Sensors & Microsystem Electronics: microcontrollers

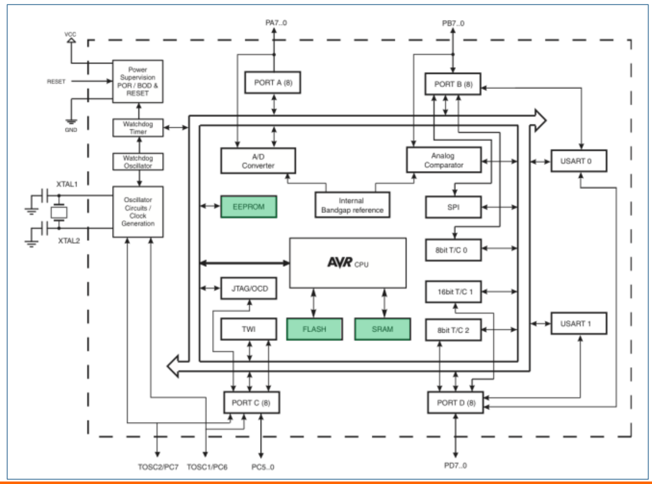
PART 2: MEMORY, TIMERS & INTERRUPTS

ETRO ELECTRONICS & INFORMATICS

Memory

Memory overview (Volatile vs Non-volatile)

- FLASH
  - 32 kilobytes
  - Program storage
  - non-volatile
  - 16-bit addresses
- SRAM
  - 2 kilobytes
  - Data memory
  - volatile
  - 16-bit addresses
- EEPROM
  - 1 kilobyte
  - Data memory
  - non-volatile
  - 16-bit addresses



The microcontroller has three different memories: FLASH, SRAM and EEPROM.

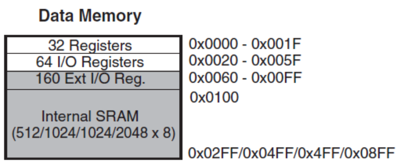
Both FLASH and EEPROM are non-volatile, meaning they keep their content between power cycles. This is not the case with the SRAM which is volatile.

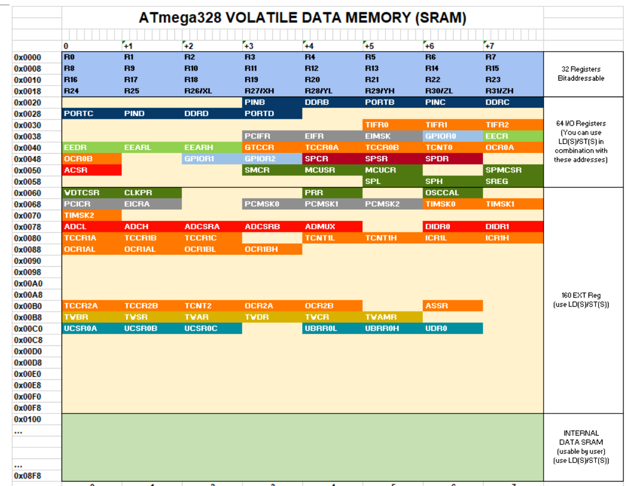
The flash memory is also called the program memory. This is because your compiled program is written here, and loaded when the microcontroller is running. It is not possible to write to this memory from your running program. Writing can only be done by the programmer. As a programmer you can store blocks of readonly data here to be read by your program.

The SRAM is your “working memory” this part of the memory can be used to temporarily store values. The SRAM has very fast access times and has a virtually unlimited number of read/write cycles.

