Binary and hexadecimal numbers

Binary - is a number expressed in the base-2 numeral system.

- Counting with 4-bit binary numbers from

conventional counting to -----> binary

0 ----------> 0000
1 ----------> 0001
2 ----------> 0010
3 ----------> 0011
4 ----------> 0100
5 ----------> 0101
6 ----------> 0110
7 ----------> 0111
8 ----------> 1000
9 ----------> 1001
10 ----------> 1010
11 ----------> 1011
12 ----------> 1100
13 ----------> 1101
14 ----------> 1110
15 ----------> 1111

Hexadecimal numbers: is a number expressed in the base-16 numeral system.

- Counting with hexadecimal numbers

conventional counting to -----> hexadecimal

0 ----------> 0x0
1 ----------> 0x1
2 ----------> 0x2
3 ----------> 0x3
4 ----------> 0x4
5 ----------> 0x5
6 ----------> 0x6
7 ----------> 0x7
8 ----------> 0x8
9 ----------> 0x9
10 ----------> 0xA
11 ----------> 0xB
12 ----------> 0xC
13 ----------> 0xD
14 ----------> 0xE
15 ----------> 0xF

If we want to represent the numbers 16 and above, we will need two hexadecimal numbers

16----------> 0x10
17----------> 0x11
18----------> 0x12
19----------> 0x13
20----------> 0x14
22----------> 0x15
23----------> 0x16
24----------> 0x17
25----------> 0x18
26----------> 0x19
27----------> 0x1A
28----------> 0x1B
29----------> 0x1C
30----------> 0x1D
31----------> 0x1E
32----------> 0x1F

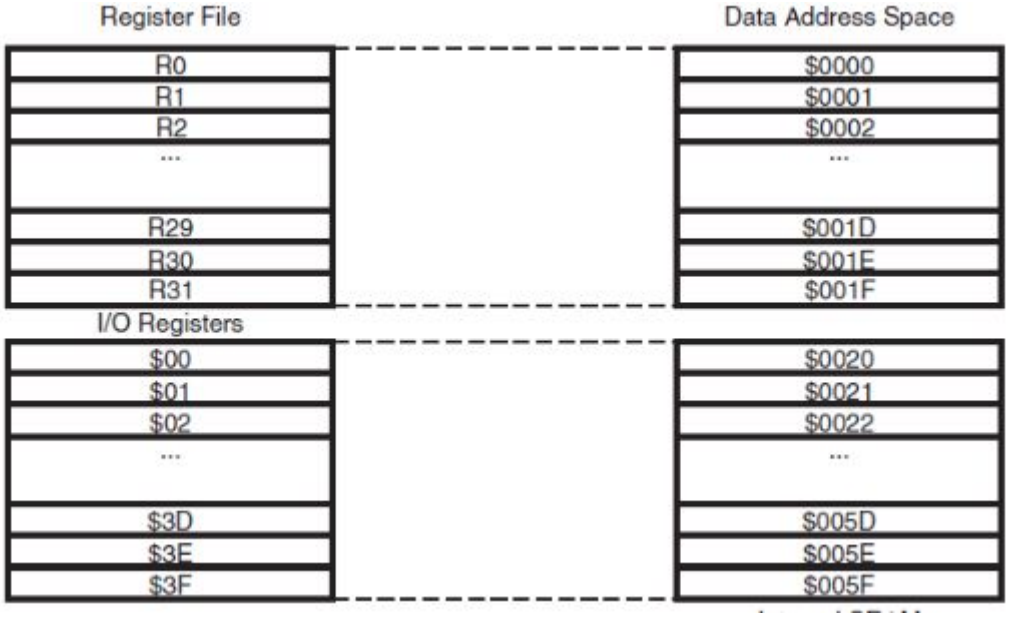
Converting from Binary to hexadecimal numbers

´-Since we need 4bits to represent 16 VALUES (0-15) by using, we can also change 4 bits to represent a hex number.

