EMBEDDS

Accessing AVR EEPROM memory in AVRGCC

BY ADMIN AVR TUTORIAL

AVR microcontrollers have some amount of EEPROM memory on-chip. For instance, Atmega328 has 1K of byte-addressable EEPROM. EEPROM memory can be used to store and read variables during program execution and is nonvolatile. It means that it retains values when the power supply is off. EEPROM memory comes in handy when we need to store calibration values, remember program state before powering off (or power failure) or store constants in EEPROM memory when you are short of program memory space, especially when using smaller AVRs. Think of a simple security system – EEPROM is the ideal place to store lock combinations, code sequences, and passwords. AVR Datasheets claim that EEPROM can withhold at least 100000 writes/erase cycles.

Accessing AVR EEPROM memory

Standard C functions don't understand how one or another memory is accessed. So reading and writing EEPROM has to be done by following a special logical system. Simple EEPROM byte read and write operations have to be done via special registers. Atmega328 has the following registers to control EEPROM:

- 16-bit EEAR (EEARH and EEARL) – EEPROM address register;
- EEDR – EEPROM 8-bit data register;
- EECR – EEPROM control register.

As Atmega328 has 1K of EEPROM memory highest cell address will be 0x3FF, meaning that only 10 bits of 16-bit EEAR register are needed. The EEPROM writing process is controlled via the EECR

- Write data to EEDR;
- Set EEMPE bit while keeping EEPE bit zero;
- In four clock cycles, set EEPE bit to initiate writing.

You will find a nice read and write examples in the datasheet, so there is no need to repeat this. Let's do a more interesting thing – use an interrupt to write to EEPROM. As you may know, Atmega328 has one dedicated interrupt EE_READY_vect, that may be set to occur whenever EEPROM becomes ready for writing or reading. Usually, you would have to poll for EEPE bit become zero in the loop – this requires active processing power. Interrupt-driven EEPROM writing may be more efficient, especially when memory blocs have to be accessed. Let's write a simple Interruption-driven AVR EEPROM writing routine and write a simple message string.

```
#include <avr/io.h>
#include <avr/interrupt.h>
//EEPROM address initial 0
volatile uint16_t eepromaddress;
//string array index initial 0
volatile uint8_t i;
//message to be written to EEPROM
uint8_t message[] ="Write to EEPROM";
//EEPROM writing ISR
ISR(EE_READY_vect)
{
/*check if not end of string and address
didn't reach end of EEPROM*/
if ((message[i])&&(eepromaddress<=E2END))
{
    //loads address and increments for next load
    EEAR=eepromaddress++;
    //loads current byte and increments index for next load
    EEDR=message[i++];
    //master write enable
    EECR|=(1<<EEMPE);
    //strobe eeprom write
    EECR|=(1<<EEPE);
}
```

- else

```
int main(void)
{
//Enable eeprom ready interrupt
EECR |= (1<<EERIE);
//enable global interrupts
sei();
    while(1)
    {
        //do other tasks or sleep
    }
}
```

As you can see, ISR takes bytes from the message buffer and writes them to EEPROM memory until the buffer is empty or the EEPROM end is reached. Using EE_READY interrupt leaves EEPROM writing completely to hardware and frees MCU resources for other tasks. In other words, EEPROM itself asks for another byte when ready. This way, you improve performance and reduce power consumption. The same interrupt source can be used for reading EEPROM.

Using EEPROM library from AVRLibC

We discussed how you could access EEPROM memory by using special registers and safe logic. TO avoid possible errors, it is logical to use standard *eeprom.h* library from AVRLibC that comes with AVRGCC tools. To start working with EEPROM routines, we need to include the EEPROM library into our source code:

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

This opens access to several useful routines that allow us to read/write/update bytes, words, double words, floats, and even memory blocks. Let's see how simple byte write and read looks when using *eeprom.h* library.

```
#include <avr/io.h>
#include <avr/eeprom.h>
```

```
{
//put EEPROM data to this variable
volatile uint8_t EEByte;
//write EEPROM
eeprom_write_byte ((uint8_t*)EEADDR, EEDATA);
//read from EEPROM
EEByte = eeprom_read_byte((uint8_t*)EEADDR);
while(1)
{
    //do nothing
}
}
```

We have written the number byte value of 100 to EEPROM address location 0x3FF and read back to some variables. The same expressions can be used to write a word, double word, or even float. Just use one of the proper functions like *eeprom_read_word()*.