The EEPROM can be written from your code as opposed to the FLASH. It is used to store values that need to be kept between power cycles. Some examples of this are: settings, highscores, … The EEPROM has slower access times, as well as a limited number of write-cycles (10k-100k)

SRAM / Data Memory



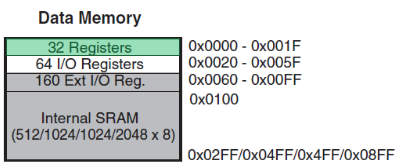


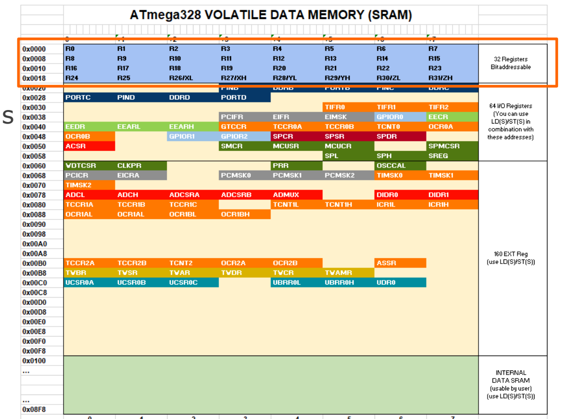
The SRAM or data memory consists of 4 parts:
The 32 registers on which one can do operations.
64 I/O registers. And 160 extended I/O registers. These registers are used to configure the peripherals of the microcontroller.
The internal SRAM: this is free memory space to be freely used by the programmer

Although the microcontroller is 8 bit and as such all registers are 8 bit, the memory is addressed using 16bit addresses. Otherwise the amount of memory space would be limited to 255 bytes. The 16bit addressing is done trough two 8 bit registers, one contains the upper 8 bit, and the other the lower 8 bit.

Registers

- Bitaddressable
- R16-R31 support LDI (load constant)
- Certain registers for specific instructions
  - See instruction set
- R27 – R31 double as pointer registers
- Not zeroed on reset





The SRAM or data memory consists of 4 parts:
The 32 registers on which one can do operations.
64 I/O registers. And 160 extended I/O registers. These registers are used to configure the peripherals of the microcontroller.
The internal SRAM: this is free memory space to be freely used by the programmer

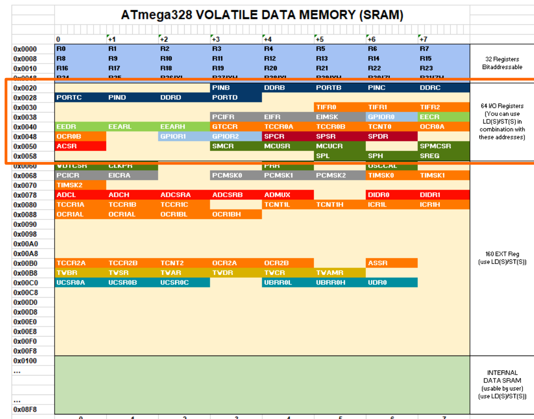
Although the microcontroller is 8 bit and as such all registers are 8 bit, the memory is addressed using 16bit addresses. Otherwise the amount of memory space would be limited to 255 bytes. The 16bit addressing is done trough two 8 bit registers, one contains the upper 8 bit, and the other the lower 8 bit.

I/O registers

- Only first 32 are bitaddressable
- To be used with **IN** and **OUT**
- Specific functions & peripheral config.
- Set to initial values on reset
  - See relevant datasheet sections

