Praktikumsaufgabe: EEPROM im AVR

Prof. Dr. Ralf Gerlich
Hochschule Furtwangen
Fakultät Computer Science & Applications
Robert-Gerwig-Platz 1
D-78120 Furtwangen
ralf.gerlich@hs-furtwangen.de

18. März 2025

Zusammenfassung
Zählen Sie im EEPROM des AVR die Anzahl der Neustarts.

Inhaltsverzeichnis

1 Schaltungsaufbau

Für diese Aufgabe müssen Sie lediglich den Arduino per USB-Schnittstelle mit Ihrem Host-Rechner verbinden. Ein gesonderter Schaltungsaufbau ist nicht erforderlich.

2 Übertragung über die serielle Schnittstelle

Übernehmen Sie die Funktionen `uart_setup` und `uart_send_char` aus der Aufgabe „Serielles Hallo“ in die Projektvorlage, um Text über die serielle Schnittstelle übertragen zu können.

Die Vorlage enthält leere Implementierungen dieser Funktionen. Überschreiben Sie diese mit Ihrer eigenen Implementierung.

3 Aufgabenstellung

Implementieren Sie einen Reboot-Zähler für den ATmega328P. Der Reboot-Zähler soll bei jedem Neustart um eins erhöht und danach dann wiederholt über die serielle Schnittstelle ausgegeben werden – etwa in der folgenden Form:

```
Bisherige Zahl von Neustarts: 15
```

Damit der Zähler auch über Neustarts mit Spannungsunterbrechung hinweg erhalten bleibt, müssen Sie ihn in einem nicht-flüchtigen Speicher ablegen. In diesem Fall werden Sie dafür das integrierte EEPROM des Mikrocontrollers verwenden.

Beim allerersten Start des Programms wird der Reboot-Zähler noch nicht initialisiert sein und muss deshalb auf 0 zurückgesetzt werden. Bei weiteren Starts soll das nicht mehr geschehen. Um zu markieren, dass eine Initialisierung erfolgt ist, schreiben Sie an den Anfang des EEPROMs neben dem Reboot-Zähler noch eine Markierung. Die ersten Bytes des EEPROM sollen somit folgende Struktur haben:

Offset (Bytes)	Länge (Bytes)	Bedeutung
0	4	Initialisierungsmarkierung (`0x0F00B002`)
4	2	Neustartzähler
8	2	Versionszähler

Prüfen Sie beim Start, ob die 32-Bit-Initialisierungsmarkierung `0x0F00B002`[1] am Offset 0 im EEPROM steht und der 16-Bit-Versionszähler am Offset 8 mit der aktuellen Version Ihres Programms übereinstimmt.

Ist dies nicht der Fall, schreiben Sie diese beiden Werte an die entsprechende Stelle und setzen Sie den Neustartzähler zurück.

Ansonsten lesen Sie den Neustartzähler aus, erhöhen den Wert um eins und schreiben Sie ihn zurück.

Danach geben Sie den aktuellen Wert des Neustartzählers in einer Endlosschleife aus.

Wenn Sie größere Änderungen an Ihrem Programm vornehmen, erhöhen Sie die Version Ihres Programms um eins, so dass beim ersten Neustart eine Neuinitialisierung des EEPROM stattfindet.

Es ist empfehlenswert, zunächst die Logik des Programms zu prüfen. Verwenden Sie deshalb in einem ersten Schritt die EEPROM-Zugriffsfunktionen der AVR C-Library.

Hinweis:

- Die `eeprom_write_*`- und `eeprom_read_*`-Funktionen der AVR-LibC erwarten, dass die zu lesende oder zu schreibende Adresse im EEPROM als Pointer (z.B. vom Typ `uint32_t*` beim Schreiben oder `const uint32_t*` beim Lesen eines 32-Bit-Werts) übergeben werden. Konstante Adressen (wie in der obigen Tabelle angegeben) müssen erst in diesen Typ konvertiert werden (z.B. `(const uint32_t*)0` für einen Lesezugriff auf die Initialisierungsmarkierung).
- Beachten Sie die in der Tabelle angegebene Länge der jeweiligen Felder und nutzen Sie die passenden `eeprom_write_*`- und `eeprom_read_*`-Funktionen dafür.

