# MCT 4334

# Embedded System Design

## Week 08 Non-volatile Memory

# Outline

- Types of non-volatile memory
- EEPROM on ATmega328p
- EEPROM registers
- Programming examples
- Working with strings and files

# Non-volatile memory (NVM)

- A type of memory than persists information even without electrical power.

Applications

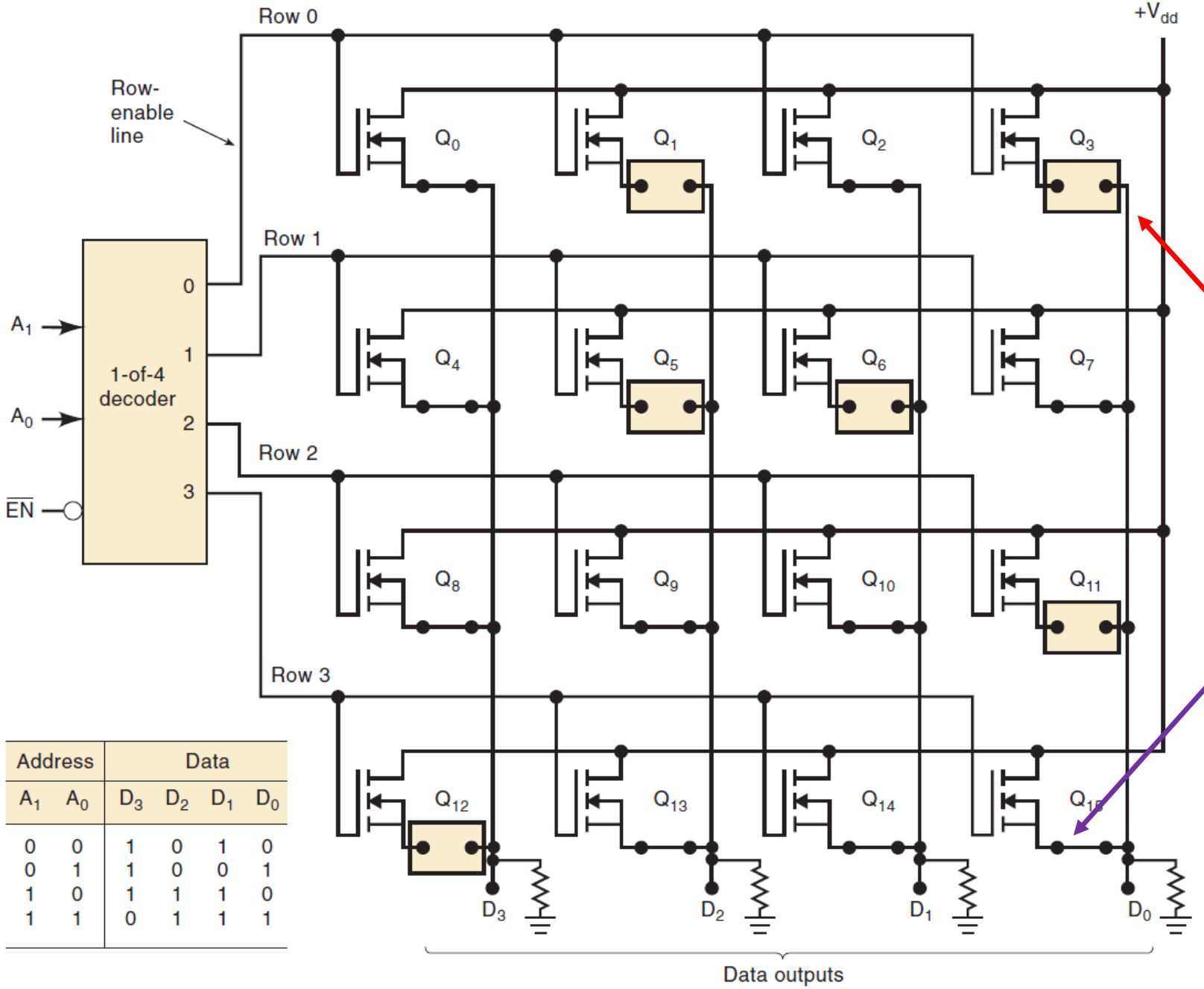
- Storing settings and user preferences.
- Storing program/instructions.
- Data logging.
- Adaptive control systems.
- Machine learning in robotics (storing learnt models).

# ROM

- Read-Only Memory (ROM).
- Data cannot be changed.

