ECE3411 – Fall 2015

Lecture 5c.

EEPROM
Watchdog Timer

Marten van Dijk, Syed Kamran Haider
Department of Electrical & Computer Engineering
University of Connecticut
Email: {vandijk, syed.haider}@engr.uconn.edu

UCONN

EEPROM: Electrically Erasable Programmable ROM

7.4 EEPROM Data Memory

The ATmega48PA/88PA/168PA/328P contains 256/512/512/1K bytes of data EEPROM memory. It is organized as a separate data space, in which single bytes can be read and written. The EEPROM has an endurance of at least 100,000 write/erase cycles. The access between the EEPROM and the CPU is described in the following, specifying the EEPROM Address Registers, the EEPROM Data Register, and the EEPROM Control Register.

"Memory Programming" on page 294 contains a detailed description on EEPROM Programming in SPI or Parallel Programming mode.

You should not access EEPROM in main in a loop, otherwise, it will not exist any more !

EEPROM reads in 2 cycles and writes in about 100 cycles
Flash (program + optional read only data): read in 2 cycles
RAM: read+write in 1 cycle each

Data in EEPROM remains even if you pull the chip out of the board or turn power on and off

EEPROM

7.6 Register Description

7.6.1 EEARH and EEARL – The EEPROM Address Register

Bit	15	14	13	12	11	10	9	8	
0x22 (0x42)	–	–	–	–	–	–	–	EEAR8	EEARH
0x21 (0x41)	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	EEARL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R/W	
	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	X	
	X	X	X	X	X	X	X	X	

- **Bits 15..9 – Res: Reserved Bits**

These bits are reserved bits in the ATmega48PA/88PA/168PA/328P and will always read as zero.

- **Bits 8..0 – EEAR8..0: EEPROM Address**

The EEPROM Address Registers – EEARH and EEARL specify the EEPROM address in the 256/512/512/1K bytes EEPROM space. The EEPROM data bytes are addressed linearly between 0 and 255/511/511/1023. The initial value of EEAR is undefined. A proper value must be written before the EEPROM may be accessed.

EEAR8 is an unused bit in ATmega48PA and must always be written to zero.

EEPROM

7.6.2 EEDR – The EEPROM Data Register

Bit	7	6	5	4	3	2	1	0	
0x20 (0x40)	MSB							LSB	EEDR
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	

- **Bits 7..0 – EEDR7.0: EEPROM Data**

For the EEPROM write operation, the EEDR Register contains the data to be written to the EEPROM in the address given by the EEAR Register. For the EEPROM read operation, the EEDR contains the data read out from the EEPROM at the address given by EEAR.

EEPROM

7.6.3 EECR – The EEPROM Control Register

Bit	7	6	5	4	3	2	1	0	
0x1F (0x3F)	–	–	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE	EECR
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	X	X	0	0	X	0	

- **Bits 7..6 – Res: Reserved Bits**

These bits are reserved bits in the ATmega48PA/88PA/168PA/328P and will always read as zero.

- **Bits 5, 4 – EEPM1 and EEPM0: EEPROM Programming Mode Bits**

The EEPROM Programming mode bit setting defines which programming action that will be triggered when writing EEPE. It is possible to program data in one atomic operation (erase the old value and program the new value) or to split the Erase and Write operations in two different operations. The Programming times for the different modes are shown in Table 7-1. While EEPE

EEPROM

is set, any write to EEPMn will be ignored. During reset, the EEPMn bits will be reset to 0b00 unless the EEPROM is busy programming.

Table 7-1. EEPROM Mode Bits

EEPM1	EEPM0	Programming Time	Operation
0	0	3.4 ms	Erase and Write in one operation (Atomic Operation)
0	1	1.8 ms	Erase Only
1	0	1.8 ms	Write Only
1	1	–	Reserved for future use

- **Bit 3 – EERIE: EEPROM Ready Interrupt Enable**

Writing EERIE to one enables the EEPROM Ready Interrupt if the I bit in SREG is set. Writing EERIE to zero disables the interrupt. The EEPROM Ready interrupt generates a constant interrupt when EEPE is cleared. The interrupt will not be generated during EEPROM write or SPM.