Hexadecimal counting to -----> binary

0x0 ----------> 0000
0x1 ----------> 0001
0x2 ----------> 0010
0x3 ----------> 0011
0x4 ----------> 0100
0x5 ----------> 0101
0x6 ----------> 0110
0x7 ----------> 0111
0x8 ----------> 1000
0x9 ----------> 1001
0xA----------> 1010
0xB----------> 1011
0xC----------> 1100
0xD----------> 1101
0xE----------> 1110
0xF----------> 1111

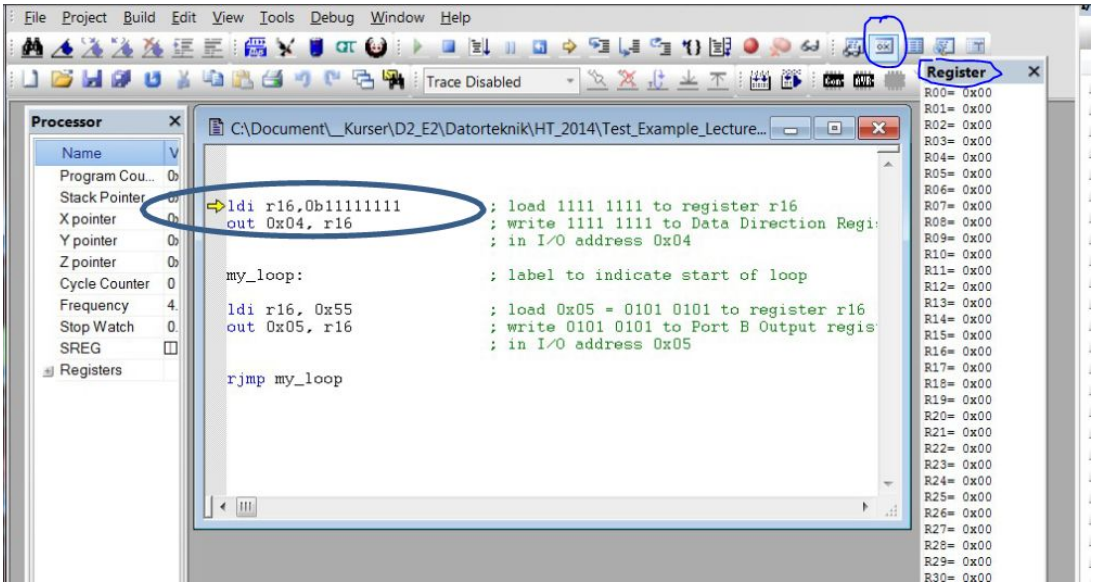
What is the connection between binary or hex and computers



All computers store their information in bits on the memory, and the processor does all its computations on bits. Hex numbers are easier to read and use, so we often use them.

Data Memory, I/O Registers, Registers

Registers: are small memories that we use to store different kinds of temporary values. The are several registers in the microprocessor like EEPROM registers, EEDR – The EEPROM Data Register, EECR – The EEPROM Control Register and more that are used for different purposes. We will see most of them throughout the course, However, for now, there are 32 8-bit registers that we can use to temporarily store values. (see more on doc2549 ATmega2560)



I/O Registers: are input-output port registers that we use to handle signals that are coming from inside or going outside. We do this by defining the PORTS

Register Description for I/O Ports

Port A Data Register – PORTA

Bit	7	6	5	4	3	2	1	0	
	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	PORTA
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	