Applications

- CD-ROMs
- Cartridges for games.




| Address | | Data | | | |
|---|---|---|---|---|---|
| $A_1$ | $A_0$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |

A type of ROM called mask-programmed ROM

The information gets stored at the time of manufacturing

Cannot be changed later

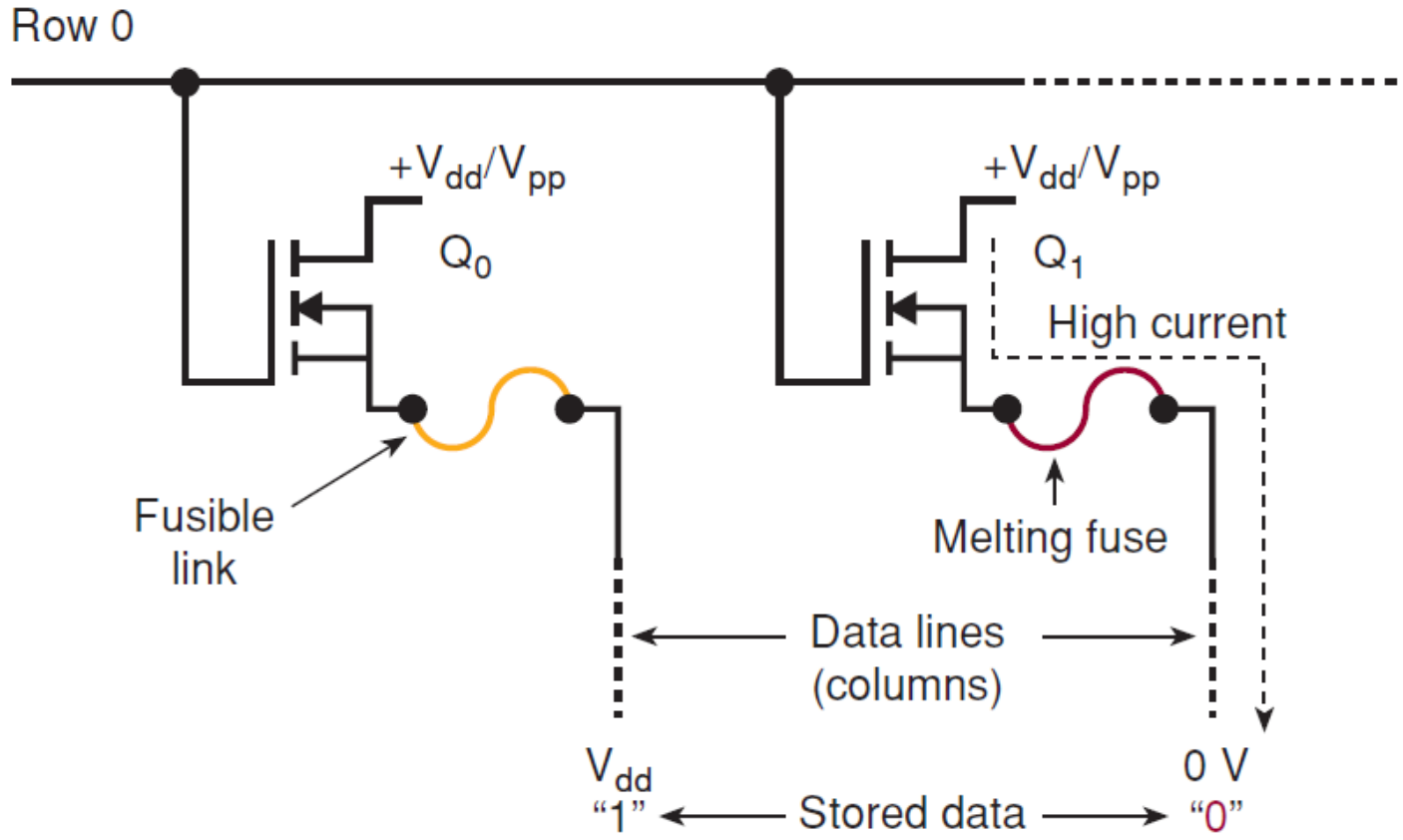
- Open source connection stores a “0”
- Close source connection stores a “1”

# PROM

- Programmable ROMs
- They are not programmed during the manufacturing process, but custom-programmed by the user

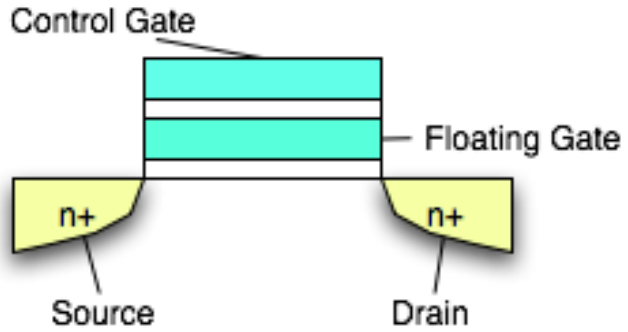
- User “burns” data by melting fuses.
- Burnt fuse = open source = 0

- Once programmed, the data cannot be changed.