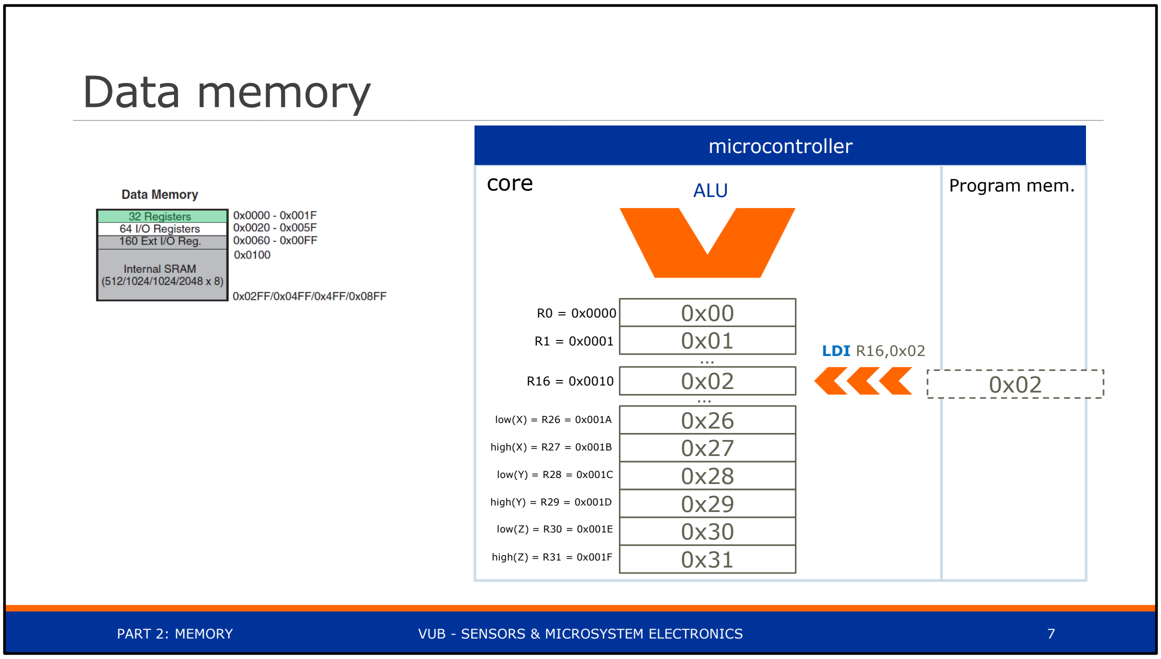
Data Memory

32 Registers	0x0000 - 0x001F
64 I/O Registers	0x0020 - 0x005F
160 Ext I/O Reg.	0x0060 - 0x00FF
Internal SRAM (512/1024/1024/2048 x 8)	0x0100
	0x02FF/0x04FF/0x4FF/0x08FF



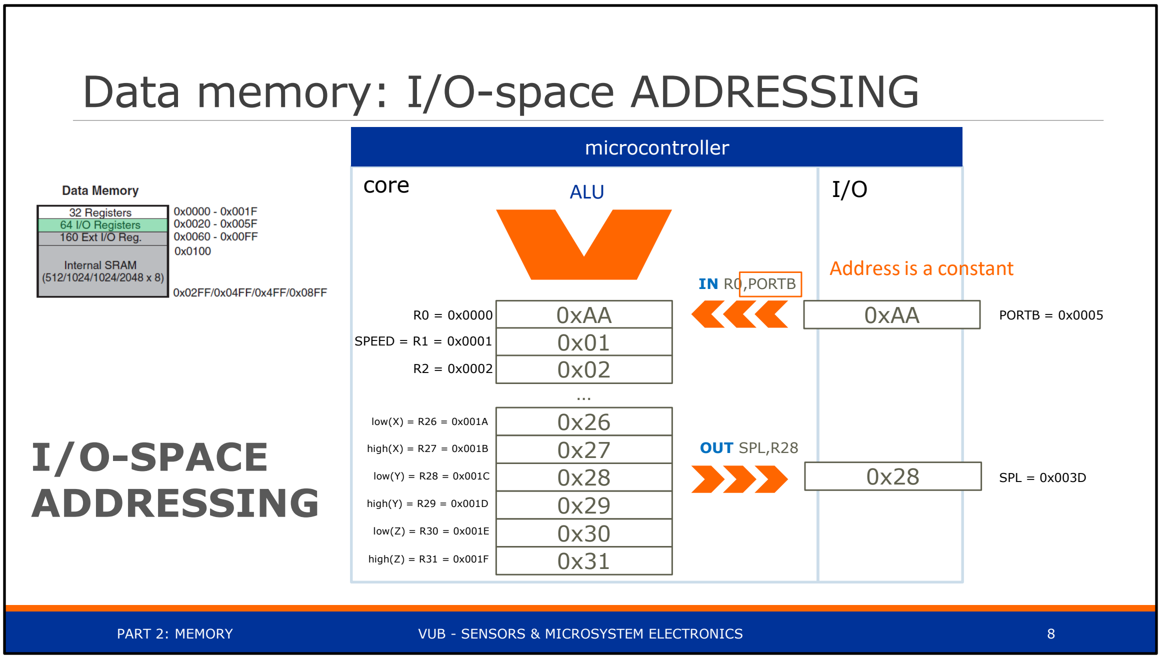
The SRAM or data memory consists of 4 parts:
The 32 registers on which one can do operations.
64 I/O registers. And 160 extended I/O registers. These registers are used to configure the peripherals of the microcontroller.
The internal SRAM: this is free memory space to be freely used by the programmer