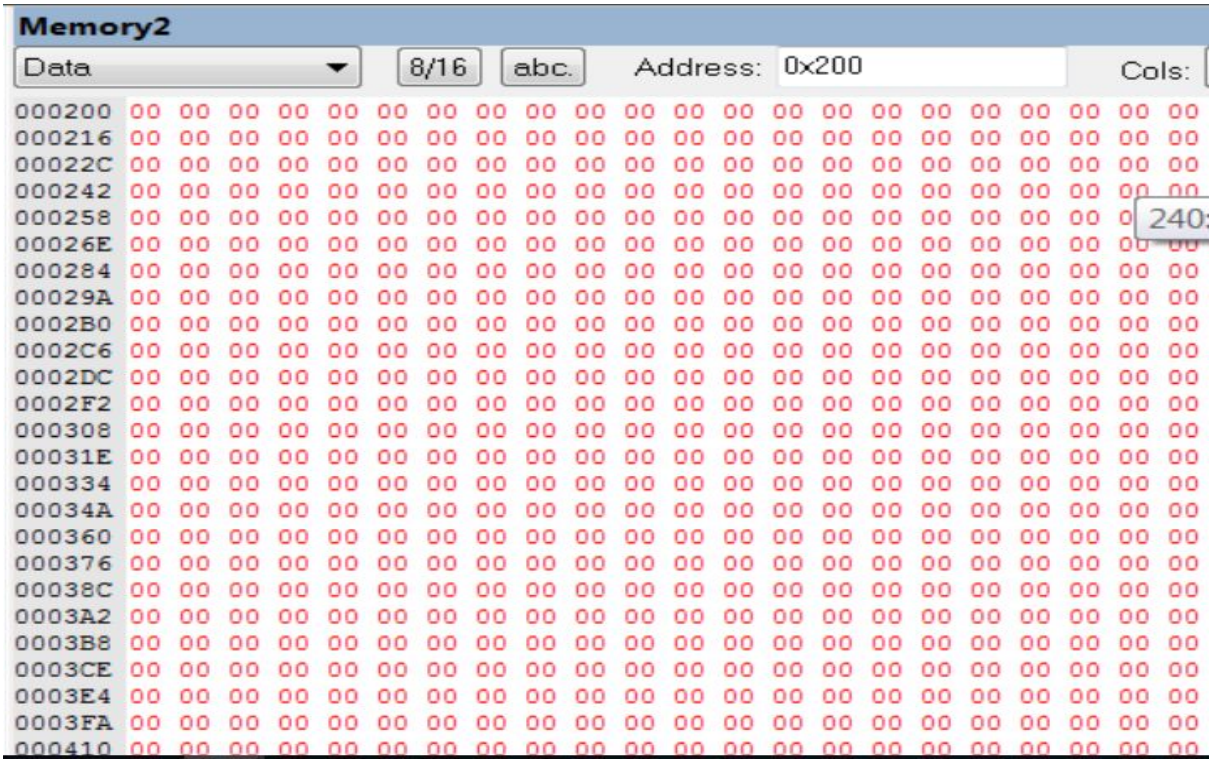
Port A Data Direction Register – DDRA

Bit	7	6	5	4	3	2	1	0	
	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	DDRA
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	

Port A Input Pins Address – PINA

Bit	7	6	5	4	3	2	1	0	
	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	PINA
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Data memory: Is the place all executable commands are physically stored on. The data memory is split into two, the internal part and the external part.(see more on doc2549 ATmega2560)



What is stack and stack pointer?

The Stack is mainly used for storing temporary data, for storing local variables and for storing return addresses after interrupts and subroutine calls.

We always use the lower part of our data memory for the purpose of the stack. We do this by defining the stack pointer to point to the lower end of the memory.

```
.include “m2560def.inc”                    ; Define names for ATmega2560
```

You can also write the correct address instead of the SPH, SPL and RAMEND as follows:

```
        ldi r20, HIGH(0x21FF)    ; R20 = high byte of RAMEND address, 0x21
        out 0x3E,R20             ; SPH = high part of pointer to STACK
        ldi R20, LOW(0x21FF)     ; R20 = low byte of RAMEND address, 0xFF
        out 0x3D,R20             ; SPL = low part of pointer to STACK
```

or like this:

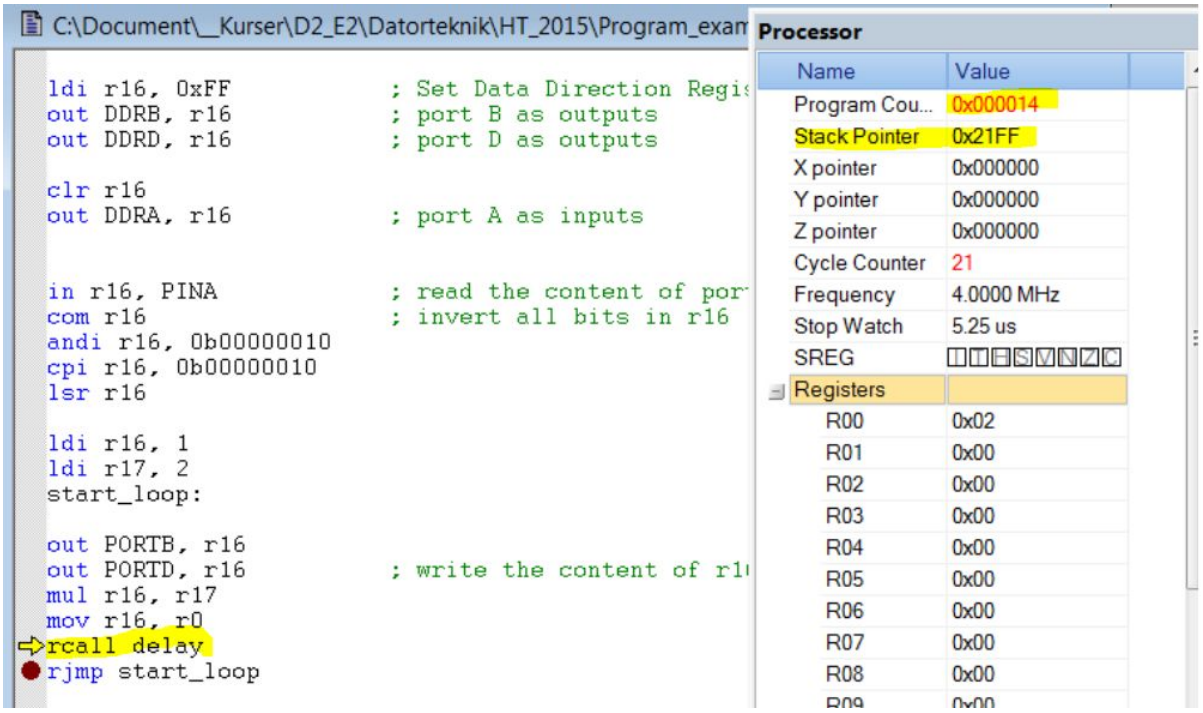
```
        ldi  r16, 0x21           ; r16 = high byte of RAMEND address, 0x21
        out  0x3E,r16            ; 0x3E = SPH = high part of pointer to STACK
        ldi  r16,0xFF            ; r16 = low byte of RAMEND address, 0xFF
        out  0x3D,r16            ; 0x3D = SPL = low part of pointer to STACK
```

Stackpointer

The Stack Pointer points to the data SRAM Stack area where the Subroutine and Interrupt Stacks are located. This Stack space in the data SRAM must be defined by the program before any subroutine calls are executed or interrupts are enabled. The Stack Pointer must be set to point above 0x0200. The initial value of the stack pointer is the last address of the internal SRAM. The Stack Pointer is decremented by one when data is pushed onto the Stack with the PUSH instruction, and it is decremented by two for ATmega640/1280/1281 and three for ATmega2560/2561 when the return address is pushed onto the Stack with subroutine call or interrupt. The Stack Pointer is incremented by one when data is popped from the Stack with the POP instruction, and it is incremented by two for ATmega640/1280/1281 and three for ATmega2560/2561 when data is popped from the Stack with return from subroutine RET or return from interrupt RETI.

The AVR Stack Pointer is implemented as two 8-bit registers in the I/O space. The number of bits actually used is implementation dependent. Note that the data space in some implementations of the AVR architecture is so small that only SPL is needed. In this case, the SPH Register will not be present.

The program counter (PC): is a counter that keeps track of the currently executed command's address.



The difference between jmp(rjmp)
And call(rcall)

JMP command: the way we use the jump command is “JMP K” where K is a specific address in the memory. This command will change the PC to be K. The difference between JMP and RJMP is that RJMP can jump from $-2K \leq k < 2K$ and can address the entire memory, Jump to an address within the entire 4M (words) Program memory. JMP instruction is not available in all devices

Other commands like BRBC, BRNE,BRMI and others also use the jump command if a certain condition is met.

JMP – Jump

Description:

Jump to an address within the entire 4M (words) Program memory. See also RJMP.

This instruction is not available in all devices. Refer to the device specific instruction set summary.

	Operation:
(i)	PC ← k

	Syntax:	Operands:	Program Counter:	Stack:
(i)	JMP k	0 ≤ k < 4M	PC ← k	Unchanged

32-bit Opcode:

1001	010k	kkkk	110k
kkkk	kkkk	kkkk	kkkk

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

RJMP – Relative Jump

Description:

Relative jump to an address within PC - 2K +1 and PC + 2K (words). In the assembler, labels are used instead of relative operands. For AVR microcontrollers with Program memory not exceeding 4K words (8K bytes) this instruction can address the entire memory from every address location.

Operation:

(i) PC ← PC + k + 1

	Syntax:	Operands:	Program Counter:	Stack
(i)	RJMP k	-2K ≤ k < 2K	PC ← PC + k + 1	Unchanged

16-bit Opcode:

1100	kkkk	kkkk	kkkk

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

CALL command: the way we use the call command is "CALL K" where K is a specific address in the memory. This command will change the PC to be K, however, the difference in CALL command is before it changes the PC to K, it first stores it in the stack. Then when it reaches a RET command it will fetch back the address from the stack and assigns it to the PC. CALL can address the entire memory from every address location. The Stack Pointer uses a post-decrement scheme during both CAL and RCALL.