# EPROM

- Erasable Programmable ROM.
- Floating-gate transistors are used.
- Data can be changed more than once.
- EPROMs have a quartz window that allows them to be erased with ultraviolet light. The whole memory will be erased. The erase operation takes up to 20 minutes.
- Writing new data to an EPROM requires a special programmer circuit. (Old data must be erased before new data get written)



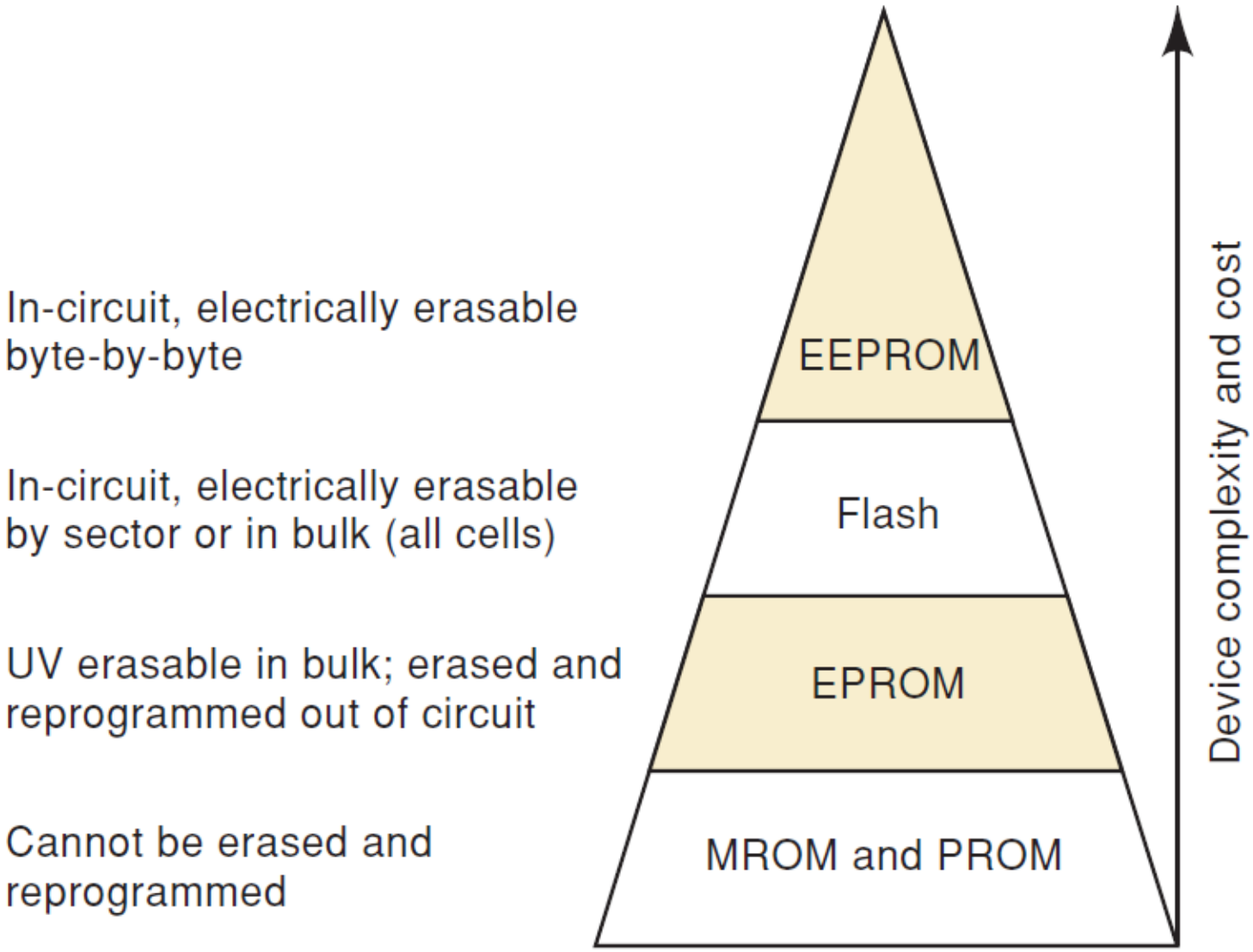

# Flash Memory

- Improvement over EPROM.
- Electronically erasable (without UV exposure)
- It can only erase one block or sector at a time.