Although the microcontroller is 8 bit and as such all registers are 8 bit, the memory is addressed using 16bit addresses. Otherwise the amount of memory space would be limited to 255 bytes. The 16bit addressing is done trough two 8 bit registers, one contains the upper 8 bit, and the other the lower 8 bit.



Only the first 32 registers are usable with instructions. Furthermore there can be some limitations on the instructions, e.g. Multiply stores the result in specific registers. As well as registers with additional functions such as the addressing registers.

The LDI instruction can load a constant in a registers. Like LDI R16,0x02 loads 0x02 in register 16. Note: the LDI instruction can only be used on R16-R31. To load a constant in any of the registers below 16, one needs to free up a register in the upper half, load it there with LDI, then copy it to the intended register, (and restore the temporary register again if necessary)

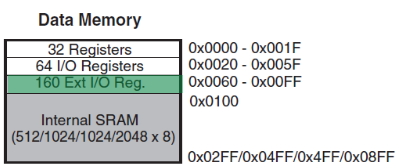


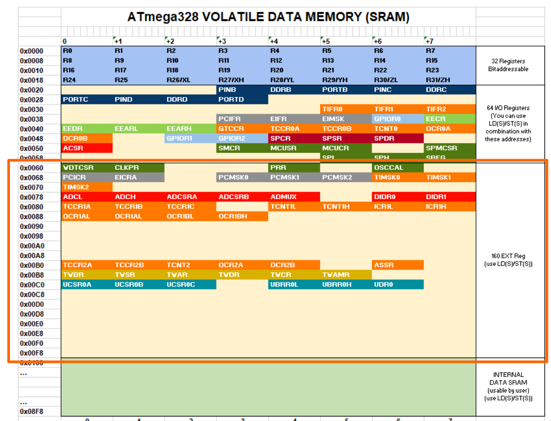
The IN and OUT instructions can move data from one of the 32 registers to the I/O registers.

The include file (atmega328.inc) defines a number of identifiers for certain registers. For any register in the 64 I/O registers, these identifiers are only valid to be used with IN and OUT. These registers can be accessed using LDS and STS, but then the identifiers are not valid.

Extended I/O registers

- Access with **LD(S)** and **ST(S)**
- Specific functions & peripheral config.
- Set to initial values on reset
  - See relevant datasheet sections



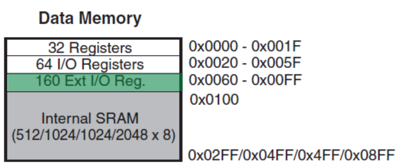


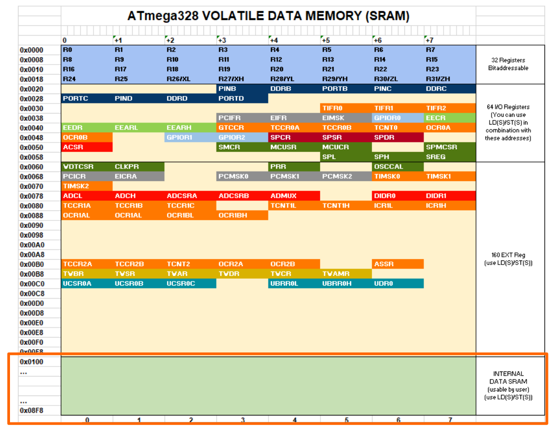
The SRAM or data memory consists of 4 parts:
The 32 registers on which one can do operations.
64 I/O registers. And 160 extended I/O registers. These registers are used to configure the peripherals of the microcontroller.
The internal SRAM: this is free memory space to be freely used by the programmer

Although the microcontroller is 8 bit and as such all registers are 8 bit, the memory is addressed using 16bit addresses. Otherwise the amount of memory space would be limited to 255 bytes. The 16bit addressing is done trough two 8 bit registers, one contains the upper 8 bit, and the other the lower 8 bit.