Sobald Sie sich davon überzeugt haben, dass Ihr Programm prinzipiell funktioniert, ersetzen Sie die Aufrufe der verwendeten EEPROM-Zugriffsfunktionen

[1] Gelesen als „FOO-Boot“ – FOO ist der Raum C2.14 am Campus Furtwangen

durch eine eigene Implementierung, in der Sie direkt den EEPROM-Controller über seine Register ansteuern.

Einen relevanten Auszug aus dem Datenblatt des ATMega328P-Mikrocontrollers dazu finden Sie im Anhang A.

Beachten Sie die Byte-Reihenfolge (Little Endian/Big Endian) des AVR und die Besonderheiten beim Schreibzugriff. Eine Beschreibung hierzu finden Sie beim `EEPE`-Bit im `EECR`-Register.

Testen Sie Ihr Programm jeweils, indem Sie den Arduino mehrfach neustarten und beobachten, wie sich der Zähler verhält.

A Datenblattauszug ATmega328P – AVR Memories

8. AVR Memories

8.1 Overview

This section describes the different memories in the ATmega48A/PA/88A/PA/168A/PA/328/P. The AVR architecture has two main memory spaces, the Data Memory and the Program Memory space. In addition, the ATmega48A/PA/88A/PA/168A/PA/328/P features an EEPROM Memory for data storage. All three memory spaces are linear and regular.

8.2 In-System Reprogrammable Flash Program Memory

The ATmega48A/PA/88A/PA/168A/PA/328/P contains 4/8/16/32Kbytes On-chip In-System Reprogrammable Flash memory for program storage. Since all AVR instructions are 16 or 32 bits wide, the Flash is organized as 2/4/8/16K x 16. For software security, the Flash Program memory space is divided into two sections, Boot Loader Section and Application Program Section in ATmega88PA and ATmega168PA. See SPMEN description in section ”SPMCSR – Store Program Memory Control and Status Register” on page 287 for more details.

The Flash memory has an endurance of at least 10,000 write/erase cycles. The ATmega48A/PA/88A/PA/168A/PA/328/P Program Counter (PC) is 11/12/13/14 bits wide, thus addressing the 2/4/8/16K program memory locations. The operation of Boot Program section and associated Boot Lock bits for software protection are described in detail in ”Self-Programming the Flash, ATmega 48A/48PA” on page 264 and ”Boot Loader Support – Read-While-Write Self-Programming” on page 272. ”Memory Programming” on page 289 contains a detailed description on Flash Programming in SPI- or Parallel Programming mode.

Constant tables can be allocated within the entire program memory address space (see the LPM – Load Program Memory instruction description).

Timing diagrams for instruction fetch and execution are presented in ”Instruction Execution Timing” on page 23.