# EEPROM

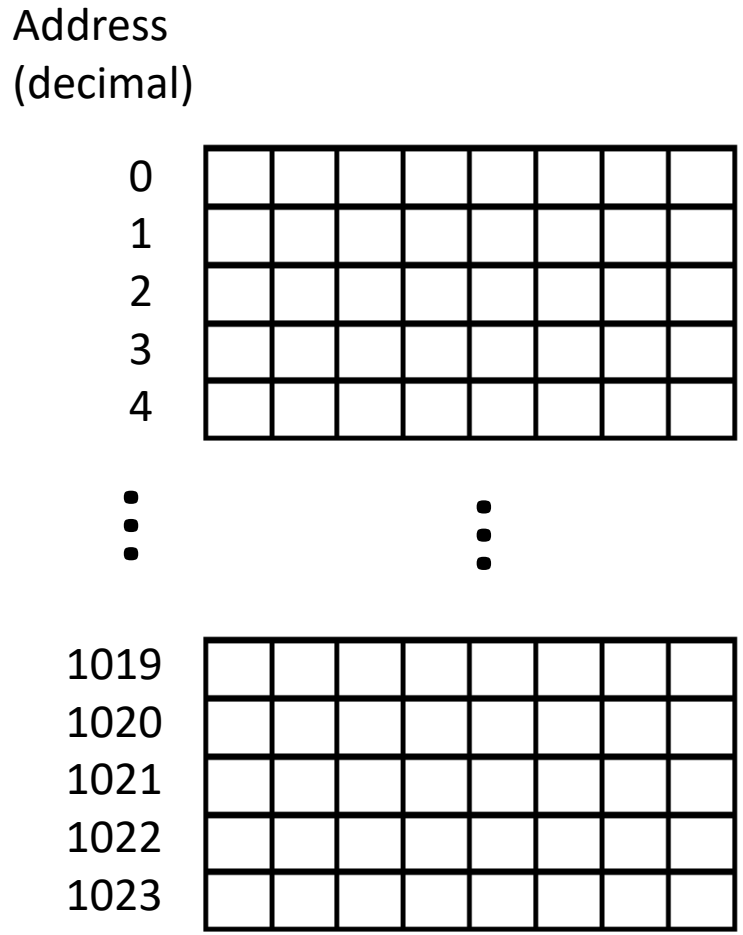
- Electronically Erasable Programmable ROM
- Unlike flash memory, individual bytes are electronically erasable.

# Memory complexity and cost



# EEPROM on ATmega328p

- ATmega328p has an on-chip 1 KB EEPROM.
- The memory is byte-addressable (from 0 to 1023).
- There are hardware peripherals for reading and writing EEPROM.
- There are 4 registers to perform read and write operations.



# EEAR

EEPROM Address Register – low byte and high byte (x41 and x42)

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0x42 | - | - | - | - | - | - | EEAR9 | EEAR8 |
| Read/Write | R | R | R | R | R | R | R/W | R/W |
| Default | 0 | 0 | 0 | 0 | 0 | 0 | - | - |

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0x41 | EEAR7 | EEAR6 | EEAR5 | EEAR4 | EEAR3 | EEAR2 | EEAR1 | EEAR0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | - | - | - | - | - | - | - | - |

These registers store 10-bit address to read and write from EEPROM.

# EEDR

EEPROM Data Register (x40)

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0x40 | EEDR7 | EEDR6 | EEDR5 | EEDR4 | EEDR3 | EEDR2 | EEDR1 | EEDR0 |
| Read/Write | R/W | R/W | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- For read operation, this register contains data read out from the EEPROM at the address given by EEAR.

- For write operation, this register contains data to be written to EEPROM at the address given by EEAR

# EECR

## EEPROM Control Register (x3F)

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0x3F | - | - | EEPM1 | EEPM0 | EERIE | EEMPE | EEPE | EERE |
| Read/Write | R | R | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | - | - | 0 | 0 | - | 0 |

| EEPM | Description | Time taken |
|---|---|---|
| **00** | Erase and Write | 3.4 ms |
| **01** | Erase only | 1.8 ms |
| **10** | Write only | 1.8 ms |
| **11** | - | - |

**EEPM:** Programming (writing) mode. Refer to the table

**EERIE:** Interrupt Enable

**EEMPE:** Master Write Enable

**EEPE:** Write Enable (Set to initiate writing)

**EERE:** Read Enable (Set to initiate reading)

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0x3F | - | - | EEPM1 | EEPM0 | EERIE | EEMPE | EEPE | EERE |
| Read/Write | R | R | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | - | - | 0 | 0 | - | 0 |

**EEMPE**

To write data to EEPROM, this bit has to be set first. After 4 CPU clock cycles, this bit will automatically be cleared.