Internal SRAM

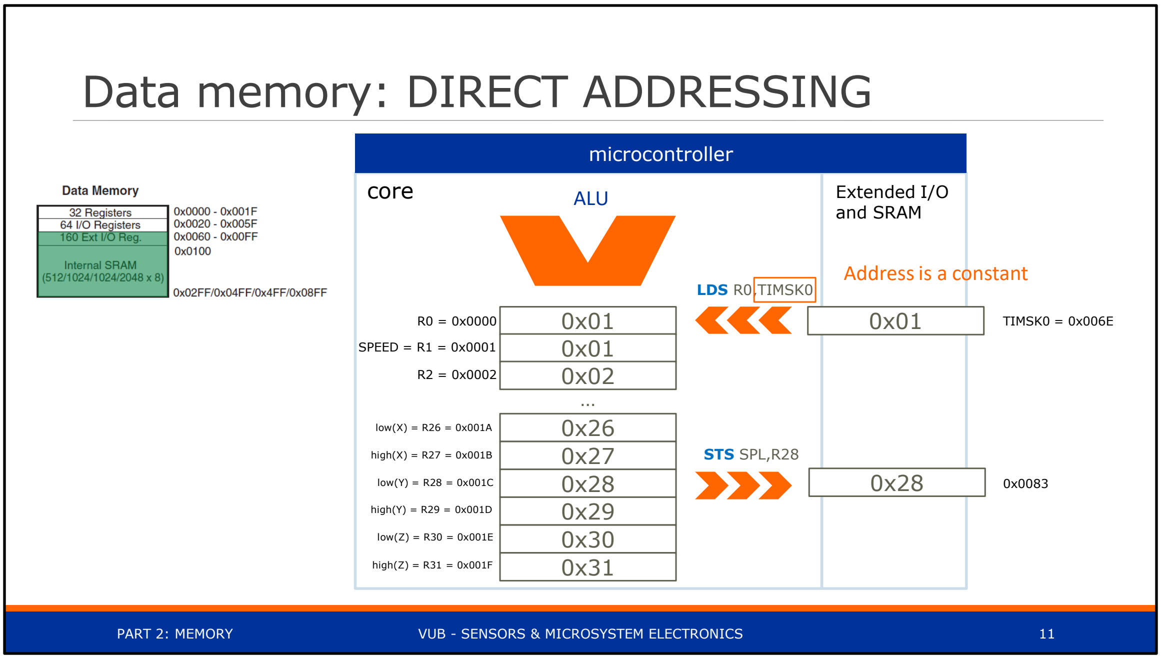
- Access with **LD(S)** and **ST(S)**
- Not zeroed on reset
- Completely usable by user
- Careful: stack is located at the end and counts backwards