Figure 8-1. Program Memory Map ATmega 48A/48PA

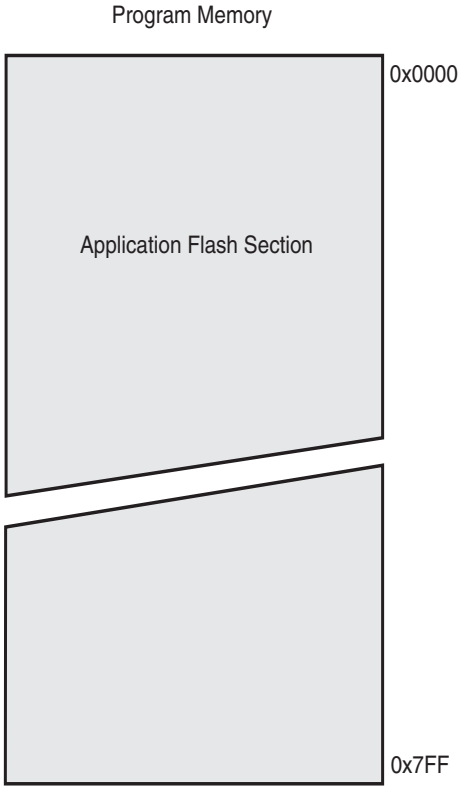
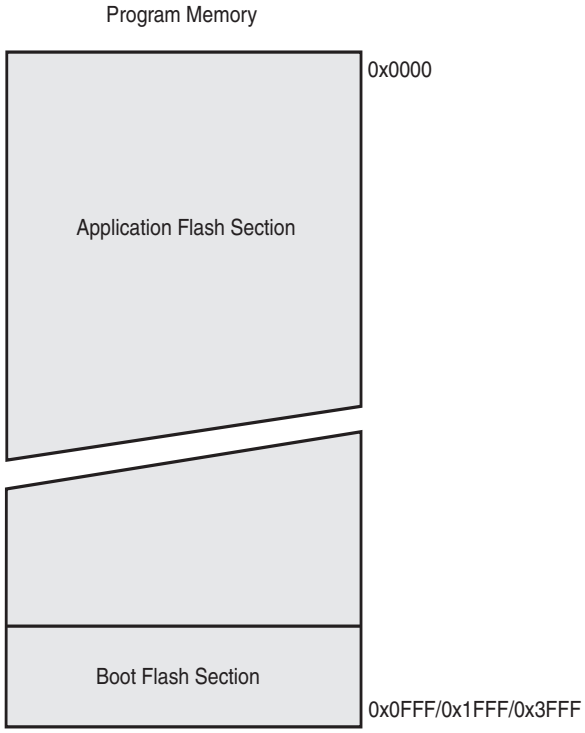


Figure 8-2. Program Memory Map ATmega88A, ATmega88PA, ATmega168A, ATmega168PA, ATmega328 and ATmega328P



8.3 SRAM Data Memory

Figure 8-3 shows how the ATmega48A/PA/88A/PA/168A/PA/328/P SRAM Memory is organized.

The ATmega48A/PA/88A/PA/168A/PA/328/P is a complex microcontroller with more peripheral units than can be supported within the 64 locations reserved in the Opcode for the IN and OUT instructions. For the Extended I/O space from 0x60 - 0xFF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used.

The lower 768/1280/1280/2303 data memory locations address both the Register File, the I/O memory, Extended I/O memory, and the internal data SRAM. The first 32 locations address the Register File, the next 64 location the standard I/O memory, then 160 locations of Extended I/O memory, and the next 512/1024/1024/2048 locations address the internal data SRAM.

The five different addressing modes for the data memory cover: Direct, Indirect with Displacement, Indirect, Indirect with Pre-decrement, and Indirect with Post-increment. In the Register File, registers R26 to R31 feature the indirect addressing pointer registers.

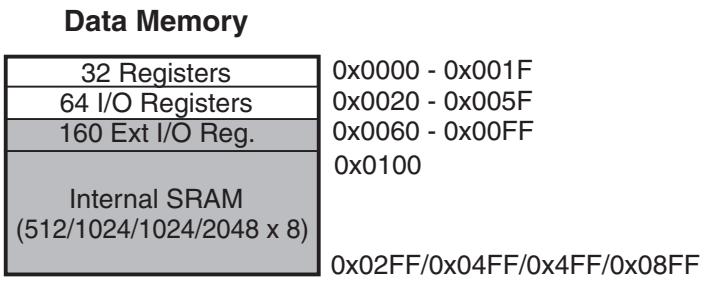
The direct addressing reaches the entire data space.

The Indirect with Displacement mode reaches 63 address locations from the base address given by the Y- or Z-register.

When using register indirect addressing modes with automatic pre-decrement and post-increment, the address registers X, Y, and Z are decremented or incremented.

