the standard C library to calculate the absolute value. Write a test program for the subroutine.

```
; Subroutine:   add_list_abs
;
; Arguments:    r25:r24 - address of the list.
;               r22     - number of elements.
; Return:       r25:r24 - result of the sum (16 bits).
; Description:  Adds the absolute value of all the
;               elements in the list.
;               Returns 0 if the number of elements is zero.
;               Uses the abs(int) function from the C library.

; Read macros with port numbers (and more)
#include <avr/io.h>

.text

.global add_list_abs
add_list_abs:
        tst r22
        brne init
        clr r25
        clr r24
        ret
init:
        ; use Z as pointer and
        ; r19:r18 as accumulator
        mov  zl, r24
        mov  zh, r25
        clr  r19
        clr  r18
        ; main loop: read data from the list,
        ; calculate abs, accumulate and repeat.
loop:
        ld r24, z+
        ld r25, z+
        call abs
        add r18, r24
        adc r19, r25
        subi r22, 1
        brne loop

        ; save result and return
        movw r24, r18
        ret
```

GCC will automatically link to the C library and find the “abs” function.

Interrupts

- Interrupts are activated by peripherals or external signals.
- Interrupts cause the processor to jump to a service routine depending on the interrupt line activated.
  - Similar to a “call” but triggered by hardware.
  - Service routines return with “reti” instruction.
- Interrupts allow doing things “just in time” and avoid continuous “polling”.
  - An input bit in a digital port changes.
  - Data arrives to a serial port.
  - A timer has finished, etc.
- While not “interrupted” the processor can (should) be in a “sleep” mode, not executing instructions, and saving power.
  - Most real-life programs execute most or all of their functions in interrupt service routines.
  - Essential in battery-powered systems.

Interrupts

- Interrupts can be selectively enabled or disabled.
- The “I” flag in the status register enables (1) or disables (0) interrupts globally. Use SEI instruction to enable and CLI instruction to disable.
- Most peripherals may generate interrupts and there are also external interrupt lines.
- A peripheral may generate an interrupt when configured to do so.
  - Check the peripheral’s control registers for bits that enable interrupts.

Interrupt vectors

- An interrupt activation will make the processor to jump to an interrupt vector.
  - The specific executed interrupt vector depends on the source of the interrupt .
- Interrupt vectors are located at the beginning of program memory.
- Each vector uses two word addresses in the ATmega328P.
- Each vector will typically contain a “jmp” instruction to the interrupt services routine. This way:
  - Interrupt service routines can be easily defined.
  - Interrupts can be easily disabled by software.
- The GNU GCC compiler will create the interrupt vector in program memory automatically.
  - The programmer just have to define the interrupt vector destination routines using standard labels: “<interrupt_source_name>_vect:”

Interrupt vectors

Table 12-6. Reset and Interrupt Vectors in ATmega328 and ATmega328P

VectorNo.	Program Address(2)	Source	Interrupt Definition
1	0x0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2_COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2_COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2_OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1_CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1_COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1_COMPB	Timer/Counter1 Compare Match B
14	0x001A	TIMER1_OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0_COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0_COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0_OVF	Timer/Counter0 Overflow
18	0x0022	SPI_STC	SPI Serial Transfer Complete
19	0x0024	USART_RX	USART Rx Complete
20	0x0026	USART_UDRE	USART, Data Register Empty
21	0x0028	USART_TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE_READY	EEPROM Ready
24	0x002E	ANALOG_COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM_Ready	Store Program Memory Ready

Sleep modes

- The processor core is the most energy consuming device in the MCU.
- Ideally, the processor core should be stopped while there is nothing useful to do (just wait for interrupts).
  - It is essential in battery-powered devices.
- The SLEEP instruction will stop the processor and it will be waken up when necessary (an interrupt occurs).
- Sleep mode have to be enabled for the SLEEP instruction to do something useful.
- The AVR has different sleep modes, with different power savings and wake up conditions.
- In addition, unused peripherals can be shut down to save even more power by programming the Power Reduction Register (PRR).

Sleep modes

SMCR – Sleep Mode Control Register

The Sleep Mode Control Register contains control bits for power management.

Bit	7	6	5	4	3	2	1	0	
0x33 (0x53)	–	–	–	–	SM2	SM1	SM0	SE	SMCR
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits [7:4]: Reserved**

These bits are unused in the ATmega48A/PA/88A/PA/168A/PA/328/P, and will always be read as zero.