- **Bit 2 – EEMPE: EEPROM Master Write Enable**

The EEMPE bit determines whether setting EEPE to one causes the EEPROM to be written. When EEMPE is set, setting EEPE within four clock cycles will write data to the EEPROM at the selected address If EEMPE is zero, setting EEPE will have no effect. When EEMPE has been written to one by software, hardware clears the bit to zero after four clock cycles. See the description of the EEPE bit for an EEPROM write procedure.

One can do sequential writes before an erasure:

- During a write one can only flip bits from 0 to 1
- If a bit needs to flip from a 1 to 0, an erasure is required before doing a write
- Hence, sequential writes may be possible if only 0 to 1 bit flips need to be written
- The lifetime of an eeprom bit is about 100.000 0 →1 write and 1→0 erasure cycles
- So, if sequential writes before an erasure are possible, the lifetime of eeprom is not unnecessarily shortened

EEPROM

- **Bit 1 – EEPE: EEPROM Write Enable**

The EEPROM Write Enable Signal EEPE is the write strobe to the EEPROM. When address and data are correctly set up, the EEPE bit must be written to one to write the value into the EEPROM. The EEMPE bit must be written to one before a logical one is written to EEPE, otherwise no EEPROM write takes place. The following procedure should be followed when writing the EEPROM (the order of steps 3 and 4 is not essential):

1. Wait until EEPE becomes zero.
2. Wait until SELFPRGEN in SPMCSR becomes zero.
3. Write new EEPROM address to EEAR (optional).
4. Write new EEPROM data to EEDR (optional).
5. Write a logical one to the EEMPE bit while writing a zero to EEPE in EECR.
6. Within four clock cycles after setting EEMPE, write a logical one to EEPE.

The EEPROM can not be programmed during a CPU write to the Flash memory. The software must check that the Flash programming is completed before initiating a new EEPROM write. Step 2 is only relevant if the software contains a Boot Loader allowing the CPU to program the Flash. If the Flash is never being updated by the CPU, step 2 can be omitted. See ”Boot Loader Support – Read-While-Write Self-Programming, ATmega88PA, ATmega168PA and ATmega328P” on page 277 for details about Boot programming.

Caution: An interrupt between step 5 and step 6 will make the write cycle fail, since the EEPROM Master Write Enable will time-out. If an interrupt routine accessing the EEPROM is interrupting another EEPROM access, the EEAR or EEDR Register will be modified, causing the interrupted EEPROM access to fail. It is recommended to have the Global Interrupt Flag cleared during all the steps to avoid these problems.

EEPROM

When the write access time has elapsed, the EEPE bit is cleared by hardware. The user software can poll this bit and wait for a zero before writing the next byte. When EEPE has been set, the CPU is halted for two cycles before the next instruction is executed.

- **Bit 0 – EERE: EEPROM Read Enable**

The EEPROM Read Enable Signal EERE is the read strobe to the EEPROM. When the correct address is set up in the EEAR Register, the EERE bit must be written to a logic one to trigger the EEPROM read. The EEPROM read access takes one instruction, and the requested data is available immediately. When the EEPROM is read, the CPU is halted for four cycles before the next instruction is executed.

The user should poll the EEPE bit before starting the read operation. If a write operation is in progress, it is neither possible to read the EEPROM, nor to change the EEAR Register.

The calibrated Oscillator is used to time the EEPROM accesses. Table 7-2 lists the typical programming time for EEPROM access from the CPU.

Table 7-2. EEPROM Programming Time

Symbol	Number of Calibrated RC Oscillator Cycles	Typ Programming Time
EEPROM write (from CPU)	26,368	3.3 ms

The following code examples show one assembly and one C function for writing to the EEPROM. The examples assume that interrupts are controlled (e.g. by disabling interrupts globally) so that no interrupts will occur during execution of these functions. The examples also assume that no Flash Boot Loader is present in the software. If such code is present, the EEPROM write function must also wait for any ongoing SPM command to finish.