**EEPE**

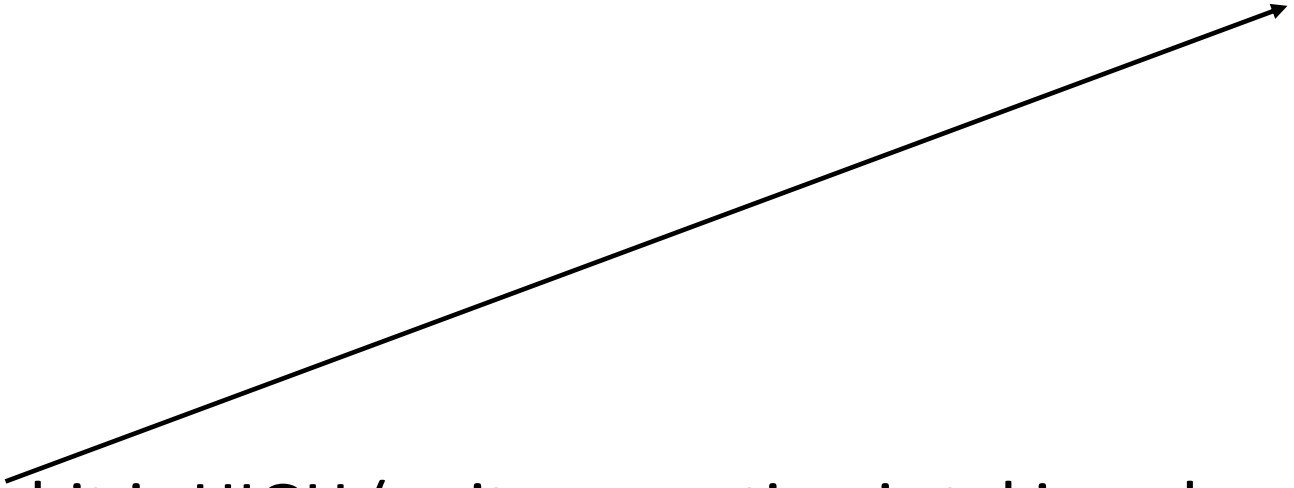
After setting the EEMPE bit, within 4 CPU clock cycles, this bit has to be set to initiate writing. After the write operation, this bit will automatically get cleared. Note: the CPU will not be halted while the write operation is being carried out.

**EERE**

This bit needs to be set to initiate reading. The CPU will be halted until the read operation is over. After the read operation, this bit will automatically get cleared.

# Procedure for reading

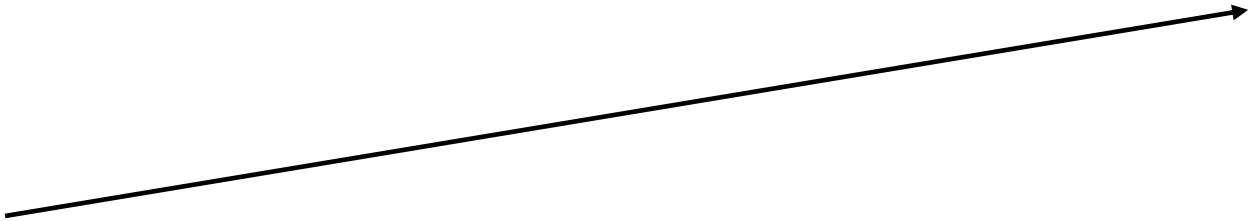
| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0x3F | - | - | EEPM1 | EEPM0 | EERIE | EEMPE | EEPE | EERE |
| Read/Write | R | R | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | - | - | 0 | 0 | - | 0 |



1) If this bit is HIGH (write operation is taking place), wait until it becomes low
2) Store the desired address in EEAR.
3) Simply set the first bit of the EECR to initiate reading
4) Read the contents of EEDR.

# Procedure for writing

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 0x3F | - | - | EEPM1 | EEPM0 | EERIE | EEMPE | EEPE | EERE |
| Read/Write | R | R | R/W | R/W | R/W | R/W | R/W | R/W |
| Default | 0 | 0 | - | - | 0 | 0 | - | 0 |



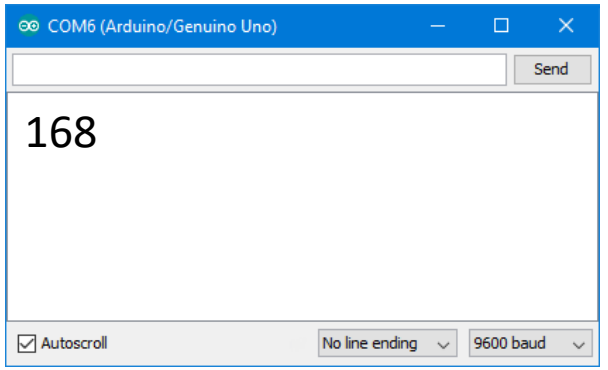
1) If this bit is HIGH (previous write operation is taking place), wait until it becomes low
2) Set the desired mode (erase + write, erase only or write only)
3) Store the desired address in EEAR.
4) Store the desired data in EEDR
5) Set EEMPE
6) Set EEPE to initiate writing