- **Bits 3:1 – SM[2:0]: Sleep Mode Select Bits 2, 1, and 0**

These bits select between the five available sleep modes as shown in Table 10-2.

Table 10-2. Sleep Mode Select

SM2	SM1	SM0	Sleep Mode
0	0	0	Idle
0	0	1	ADC Noise Reduction
0	1	0	Power-down
0	1	1	Power-save
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Standby(1)
1	1	1	External Standby(1)

Note: 1. Standby mode is only recommended for use with external crystals or resonators.

Typical “sleepy” main loop

```
    ; Main program
    sei                          ; enable interrupts
    ldi r16, _BV(SE)             ; enable "idle"
    out _SFR_IO_ADDR(SMCR), r16 ; sleep mode
loop:
    sleep                        ; wait for interrupts
    rjmp loop
```

Sleep modes

Table 10-1. Active Clock Domains and Wake-up Sources in the Different Sleep Modes

	Active Clock Domains					Oscillators		Wake-up Sources								
	clk_{CPU}	clk_{FLASH}	clk_{IO}	clk_{ADC}	clk_{ASY}	Main Clock Source Enabled	Timer Oscillator Enabled	INT1, INT0 and Pin Change	TWI Address Match	Timer2	SPM/EEPROM Ready	ADC	WDT	Other I/O	Software BOD Disable
Idle			X	X	X	X	X(2)	X	X	X	X	X	X	X	
ADC Noise Reduction				X	X	X	X(2)	X	X	X(2)	X	X	X		
Power-down								X	X				X		X
Power-save					X		X(2)	X	X	X			X		X
Standby(1)						X		X	X				X		X
Extended Standby					X(2)	X	X(2)	X	X	X			X		X

Notes: 1. Only recommended with external crystal or resonator selected as clock source.
2. If Timer/Counter2 is running in asynchronous mode.

Interrupt on pin change (PCINT)

- When enabled, generate interrupts on MCU pin change.
- Pin change signals are shared with other pins of the MCU.
  - E.g.: PCINT[7:0] are PB[7:0] in the ATmega328P
- 3 interrupt vectors cover 24 pin change signals:
  - Vector PCINT2: PCINT[23:16]
  - Vector PCINT1: PCINT[15:8]
  - Vector PCINT0: PCINT[7:0]
- Procedure
  - Write an interrupt service routine for the desired vector.
  - Enable the required interrupt in the PCICR.
  - Select the active pins with the PCMSK[2:0] registers.
  - Make sure global interrupts are enabled (SEI).
  - Alternatively, check if a PCINT has been requested in the PCIFR

Pin change control registers

PCICR – Pin Change Interrupt Control Register

Bit	7	6	5	4	3	2	1	0	
(0x68)	–	–	–	–	–	PCIE2	PCIE1	PCIE0	PCICR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Enable this to enable vector PCINT0. PCINT[7:0] = PB[7:0]

- **Bit 0 – PCIE0: Pin Change Interrupt Enable 0**

When the PCIE0 bit is set (one) and the I-bit in the Status Register (SREG) is set (one), pin change interrupt 0 is enabled. Any change on any enabled PCINT[7:0] pin will cause an interrupt. The corresponding interrupt of Pin Change Interrupt Request is executed from the PCI0 Interrupt Vector. PCINT[7:0] pins are enabled individually by the PCMSK0 Register.

PCMSK0 – Pin Change Mask Register 0

Bit	7	6	5	4	3	2	1	0	
(0x6B)	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	PCMSK0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Enable this to trigger PCINT0 vector on PCINT1=PB1 change.

- **Bit 7:0 – PCINT[7:0]: Pin Change Enable Mask 7...0**

Each PCINT[7:0] bit selects whether pin change interrupt is enabled on the corresponding I/O pin. If PCINT[7:0] is set and the PCIE0 bit in PCICR is set, pin change interrupt is enabled on the corresponding I/O pin. If PCINT[7:0] is cleared, pin change interrupt on the corresponding I/O pin is disabled.

Pin change control registers

PCIFR – Pin Change Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x1B (0x3B)	–	–	–	–	–	PCIF2	PCIF1	PCIF0	PCIFR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 0 – PCIF0: Pin Change Interrupt Flag 0**

When a logic change on any PCINT[7:0] pin triggers an interrupt request, PCIF0 becomes set (one). If the I-bit in SREG and the PCIE0 bit in PCICR are set (one), the MCU will jump to the corresponding Interrupt Vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it.