Assembly Code Example

```
EEPROM_write:
  ; Wait for completion of previous write
  sbic EECR,EEPE
  rjmp EEPROM_write
  ; Set up address (r18:r17) in address register
  out  EEARH, r18
  out  EEARL, r17
  ; Write data (r16) to Data Register
  out  EEDR,r16
  ; Write logical one to EEMPE
  sbi  EECR,EEMPE
  ; Start eeprom write by setting EEPE
  sbi  EECR,EEPE
  ret
```

C Code Example

```
void EEPROM_write(unsigned int uiAddress, unsigned char ucData)
{
  /* Wait for completion of previous write */
  while(EECR & (1<<EEPE))
    ;
  /* Set up address and Data Registers */
  EEAR = uiAddress;
  EEDR = ucData;
  /* Write logical one to EEMPE */
  EECR |= (1<<EEMPE);
  /* Start eeprom write by setting EEPE */
  EECR |= (1<<EEPE);
}
```

Assembly Code Example

```
EEPROM_read:
  ; Wait for completion of previous write
  sbic EECR,EEPE
  rjmp EEPROM_read
  ; Set up address (r18:r17) in address register
  out  EEARH, r18
  out  EEARL, r17
  ; Start eeprom read by writing EERE
  sbi  EECR,EERE
  ; Read data from Data Register
  in   r16,EEDR
  ret
```

C Code Example

```
unsigned char EEPROM_read(unsigned int uiAddress)
{
  /* Wait for completion of previous write */
  while(EECR & (1<<EEPE))
    ;
  /* Set up address register */
  EEAR = uiAddress;
  /* Start eeprom read by writing EERE */
  EECR |= (1<<EERE);
  /* Return data from Data Register */
  return EEDR;
}
```

EEPROM

```
#include <avr/eeprom.h>
#define eeprom_true 0  //Suppose you want to store a flag at position 0
#define eeprom_data 1 //Suppose you want to store data at position 1

// Code snippet in e.g. an initialization
if (eeprom_read_byte((uint8_t*)eeprom_true) == 'T')
 {
        time = eeprom_read_byte((uint8_t*)eeprom_data);
 }
else
{
        time = 0; //Initialize time to 0 as this is the first time the code is running on the MCU
                  //before it has ever been reset
}
```

Use 'T' or something else from default 0 byte data values

(unint8_t*) is used to cast eeprom_data and eeprom_true into a byte pointer

EEPROM

```
// Code snippet in some task:
if (SW1_Pressed)
{
        if (eeprom_read_byte((uint8_t*)eeprom_true) != 'T')
         {
                eeprom_write_byte((uint8_t*)eeprom_true,'T');
         }
        eeprom_write_byte((uint8_t*)eeprom_data,time);  //Write time to EEPROM
        fprintf(stdout,"button push at %d \n\r", time);        //Write time to UART
}
```

Watchdog Timer

In Interrupt mode, the WDT gives an interrupt when the timer expires. This interrupt can be used to wake the device from sleep-modes, and also as a general system timer. One example is to limit the maximum time allowed for certain operations, giving an interrupt when the operation has run longer than expected. In System Reset mode, the WDT gives a reset when the timer expires. This is typically used to prevent system hang-up in case of runaway code. The third mode, Interrupt and System Reset mode, combines the other two modes by first giving an interrupt and then switch to System Reset mode. This mode will for instance allow a safe shutdown by saving critical parameters before a system reset.