# Code for Reading

```
char ReadByte(int address)
{
    char* data_register = (char*) 0x40;                    //Points to EEDR
    volatile char *control_register = (char*) 0x3F;        //Points to EECR
    int* address_register = (int*) 0x41; //Points to EEAR. Note that int is 16bit on ATmega328p
                                         //This pointer points to both low byte and high byte
                                         //of EEAR

    while (((*control_register) & 2)) //If the data is being written (EEPE is high)
    {
        //do nothing
    }
    *address_register = address; //Stores the address in the EEAR
    *control_register = 1;  //Set EERE (Initiate reading)
    return *data_register; //Return the contents of the data register (EEDR)
}

int main()
{
    Serial.begin(9600);
    char a = ReadByte(115);  //Read the value of the EEPROM at memory address 115
    Serial.println(a);       //Prints the value of the EEPROM at memory address 115

}
```

# Code for Writing

```
void WriteByte(int address, char data)
{
    char* data_register = (char*) 0x40;                    //Points to EEDR
    volatile char *control_register = (char*) 0x3F;        //Points to EECR
    int* address_register = (int*) 0x41; //Points to EEAR. Note that int is 16bit on ATmega328p
                                         //This pointer points to both low byte and high byte
                                         //of EEAR

    while (((*control_register) & 2)) //If the data is being written (EEPE is high)
    {
        //do nothing
    }
    *address_register = address;     //Stores the address in the EEAR
    *data_register = data;           //Stores the data in the EEDR
    *control_register = 4;           //Enable Master Write
    *control_register |= 2;          //Start writing
}

int main()
{
    Serial.begin(9600);
    WriteByte(115, 168);                        //Write a value of 168 to the memory location 115
    unsigned char a = ReadByte(115);            //Read the value at memory location 115
    Serial.println(a);                          //Prints the value
}
```



```
class EEPROM
{
    public:
        void WriteByte(int address, char data)
        {
             //Code for Writing
        }
        char ReadByte(int address)
        {
            //Code for Writing
        }

};

int main()
{
    EEPROM eeprom;                          //Creates an instance of EEPROM class
    Serial.begin(9600);
    eeprom.WriteByte(115, 168);               //Write a value of 168 to the memory location 115
    unsigned char a = eeprom.ReadByte(115); //Read the value at memory location 115
    Serial.println(a);                      //Prints the value

}
```

- The Read and Write functions can be inserted in a class called EEPROM.
- To invoke these function, an object of the EEPROM class has to be created first.

```
class EEPROM
{
    public:
        static void WriteByte(int address, char data)
        {
             //Code for Writing
        }
        static char ReadByte(int address)
        {
            //Code for Writing
        }

};

int main()
{
    Serial.begin(9600);
    EEPROM::WriteByte(115, 168);                //Write a value of 168 to the memory location 115
    unsigned char a = EEPROM::ReadByte(115); //Read the value at memory location 115
    Serial.println(a);                          //Prints the value

}
```

- Now the Read and Write functions have been declared to be static members of the EEPROM class.
- Static members can be accessed using the :: operator (without creating an object of the class)

# Example:

## EEPROM class with more features

```
class EEPROM
{
    public:
        static void WriteByte(int address, char data){//Code here//}

        static char ReadByte(int address){//Code here//}

        static void EraseByte(int address){//Code here//}

        static void EraseAll(){//Code here//}

        static void WriteString(int address, char* data){//Code here//}

        static char* ReadString(int address){//Code here//}

        static void WriteTextFile(int address, char** data){//Code here//}

        static char** ReadTextFile(int address){//Code here//}


};
```

# More examples

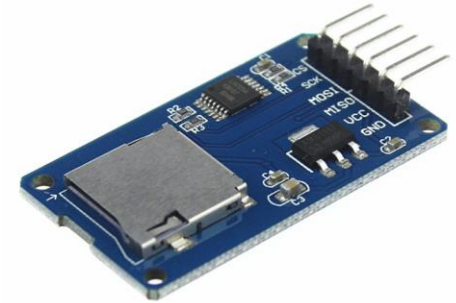
- Any data can potentially be stored on EEPROM.
- Text files, sound files, HTML files, pictures, videos

# External NVM

External NVM can be utilized if more than 1K of storage space is required.

MICROCHIP 24AA1026-I/P
Size: 1 MB
Interface: I2C
Price: RM 18

If gigabytes of storage space is required, SD cards can be utilized.