This is enabled whenever PCINT0 is requested, even if PCINT0 interrupt is not enabled.

Interrupt on pin change

Example 12

Activate PB5 whenever PB0 is activated using pin change interrupts.

Activate PB1 while the MCU is not sleeping (just to check sleep mode).

```
#include <avr/io.h>
.text
.global main
main:
    ; PCINT0 configuration:
    ldi r16, _BV(PCIE0)             ; enable PCINT0 interrupt
    sts PCICR, r16
    ldi r16, _BV(PCINT0)            ; enable interrupt on
    sts PCMSK0, r16                 ; PCINT0 (PB0) change

    ; PORTB configuration
    ldi r16, _BV(PB5) | _BV(PB1)    ; PB1 and PB5 as output
    out _SFR_IO_ADDR(DDRB), r16     ; rest of PORTB as input
    ldi r16, ~_BV(PB5)              ; activate pull-ups
    out _SFR_IO_ADDR(PORTB), r16

    ; Main program
    sei                             ; enable interrupts globally
    ldi r16, _BV(SM1) | _BV(SE)     ; enable "power-down"
    out _SFR_IO_ADDR(SMCR), r16     ; sleep mode
loop:
    cbi _SFR_IO_ADDR(PORTB), PB1    ; clear PB1 before sleep
    sleep                           ; wait for interrupts
    sbi _SFR_IO_ADDR(PORTB), PB1    ; set PB1 at wake-up
    rjmp loop
```

```
;
; PCINT0 interrupt handler
;
.global PCINT0_vect
PCINT0_vect:
    ; set PB5 to PB0
    sbis _SFR_IO_ADDR(PINB), PB0
    rjmp int_clear_output
    sbi _SFR_IO_ADDR(PORTB), PB5
    rjmp int_continue
int_clear_output:
    cbi _SFR_IO_ADDR(PORTB), PB5
int_continue:
    reti
```

"Power-down" mode is good here because it switches off most devices.

AVR Timers

- AVR timers applications:
  - Measure time.
  - PWM generation.
  - Execute periodic tasks (using interrupts).
  - ...
- Timer/Counter1 in the AVR has the following registers:
  - Control registers: ~~TCCR1A~~, TCCR1B, ~~TCCR1C~~.
  - Timer/Counter registers: TCNT1(H/L).
  - Output compare registers: OCR1A(H/L), ~~OCR1B(H/L)~~.
  - Input capture register: ~~ICR1(H/L)~~.
  - Interrupt mask register: TIMSK1.
  - Interrupt flag register: TIFR1

Timer/Counter 1

TCCR1B – Timer/Counter1 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit 3 – WGM12: Waveform Generation Mode
Used together with WGM11:0 in TCCR1A and WGM13. When all the other bits are 0:

- WGM12=0: Normal mode. The counter counts the full range.
- WGM12=1: CTC mode. The counter is cleared when it matches the compare register (Clear Timer on Compare match).

Table 16-5. Clock Select Bit Description

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

Timer/Counter 1

TCNT1H and TCNT1L – Timer/Counter1

Bit	7	6	5	4	3	2	1	0	
(0x85)	TCNT1[15:8]								TCNT1H
(0x84)	TCNT1[7:0]								TCNT1L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The two *Timer/Counter* I/O locations (TCNT1H and TCNT1L, combined TCNT1) give direct access, both for read and for write operations, to the Timer/Counter unit 16-bit counter. To ensure that both the high and low bytes are read and written simultaneously when the CPU accesses these registers, the access is performed using an 8-bit temporary High Byte Register (TEMP). This temporary register is shared by all the other 16-bit registers. See ”Accessing 16-bit Registers” on page 122.

OCR1AH and OCR1AL – Output Compare Register 1 A

Bit	7	6	5	4	3	2	1	0	
(0x89)	OCR1A[15:8]								OCR1AH
(0x88)	OCR1A[7:0]								OCR1AL
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The Output Compare Registers contain a 16-bit value that is continuously compared with the counter value (TCNT1). A match can be used to generate an Output Compare interrupt, or to generate a waveform output on the OC1x pin.

Timer/Counter 1

TIMSK1 – Timer/Counter1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 1 – OCIE1A: Timer/Counter1, Output Compare A Match Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Output Compare A Match interrupt is enabled. The corresponding Interrupt Vector (see "Interrupts" on page 66) is executed when the OCF1A Flag, located in TIFR1, is set.