The 32 general purpose working registers, 64 I/O Registers, 160 Extended I/O Registers, and the 512/1024/1024/2048 bytes of internal data SRAM in the ATmega48A/PA/88A/PA/168A/PA/328/P are all accessible through all these addressing modes. The Register File is described in ”General Purpose Register File” on page 20.

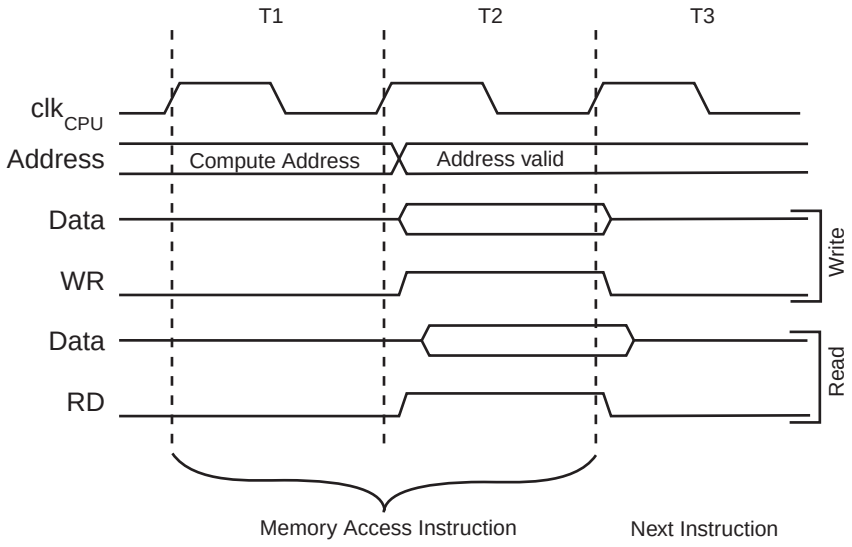
Figure 8-3. Data Memory Map



8.3.1 Data Memory Access Times

This section describes the general access timing concepts for internal memory access. The internal data SRAM access is performed in two clk_{CPU} cycles as described in Figure 8-4.

Figure 8-4. On-chip Data SRAM Access Cycles



8.4 EEPROM Data Memory

The ATmega48A/PA/88A/PA/168A/PA/328/P contains 256/512/512/1Kbytes 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 289 contains a detailed description on EEPROM Programming in SPI or Parallel Programming mode.

8.4.1 EEPROM Read/Write Access

The EEPROM Access Registers are accessible in the I/O space.

The write access time for the EEPROM is given in Table 8-2. A self-timing function, however, lets the user software detect when the next byte can be written. If the user code contains instructions that write the EEPROM, some precautions must be taken. In heavily filtered power supplies, V_{CC} is likely to rise or fall slowly on power-up/down. This causes the device for some period of time to run at a voltage lower than specified as minimum for the clock frequency used. See ”Preventing EEPROM Corruption” on page 30 for details on how to avoid problems in these situations.

In order to prevent unintentional EEPROM writes, a specific write procedure must be followed. Refer to the description of the EEPROM Control Register for details on this.

When the EEPROM is read, the CPU is halted for four clock cycles before the next instruction is executed. When the EEPROM is written, the CPU is halted for two clock cycles before the next instruction is executed.

8.4.2 Preventing EEPROM Corruption

During periods of low $V_{CC,}$ the EEPROM data can be corrupted because the supply voltage is too low for the CPU and the EEPROM to operate properly. These issues are the same as for board level systems using EEPROM, and the same design solutions should be applied.

An EEPROM data corruption can be caused by two situations when the voltage is too low. First, a regular write sequence to the EEPROM requires a minimum voltage to operate correctly. Secondly, the CPU itself can execute instructions incorrectly, if the supply voltage is too low.

EEPROM data corruption can easily be avoided by following this design recommendation:

Keep the AVR RESET active (low) during periods of insufficient power supply voltage. This can be done by enabling the internal Brown-out Detector (BOD). If the detection level of the internal BOD does not match the needed detection level, an external low V_{CC} reset Protection circuit can be used. If a reset occurs while a write operation is in progress, the write operation will be completed provided that the power supply voltage is sufficient.