# Working with strings on NVM

- Recall from Week 02 Lecture that arrays and pointers are identical.
- String is an array of characters

- The following two snippets of code produce the same output

Output

```
H
E
L
```

```c
char *a = "HELLO";

printf("%c \n", *a);
printf("%c \n", *(a+1));
printf("%c \n", *(a+2));
```

```c
char a[] = "HELLO";

printf("%c \n", a[0]);
printf("%c \n", a[1]);
printf("%c \n", a[2]);
```

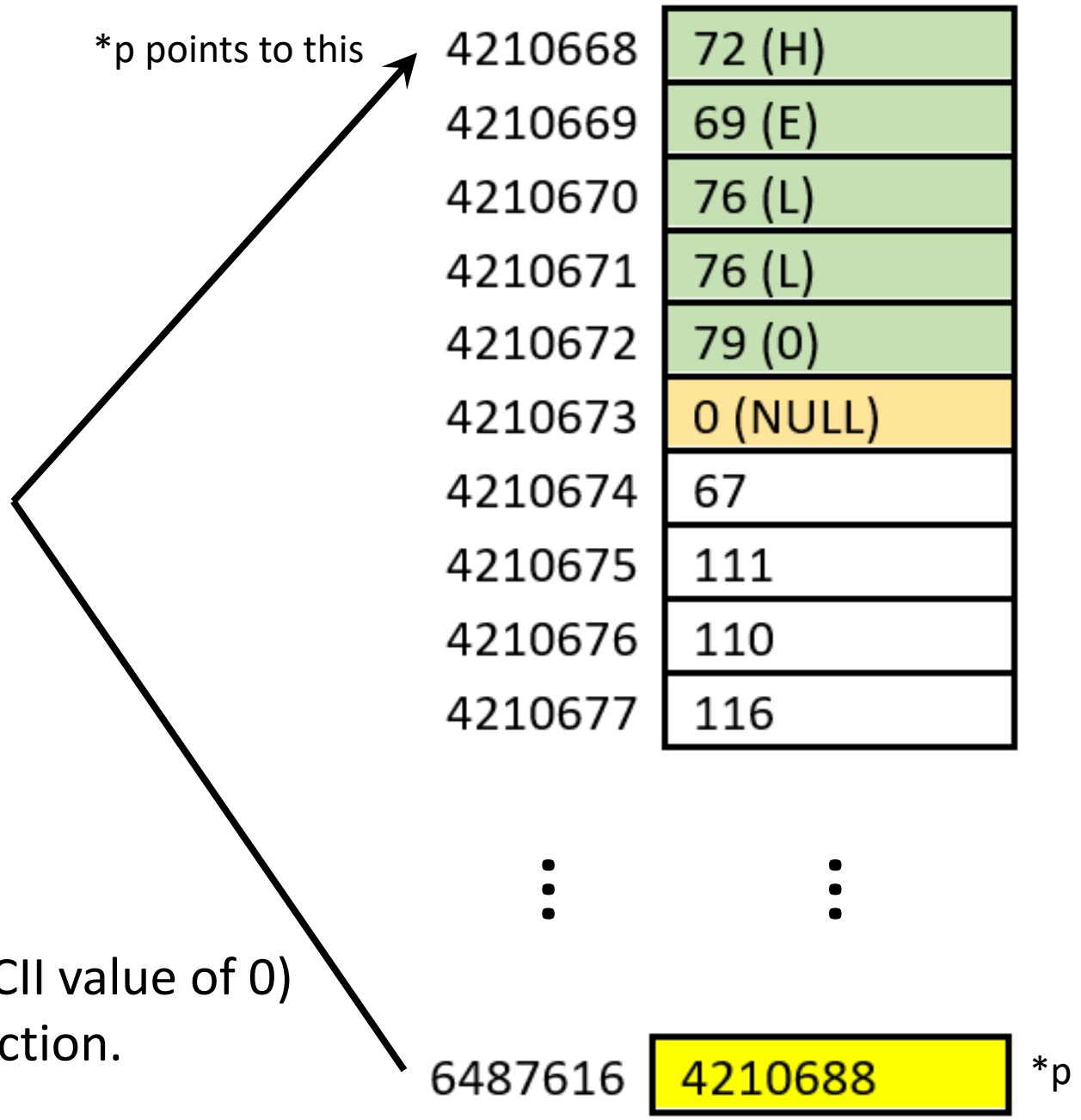
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{

	char *p = "HELLO";

	printf("Address *p = %d \n", &p);
	printf("Content of *p = %d \n", p);

	for (int i=0; i<10; i++)
	{
		printf("%d\n", p[i]);
	}

	return 0;
}
```

- string p is 5 characters in length
- *p pointers to the memory location of char “H”
- What is the output of this program?

Output