Table 10-1. Watchdog Timer Configuration

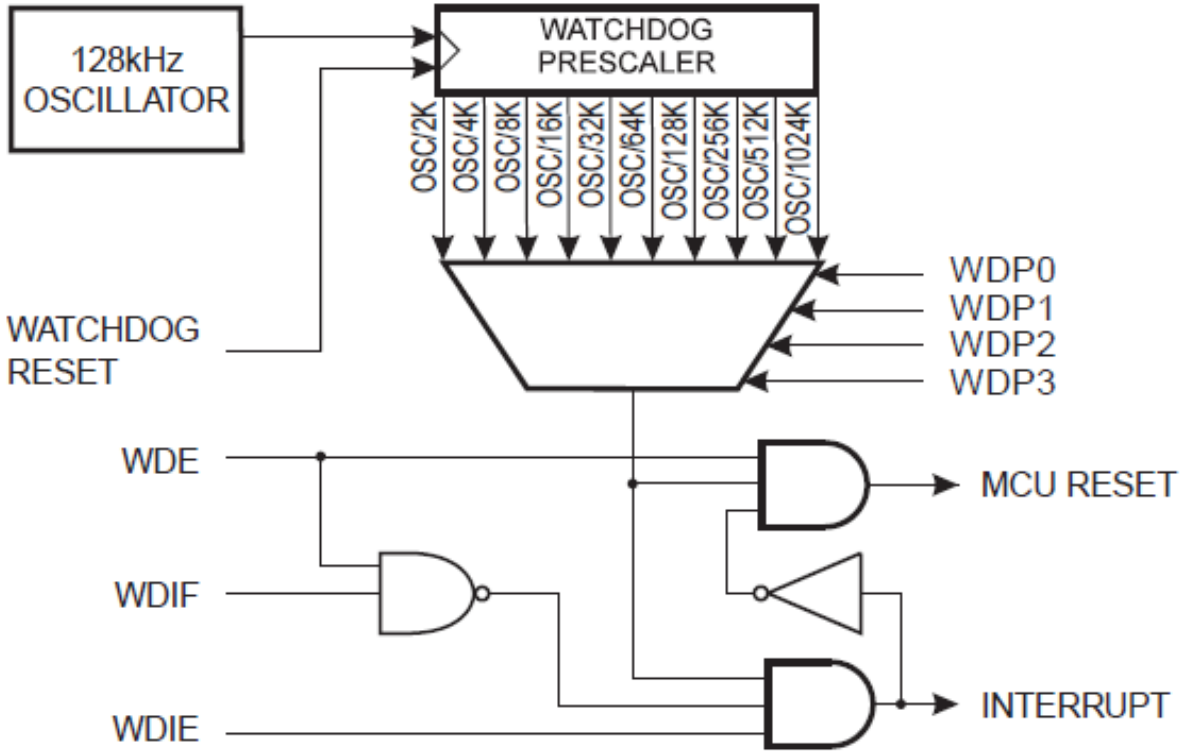
WDTON[(1)]	WDE	WDIE	Mode	Action on Time-out
1	0	0	Stopped	None
1	0	1	Interrupt Mode	Interrupt
1	1	0	System Reset Mode	Reset
1	1	1	Interrupt and System Reset Mode	Interrupt, then go to System Reset Mode
0	x	x	System Reset Mode	Reset

Watchdog Timer

- Suppose your application is heating a room
- Once the temperature is too high, the application should turn off
- Implement a watchdog timer:
  - An independent heat sensor causes an ISR (e.g., external interrupt) when the measured temperature is low enough
  - This ISR resets the watchdog and disables itself
  - The main program regularly enables the ISR
  - The watchdog will turn off the system if
    - The sensor breaks → no ISR will be called → the watchdog is not reset → the watchdog will count down to 0 and causes the system to be reset (with or without executing a watchdog ISR before reset)
    - The temperature is too high → no ISR will be called → etc.
- Safety: if sensor breaks or if temperature is to high, the system is reset
  - In SW one can always reset the system if the temperature is too high, but how does it know whether the sensor that measures the temperature is functioning correctly?

Watchdog Timer

Figure 10-7. Watchdog Timer



Watchdog reset restarts the counter before the time-out value is reached:

- #include <avr/wdt.h>
- wdt_reset();

Watchdog Timer

10.9.2 WDTCSR – Watchdog Timer Control Register

Bit	7	6	5	4	3	2	1	0	
(0x60)	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0	WDTCSR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	X	0	0	0	

To further ensure program security, alterations to the Watchdog set-up must follow timed sequences. The sequence for clearing WDE and changing time-out configuration is as follows:

1. In the same operation, write a logic one to the Watchdog change enable bit (WDCE) and WDE. A logic one must be written to WDE regardless of the previous value of the WDE bit.
2. Within the next four clock cycles, write the WDE and Watchdog prescaler bits (WDP) as desired, but with the WDCE bit cleared. This must be done in one operation.