CALL – Long Call to a Subroutine

Description:

Calls to a subroutine within the entire Program memory. The return address (to the instruction after the CALL) will be stored onto the Stack. (See also RCALL). The Stack Pointer uses a post-decrement scheme during CALL.

This instruction is not available in all devices. Refer to the device specific instruction set summary.

Operation:

(i) PC ← k Devices with 16 bits PC, 128K bytes Program memory maximum.
(ii) PC ← k Devices with 22 bits PC, 8M bytes Program memory maximum.

	Syntax:	Operands:	Program Counter	Stack:
(i)	CALL k	0 ≤ k < 64K	PC ← k	STACK ← PC+2 SP ← SP-2, (2 bytes, 16 bits)
(ii)	CALL k	0 ≤ k < 4M	PC ← k	STACK ← PC+2 SP ← SP-3 (3 bytes, 22 bits)

32-bit Opcode:

1001	010k	kkkk	111k
kkkk	kkkk	kkkk	kkkk

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

RCALL – Relative Call to Subroutine

Description:

Relative call to an address within PC - 2K + 1 and PC + 2K (words). The return address (the instruction after the RCALL) is stored onto the Stack. (See also CALL). In the assembler, labels are used instead of relative operands. For AVR microcontrollers with Program memory not exceeding 4K words (8K bytes) this instruction can address the entire memory from every address location. The Stack Pointer uses a post-decrement scheme during RCALL.

Operation:

(i) PC ← PC + k + 1 Devices with 16 bits PC, 128K bytes Program memory maximum.
(ii) PC ← PC + k + 1 Devices with 22 bits PC, 8M bytes Program memory maximum.

	Syntax:	Operands:	Program Counter:	Stack:
(i)	RCALL k	-2K ≤ k < 2K	PC ← PC + k + 1	STACK ← PC + 1 SP ← SP - 2 (2 bytes, 16 bits)
(ii)	RCALL k	-2K ≤ k < 2K	PC ← PC + k + 1	STACK ← PC + 1 SP ← SP - 3 (3 bytes, 22 bits)

16-bit Opcode:

1101	kkkk	kkkk	kkkk

Status Register (SREG) and Boolean Formula:

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

PUSH and POP commands: we use PUSH to temporarily store a value in our stack, and we use POP when we want to fetch values from a stack. NOTE:- we have to be careful when using the stack. Let say if we PUSH r16 and PUSH r17 respectively, then we have to do the opposite when using POP-first we POP r17 then POP 16.