```
Address of *p = 6487616
Contents of *p = 4210668
72
69
76
76
79
0
67
111
110
116
```

Data of other variables




**Important observations:**

- Strings are terminated by null characters (ASCII value of 0)
- Arrays/pointers do not have any length restriction.

- The string.h header file contains a function called strlen which returns the length of any given string.
- Arduino library utilizes string.h
- strlen loops through the string until the NULL character is found. It simply counts the number of character until the NULL character

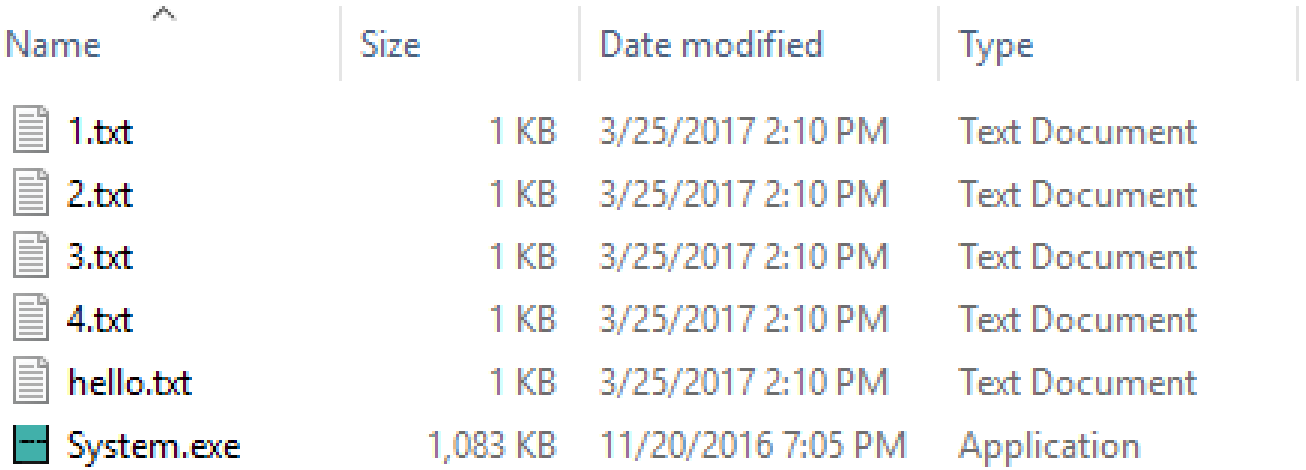
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{

	char *p = "HELLO";
	printf("Length of p = %d \n", strlen(a));

	return 0;
}

unsigned int strlen(char *str)
{
	unsigned int length = 0;

	while(*(str+length)!=0)
	{
		length++;
	}
	return length;
}
```

Output

```
5
```

# Working with files on NVM

- In many embedded systems (such as compact cameras, CCTVs, etc) data are written into files.
- Files are in turn hierarchically organized into folders.
- Files have names and meta-data such as date created and date modified.
- A NVM just stores bytes.
- Without a **file system** (a standard), there is no way of telling where a file begins and ends in the NVM.



# Working with files on NVM

- If a particular embedded system is just meant to store some data without any need to access them on other platforms, you do not really need to worry about file systems. The data can be stored in any manner that you wish to.
- Otherwise, you need to follow a file system. For example, a robot that periodically captures photos that need to be accessed on a computer.

| Name | Modified | Size |
|---|---|---|
| Parent directory | - | - |
| P170324_000000_001501.avi | 24-Mar-2017 00:15 | 97.8M |
| P170324_001501_003003.avi | 24-Mar-2017 00:30 | 98.1M |
| P170324_003002_004504.avi | 24-Mar-2017 00:45 | 98.0M |
| P170324_004503_010006.avi | 24-Mar-2017 01:00 | 98.2M |
| P170324_010004_011508.avi | 24-Mar-2017 01:15 | 98.1M |
| P170324_011505_013009.avi | 24-Mar-2017 01:30 | 98.1M |
| P170324_013006_014507.avi | 24-Mar-2017 01:45 | 97.7M |
| P170324_014507_020009.avi | 24-Mar-2017 02:00 | 98.1M |
| P170324_020008_021511.avi | 24-Mar-2017 02:15 | 98.2M |
| P170324_021509_023012.avi | 24-Mar-2017 02:30 | 98.2M |
| P170324_023010_024514.avi | 24-Mar-2017 02:45 | 98.1M |
| P170324_024511_030012.avi | 24-Mar-2017 03:00 | 97.8M |
| P170324_030011_031514.avi | 24-Mar-2017 03:15 | 98.1M |

# File Systems

- A file system defines formats for organizing files and folders.
- A file system defines

    - Where a file begins and ends in the memory
    - Which folder a particular file is in and where that location information is stored
    - How long can a file name be
    - How metadata (date modified, last access date, etc)  are stored

# Famous File Systems

- NTFS (New Technology File System)  used in Windows operation systems.
- FAT (File Allocation Table) commonly used in Flash drives and SD cards.
- ISO 9660 commonly used for optical discs
- HFS Plus used in Mac OS