Watchdog Timer

10.9.2 WDTCSR – Watchdog Timer Control Register

Bit	7	6	5	4	3	2	1	0	
(0x60)	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0	WDTCSR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	X	0	0	0	

- WDTCSR |= (1<<WDCE) | (1<<WDE);
  - WDCE: Watchdog Change Enable allows to make changes during the next 4 cycles (= one operation)
  - WDE: Watchdog Enable
- WDTCSR = (1<<WDIE)|(1<<WDE)|(1<<WDP3);
  - This operation clears the WDCE bit as required by using = (not |=)
  - WDIE: Watchdog Interrupt Enable → E.g., we want an interrupt after 4.0 seconds
  - WDE: Watchdog Enable means a system reset is generated after 4.0 seconds
  - (1<<WDP3) sets a prescalar

Watchdog Timer

Table 10-2. Watchdog Timer Prescale Select

WDP3	WDP2	WDP1	WDP0	Number of WDT Oscillator Cycles	Typical Time-out at V_{CC} = 5.0V
0	0	0	0	2K (2048) cycles	16 ms
0	0	0	1	4K (4096) cycles	32 ms
0	0	1	0	8K (8192) cycles	64 ms
0	0	1	1	16K (16384) cycles	0.125 s
0	1	0	0	32K (32768) cycles	0.25 s
0	1	0	1	64K (65536) cycles	0.5 s
0	1	1	0	128K (131072) cycles	1.0 s
0	1	1	1	256K (262144) cycles	2.0 s
1	0	0	0	512K (524288) cycles	4.0 s
1	0	0	1	1024K (1048576) cycles	8.0 s

Watchdog Timer

```
void WDT_Prescaler_Change(void)
{
  __disable_interrupt();
  __watchdog_reset();
  /* Start timed  equence */
  WDTCSR |= (1<<WDCE) | (1<<WDE);
  /* Set new prescaler(time-out) value = 64K cycles (~0.5 s) */
  WDTCSR  = (1<<WDE) | (1<<WDP2) | (1<<WDP0);
  __enable_interrupt();
}
```

Nothing can interrupt the timed sequence

Another example from the datasheet without interrupt enable

Watchdog Timer

- If we have enabled the interrupt, an interrupt is created before the system is reset
- E.g., ISR(WDT_vect) { Store state in eeprom }
- After system reset the initialization can read the last state from eeprom

- If a reset occurs, it is good practice to turn of the watchdog as soon as the MCU starts
  - The register contents survive after restart: this means the watchdog is enabled (and reset)
  - If initialization takes too long, then the watchdog will time out and the MCU turns off: the MCU will never get through the initialization
  - So, turn off the watchdog at the start of your code, do the initialization, and turn on the watchdog before entering the main while(1){ …} loop

Watchdog Timer

```
void WDT_off(void)
{
  __disable_interrupt();
  __watchdog_reset();
  /* Clear WDRF in MCUSR */
  MCUSR &= ~(1<<WDRF);
  /* Write logical one to WDCE and WDE */
  /* Keep old prescaler setting to prevent unintentional time-out */
  WDTCSR |= (1<<WDCE) | (1<<WDE);
  /* Turn off WDT */
  WDTCSR = 0x00;
  __enable_interrupt();
}
```

Watchdog Timer

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

#include <avr/eeprom.h>
#define eeprom_true 0  //Suppose you want to store a flag at position 0
#define eeprom_data 1 //Suppose you want to store data at position 1

ISR (WDT_vect)
{
        eeprom_write_dword((uint32_t*)eeprom_data,mode);        //Write our current mode to EEPROM
        eeprom_write_byte((uint8_t*)eeprom_true, 'T');          //Set write flag TRUE
}

void Initialize(void)
{
        … all other initialization …
        WDTCSR |= (1<<WDCE) | (1<<WDE);  // Set Watchdog Condition Edit for four cycles
        WDTCSR = (1<<WDIE) | (1<<WDE) | (1<<WDP3);   // Set WDT Int and Reset; Prescalar at 4.0s.
}
```

Watchdog Timer

```
int main(void)
{
        // WDOG Interrupt and Reset Disable, this only matters if reset occurs.
        wdt_reset();                                    // Reset Watchdog timer
        MCUSR  &= ~(1<<WDRF);                           // Shut off Watchdog Reset Flag
        WDTCSR |= (1<<WDCE) | (1<<WDE);     // Set Watchdog Change Enable and WD Enable
        WDTCSR  = 0x00;                                 // Disable Watchdog

        Initialize();
        // Read TimeOut from EEPROM
        if (eeprom_read_byte((uint8_t*)eeprom_true) == 'T')
        {
                mode = eeprom_read_dword((uint32_t*)eeprom_data);
        }
        else
        {
                mode = 0; // Begin in normal mode
        }
        while (1) { ..... }
}
```