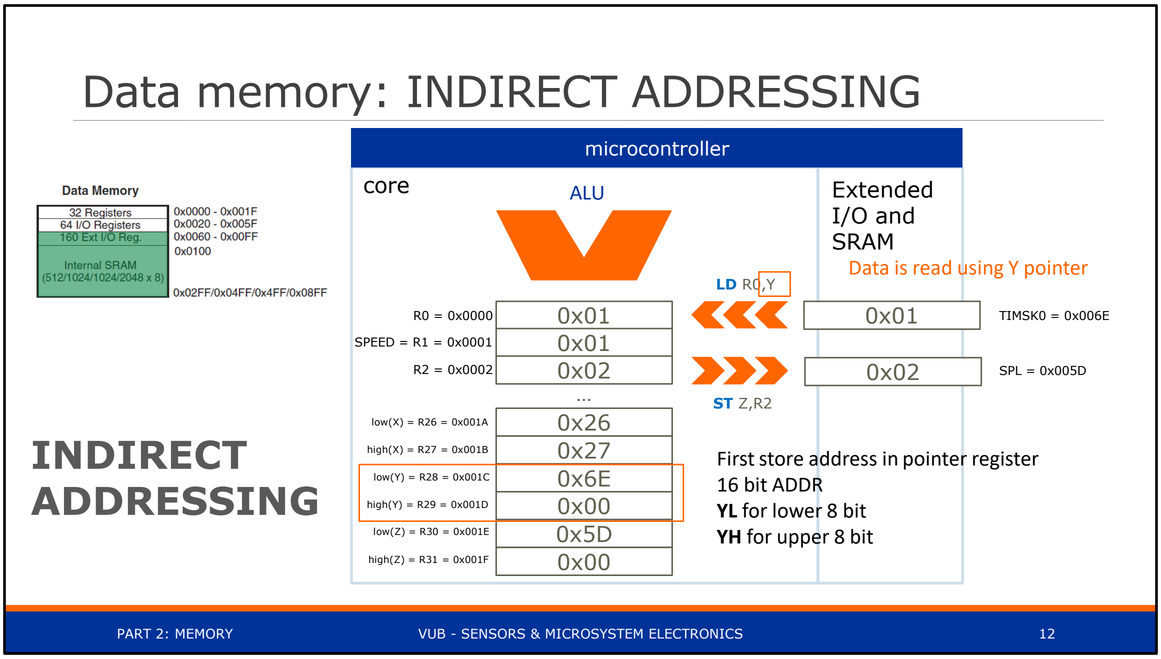
The SRAM or data memory consists of 4 parts:
The 32 registers on which one can do operations.
64 I/O registers. And 160 extended I/O registers. These registers are used to configure the peripherals of the microcontroller.
The internal SRAM: this is free memory space to be freely used by the programmer

Although the microcontroller is 8 bit and as such all registers are 8 bit, the memory is addressed using 16bit addresses. Otherwise the amount of memory space would be limited to 255 bytes. The 16bit addressing is done trough two 8 bit registers, one contains the upper 8 bit, and the other the lower 8 bit.



For the remaining part of the memory, the LDS/LD and STS/ST instructions are used. Two different methods of memory access exist: Direct and indirect.

The direct method is done trough supplying the memory address to the LDS/STS instruction.



The other method is the indirect method. Here one first stores the 16-bit memory address in one of the three memory addressing registers of pointer register (each consisting of 2 separate registers to achieve 16bit addressing). After this, the memory address the register points to can be accessed using LD and ST

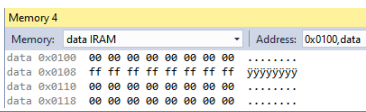
This method can be used to access consecutive memory locations or to calculate a certain memory address offset before using it to access that data.

Indirect addressing – Post-increment

```
;Code example: write 0xFF to
;addresses 0x108 to 0x10E in SRAM

;Set Y pointer to 0x108
LDI YH, high(0x0108);upper byte to YH (0x01)
LDI YL, low(0x0108);lower byte to YL (0x08)

LDI R20, 0xFF ;some data in a register
LDI R16, 8 ; loop iterator
Loop:
   ST Y+, R20; store R20 in the current
address
   DEC R16;decrement loop iterator
   BRNE Loop; end of loop
```



- First iteration (Y = 0x108):
  - ST writes R20 to 0x108
  - Y = Y + 1 after write
- 2nd iteration (Y = 0x109):
  - ST writes R20 to 0x109
  - Y = Y + 1 after write
- 3rd iteration (Y = 0x10A)

…

- 8th iteration (Y = 0x10E)
  - ST writes R20 to 0x109
  - Y = Y + 1 after write
- End of loop (Y = 0x110)

Here we have a look to the POST-Increment feature of the indirect addressing method. This is useful when reading or writing data from or to consecutive addresses.

First the Y pointer upper and lower byte is loaded with the right part of the address with the help of the HIGH and LOW directive. Next the value 0xFF is stored in R20 and a loop of 8 iterations with R16 as iterator is made.

By adding a plus after the Y pointer in the ST instruction line. The compiler knows to use the ST with Y and post-increment bytecode when compiling.