M2560def.inc file and Using the simulator tips

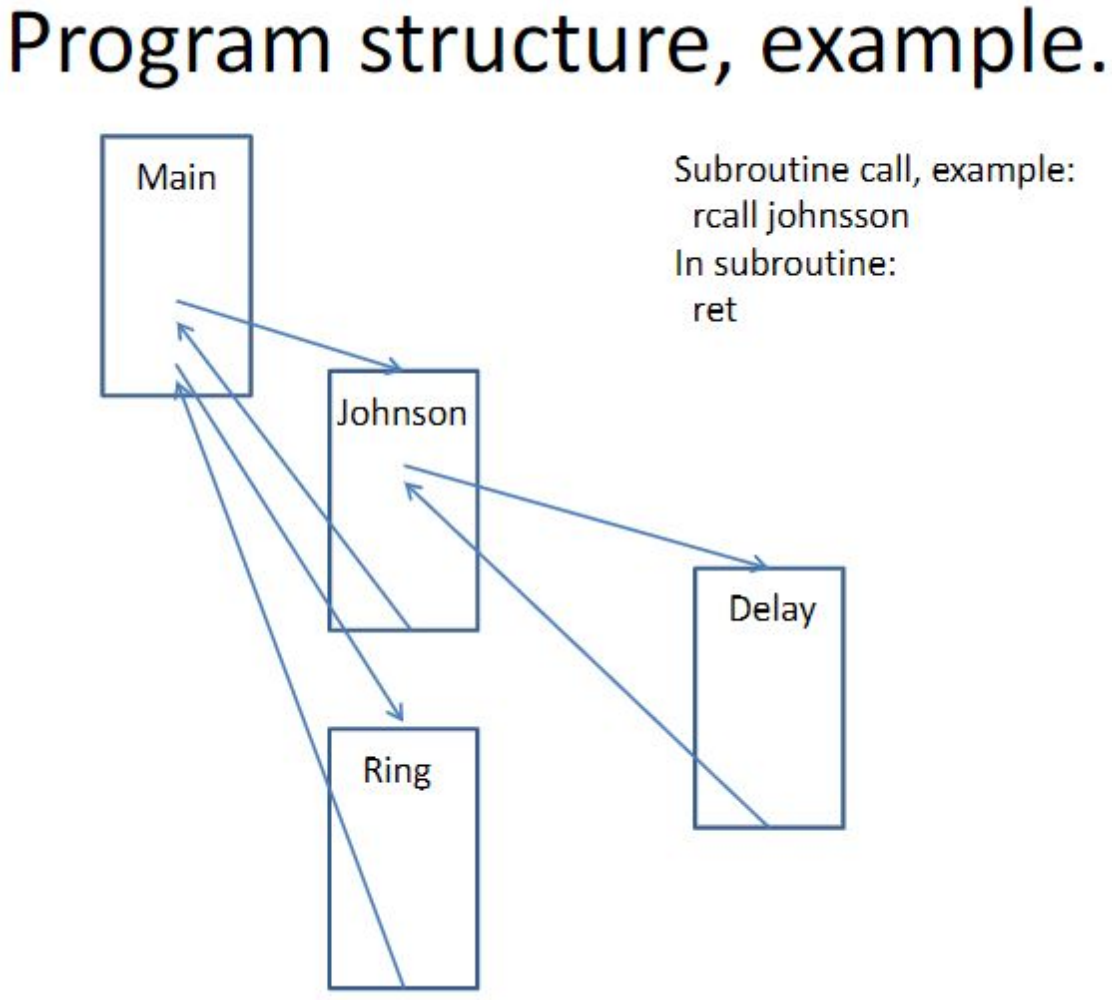
m2560def.inc : The file we always include on our code is set of already defined address in order for us to make it easier. We can see the file in the picture that it is assigning the address 0x07 to DDRC**.** **slide to dhs v**

```
.equ    PCIFR   = 0x1b
.equ    TIFR5   = 0x1a
.equ    TIFR4   = 0x19
.equ    TIFR3   = 0x18
.equ    TIFR2   = 0x17
.equ    TIFR1   = 0x16
.equ    TIFR0   = 0x15
.equ    PORTG   = 0x14
.equ    DDRG    = 0x13
.equ    PING    = 0x12
.equ    PORTF   = 0x11
.equ    DDRF    = 0x10
.equ    PINF    = 0x0f
.equ    PORTE   = 0x0e
.equ    DDRE    = 0x0d
.equ    PINE    = 0x0c
.equ    PORTD   = 0x0b
.equ    DDRD    = 0x0a
.equ    PIND    = 0x09
.equ    PORTC   = 0x08
.equ    DDRC    = 0x07
.equ    PINC    = 0x06
.equ    PORTB   = 0x05
.equ    DDRB    = 0x04
.equ    PINB    = 0x03
.equ    PORTA   = 0x02
.equ    DDRA    = 0x01
.equ    PINA    = 0x00
```

Tips for debug:

- By pressing F11 after building the project, you can see line by line how the commands are being executed.
- If you press F9 at a certain position in your code you can make a breakpoint on that code: The use of breakpoints is next time you press F5(run) it, it will run till the breakpoint

Program Structure and Subroutine usage



When we use a subroutine it is good practice to follow the following suggestions

- Give it a proper name
- Let it be possible to use RCALLto start the subroutine
- Before doing anything PUSH the registers you are going to use in the subroutine then POP them back when done
- Use RET command to go out when the subroutine finishes a task.

Other tips when doing the tasks divide your code into subroutines and let a subroutine handle one specific task.

How Lab and deadlines are handled

A	B	C	D	E
		First chance	Second chance	Third chance
Lab 1	Partner account	xxyyyxx		
	Task 1			
	Task 2			
	Task 3			
	Task 4			
	Task 5			
	Task 6			
Lab 2				
	Task 1			
	Task 2			
	Task 3			
	Task 4			
	Task 5			
	Task 6			