- **Bit 0 – TOIE1: Timer/Counter1, Overflow Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Overflow interrupt is enabled. The corresponding Interrupt Vector (See "Interrupts" on page 66) is executed when the TOV1 Flag, located in TIFR1, is set.

TIFR1 – Timer/Counter1 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x16 (0x36)	–	–	ICF1	–	–	OCF1B	OCF1A	TOV1	TIFR1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 1 – OCF1A: Timer/Counter1, Output Compare A Match Flag**

This flag is set in the timer clock cycle after the counter (TCNT1) value matches the Output Compare Register A (OCR1A).

OCF1A is automatically cleared when the Output Compare Match A Interrupt Vector is executed. Alternatively, OCF1A can be cleared by writing a logic one to its bit location.

- **Bit 0 – TOV1: Timer/Counter1, Overflow Flag**

The setting of this flag is dependent of the WGM13:0 bits setting. In Normal and CTC modes, the TOV1 Flag is set when the timer overflows. Refer to Table 16-4 on page 141 for the TOV1 Flag behavior when using another WGM13:0 bit setting.

TOV1 is automatically cleared when the Timer/Counter1 Overflow Interrupt Vector is executed. Alternatively, TOV1 can be cleared by writing a logic one to its bit location.

Timer/Counter 1
Example: periodic interrupts

- An interrupt may be executed when the timer counter TCNT1 matches the output compare register OCR1A.
- The timer counter may be reset afterwards.
- Periodic interrupts procedure with Timer/Counter 1:
  - Configure CTC mode: reset timer after counter match.
    - Set WGM12 in TCCR1B.
  - Select the prescaler factor depending on the time scale you need.
    - Define CS12, CS11 and CS10 in TCCR1B.
  - Calculate the OCR1A value for the required period/frequency.
    - Load OCR1AH and OCR1AL in that order.
  - Enable compare A match.
    - Set OCIE1A in TIMSK1.
  - Enable interrupts globally.

Timer/Counter 1

Example 13

Blink a LED every 250ms using interrupts.

Modify it to blink only when a button connected to PB0 is pressed (PB0 = 0).

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

; Define the board's CPU frequency in Hz
.equ F_CLK, 16000000

; Blinking duration in milliseconds
DELAY = 250

; Calculated end-of-count counter (OCR1A) value
; The timer will runt at F_CLK/1024
EOCVAL = F_CLK/1024/2/1000*DELAY

.text

; Timer1 interrupt vector, A data compare match
; It just invert the value of PB5
; (is there an easier way to do it?)
.global TIMER1_COMPA_vect
TIMER1_COMPA_vect:
    sbis _SFR_IO_ADDR(PORTB), PB5   ; skip if PB5 is 1
    rjmp set_output
    cbi _SFR_IO_ADDR(PORTB), PB5    ; set PB5 to 0
    reti
set_output:                         ; set PB5 to 1
    sbi _SFR_IO_ADDR(PORTB), PB5
    reti
```

```
.global main
main:
    ; TIMER1 configuration:
    ;     - CTC mode: WGM12=1
    ;     - Timer frequency is F_CLK/1024: CS1[2:0] = 0b101
    ldi r16, _BV(WGM12) | _BV(CS12) | _BV(CS10)
    sts TCCR1B, r16               ; TCCR1B = 0b00001101
    ldi r16, hi8(EOCVAL)          ; load end-of-count register
    sts OCR1AH, r16               ; OCR1AH always first
    ldi r16, lo8(EOCVAL)
    sts OCR1AL, r16
    ldi r16, _BV(OCIE1A)          ; enable interrupt when
    sts TIMSK1, r16               ; compare A matches

    ; PORTB configuration
    ldi r16, _BV(PB5)             ; PB5 as output
    out _SFR_IO_ADDR(DDRB), r16   ; rest of PORTB as input
    ldi r16, ~_BV(PB5)            ; pull-ups active in all pins
    out _SFR_IO_ADDR(PORTB), r16  ; except PB5

    ; Main loop
    sei                           ; enable interrupts globally
    ldi r16, _BV(SE)              ; enable "idle" sleep mode
    out _SFR_IO_ADDR(SMCR), r16
loop:
    sleep                         ; wait for interrupts
    rjmp loop
```

Why not "power-down" mode?

Other peripherals