The first iteration will write the value from R20 to the address Y is pointing to, in this iteration this is still address 0x108 which is the first address of the user SRAM. After the write, Y will be incremented by one, now pointing to 0x109 (YH = 0x01 and YL = 0x09)

The second iteration will write the value from R20 to the address Y is currently pointing to, in this iteration this has become address 0x10A. After the write, Y will be incremented by one, now pointing to 0x10A (YH = 0x01 and YL = 0x0A)

After the eight iteration, the SRAM addresses 0x108 to 0x10E will contain 0xFF as value. Y will now point to 0x110 (YH = 0x01 and YL = 0x10). The figure shows a screenshot of the SRAM from the simulator displaying that the writing was successful.

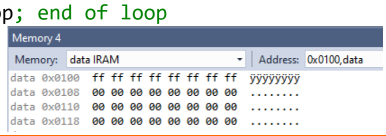
Note that when any lower pointer register overflows due to the increment, the carry will be added into the upper pointer register. This post-increment is supported by both the ST and LD instruction. The PRE-increment option does not exist in this instruction set.

Indirect addressing – Pre-decrement

```
;Code example: write 0xFF to
;addresses 0x108 to 0x100 in SRAM

;Set Y pointer to 0x108
LDI YH, high(0x0108);upper byte to YH (0x01)
LDI YL, low(0x0108);lower byte to YL (0x08)

LDI R20, 0xFF ;some data in a register
LDI R16, 8 ; loop iterator
Loop:
    ST -Y, R20; store R20 in the current
address
    DEC R16;decrement loop iterator
    BRNE Loop; end of loop
```



- First iteration (Y = 0x108):
  - Y = Y - 1 before write
  - ST writes R20 to 0x107
- 2nd iteration (Y = 0x107):
  - Y = Y - 1 before write
  - ST writes R20 to 0x106
- 3rd iteration (Y = 0x106)

...

- 8th iteration (Y = 0x101)
  - Y = Y - 1 before write
  - ST writes R20 to 0x100
- End of loop (Y = 0x100)

Here we have a look to the PRE-decrement feature of the indirect addressing method. This is useful when reading or writing data from or to consecutive addresses.

First the Y pointer upper and lower byte is loaded with the right part of the address with the help of the HIGH and LOW directive. Next the value 0xFF is stored in R20 and a loop of 8 iterations with R16 as iterator is made.

By adding a minus after the Y pointer in the ST instruction line. The compiler knows to use the ST with Y and pre-decrement bytecode when compiling.

In the first iteration will first decrement Y and then write the value from R20 to the address Y is pointing to. Before the decrement Y points to 0x108. This gets decremented to 0x107 and then the write is done to that address. After the write, Y will pointing to 0x107 (YH = 0x01 and YL = 0x07)

The second iteration will decrement Y from 0x107 to 0x106 before writing the value from R20 to the address Y is currently pointing to, in this iteration this has become address 0x106. After the write, Y will pointing to 0x106 (YH = 0x01 and YL = 0x06)

After the eight iteration, the SRAM addresses 0x107 to 0x100 will contain 0xFF as value. Y will now point to 0x100 (YH = 0x01 and YL = 0x00). The figure shows a screenshot of the SRAM from the simulator displaying that the writing was successful.

Note that when any lower register underflows due to the decrement, this carries over in the upper register. For example predecrement from 0x100 (YH = 0x01 and YL = 0x00) results in Y= 0x0FF (YH = 0x00 and YL = 0xFF). The post-decrement option does not exist in this instruction set.