8.5 I/O Memory

The I/O space definition of the ATmega48A/PA/88A/PA/168A/PA/328/P is shown in ”Register Summary” on page 621.

All ATmega48A/PA/88A/PA/168A/PA/328/P I/Os and peripherals are placed in the I/O space. All I/O locations may be accessed by the LD/LDS/LDD and ST/STS/STD instructions, transferring data between the 32 general purpose working registers and the I/O space. I/O Registers within the address range 0x00 - 0x1F are directly bit-accessible using the SBI and CBI instructions. In these registers, the value of single bits can be checked by using the SBIS and SBIC instructions. Refer to the instruction set section for more details. When using the I/O specific commands IN and OUT, the I/O addresses 0x00 - 0x3F must be used. When addressing I/O Registers as data space using LD and ST instructions, 0x20 must be added to these addresses. The ATmega48A/PA/88A/PA/168A/PA/328/P is a complex microcontroller with more peripheral units than can be supported within the 64 location reserved in Opcode for the IN and OUT instructions. For the Extended I/O space from 0x60 - 0xFF in SRAM, only the ST/STS/STD and LD/LDS/LDD instructions can be used.

For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written.

Some of the Status Flags are cleared by writing a logical one to them. Note that, unlike most other AVRs, the CBI and SBI instructions will only operate on the specified bit, and can therefore be used on registers containing such Status Flags. The CBI and SBI instructions work with registers 0x00 to 0x1F only.

The I/O and peripherals control registers are explained in later sections.

8.5.1 General Purpose I/O Registers

The ATmega48A/PA/88A/PA/168A/PA/328/P contains three General Purpose I/O Registers. These registers can be used for storing any information, and they are particularly useful for storing global variables and Status Flags. General Purpose I/O Registers within the address range 0x00 - 0x1F are directly bit-accessible using the SBI, CBI, SBIS, and SBIC instructions.

8.6 Register Description

8.6.1 EEARH and EEARL – The EEPROM Address Register

Bit	15	14	13	12	11	10	9	8	
0x22 (0x42)	–	–	–	–	–	–	EEAR9(1)	EEAR8(1)	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 [15:10] – Reserved**

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

- **Bits 9:0 – EEAR[9:0]: EEPROM Address**

The EEPROM Address Registers – EEARH and EEARL specify the EEPROM address in the 256/512/512/1Kbytes 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.

Note: 1. EEAR9 and EEAR8 are unused bits in ATmega 48A/48PA and must always be written to zero.

8.6.2 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 – EEDR[7:0]: 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.

8.6.3 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 – Reserved**

These bits are reserved bits in the ATmega48A/PA/88A/PA/168A/PA/328/P 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 8-1. While EEPE 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 8-1. EEPROM Mode Bits

EEPM1	EEPM0	Programming Time	Operation
0	0	3.4ms	Erase and Write in one operation (Atomic Operation)
0	1	1.8ms	Erase Only
1	0	1.8ms	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.

- **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 SPMEN 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” on page 272 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.

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 8-2 lists the typical programming time for EEPROM access from the CPU.

Table 8-2. EEPROM Programming Time

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

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);
}
```

The next code examples show assembly and C functions for reading the EEPROM. The examples assume that interrupts are controlled so that no interrupts will occur during execution of these functions.

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;
}
```

8.6.4 GPIOR2 – General Purpose I/O Register 2

Bit	7	6	5	4	3	2	1	0	
0x2B (0x4B)	MSB							LSB	GPIOR2
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	

8.6.5 GPIOR1 – General Purpose I/O Register 1

Bit	7	6	5	4	3	2	1	0	
0x2A (0x4A)	MSB							LSB	GPIOR1
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	

8.6.6 GPIOR0 – General Purpose I/O Register 0

Bit	7	6	5	4	3	2	1	0	
0x1E (0x3E)	MSB							LSB	GPIOR0
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	