Keywords & addresses

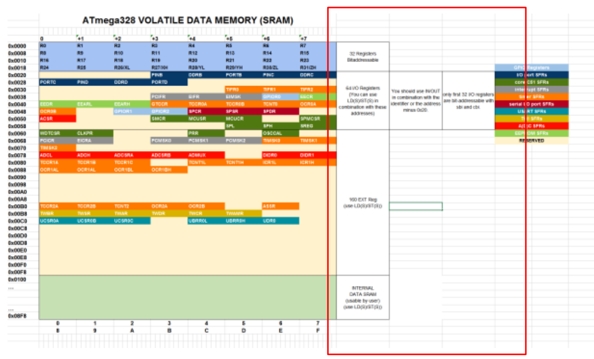
- R0 – R31 are built into the compiler
- Non memory mapped (IO registers) :
  - Use keyword with **IN** and **OUT**
  - Use exact address with **IN** and **OUT**
    - compensate 0x20 offset (**IN** Rx, [keyword] - 0x20)
  - Can be used with **LDS** and **STS** (slower)
    - Keyword + 0x20
    - Exact address from memory overview excel
- Memory mapped (ext IO registers) :
  - Does NOT work with **IN** and **OUT**
  - Can be used with **LDS** and **STS** (slower)
    - Keyword
    - Exact address

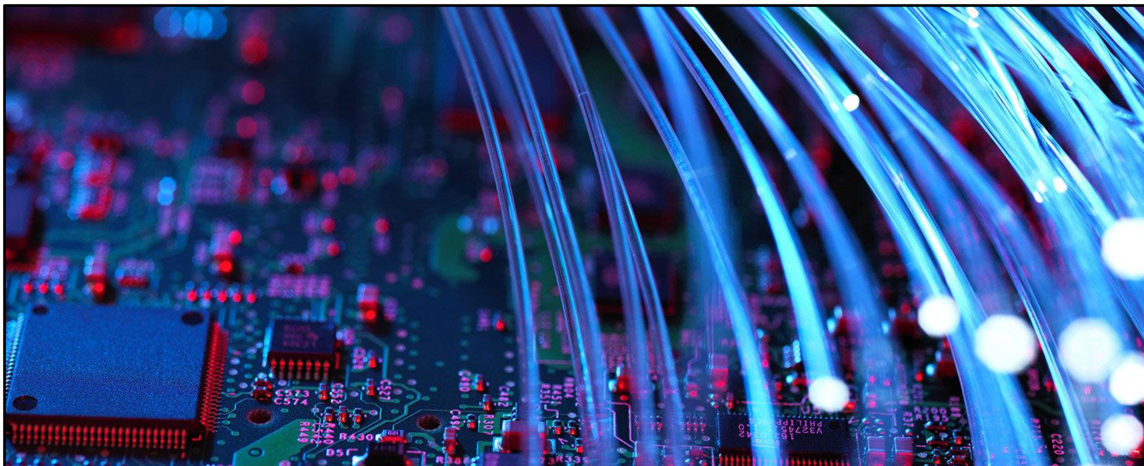
m328pdef.inc

```
.equ  TIMSK2= 0x70; MEMORY MAPPED
.equ  TIMSK1= 0x6f; MEMORY MAPPED
.equ  TIMSK0= 0x6e; MEMORY MAPPED
.equ  PCMSK1= 0x6c; MEMORY MAPPED
.equ  PCMSK2= 0x6d; MEMORY MAPPED
.equ  PCMSK0= 0x6b; MEMORY MAPPED
.equ  EICRA= 0x69; MEMORY MAPPED
.equ  PCICR= 0x68; MEMORY MAPPED
.equ  OSCCAL= 0x66; MEMORY MAPPED
.equ  PRR= 0x64; MEMORY MAPPED
.equ  CLKPR= 0x61; MEMORY MAPPED
.equ  WDTCSR= 0x60; MEMORY MAPPED
.equ  SREG= 0x3f
.equ  SPL= 0x3d
.equ  SPH= 0x3e
.equ  SPMCSR= 0x37
.equ  MCUCR= 0x35
.equ  MCUSR= 0x34
.equ  SMCR= 0x33
.equ  ACSR= 0x30
.equ  SPDR= 0x2e
.equ  SPSR= 0x2d
.equ  SPCR= 0x2c
.equ  GPIOR2= 0x2b
.equ  GPIOR1= 0x2a
.equ  OCR0B= 0x28
.equ  OCR0A= 0x27
```

Memory overview

- See Section 8. AVR Memories in
- AT328P_microcontroller.pdf
- MemoryOverview.xlsx





End of Part 2: Memory