Working with AVR EEPROM memory blocks

We have discussed one way of dealing with EEPROM memory blocks – using interrupt-driven writing from the buffer. But if you don't want to write your own routine, you can use one of the library's standard functions. There are three functions for this:

```
void eeprom_write_block (const void *__src, void *__dst, size_t __n);

void eeprom_read_block (void *__dst, const void *__src, size_t __n);

void eeprom_update_block (const void *__src, void *__dst, size_t __n);
```

You can write, read, and update memory blocks in EEPROM. Functions may not look usual to some of you. They use void pointers instead of usual data-types like uint8_t. The good thing is that these functions are universal, allowing them to deal with any data type. Let's see how this works. Lets take *eeprom_write_block()* function it has three arguments:

- const void *__src – pointer to RAM location;

There is no significant difference in what type of pointers are used. The only thing necessary to us is the number of bytes that need to be written. Let's write a simple program demonstrating block write and read routines.

```
#include <avr/io.h>
#include <avr/eeprom.h>
//start from first EEPROM cell
#define EEADDR 0
//block size to be copied
#define BLKSIZE 16
//message to be written to EEPROM
uint8_t message[] ="Write to EEPROM";
int main(void)
{
//where block has to be read
uint8_t readblock[BLKSIZE];
//write block EEPROM
eeprom_write_block((const void *)message, (void *)EEADDR, BLKSIZE);
//read block from EEPROM
eeprom_read_block ((void *)readblock, (const void *)EEADDR, BLKSIZE);
while(1)
{
    //do nothing
}
}
```

As we see in the example we only need to typecast pointers to the void pointers like *(void *)readblock*, so they meet function requirements – this action doesn't affect pointer value itself. Our real concern now is to read and write the correct number of bytes, and that's it.

Allocating variables in EEPROM with EEMEM attribute

We learned how to perform EEPROM write, and read operations by defining special addresses. This way, we have to keep track of EEPROM addresses that are used in functions. This is OK in some cases, but it becomes unpractical if we need to store lots of EEPROM data. Why not define an eeprom variable where the address would be allocated automatically by the compiler. This is

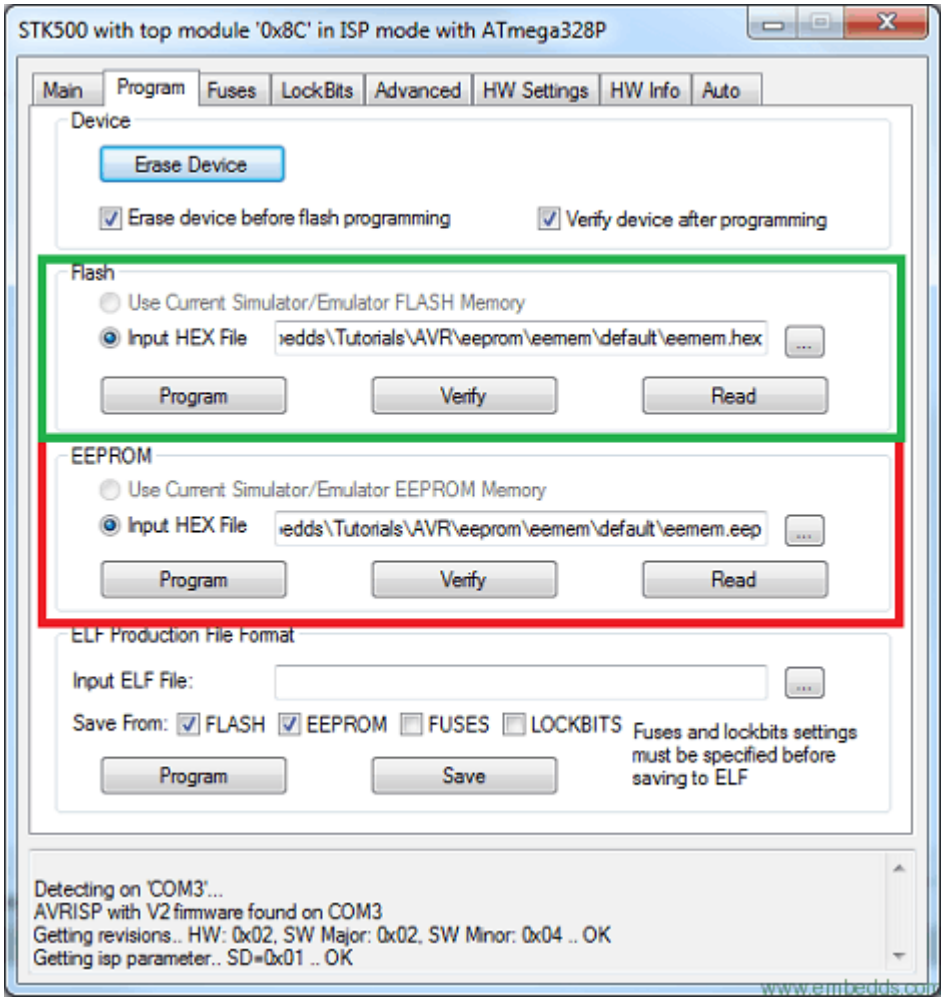
```
#include <avr/io.h>
#include <avr/eeprom.h>
#define BLKSIZE 16
#define DATA 0x10
uint8_t EEMEM eechar;
uint16_t EEMEM eeword=0x1234;
uint8_t EEMEM eestring[] = "Write to EEPROM";
int main(void)
{
uint16_t sramword;
//where block has to be read
uint8_t readblock[BLKSIZE];
//write byte to location eechar
eeprom_write_byte(&eechar, DATA);
sramword=eeprom_read_word(&eeword);
//read block from EEPROM
eeprom_read_block ((void *)readblock, (const void *)eestring, BLKSIZE);
while(1)
{
    //do nothing
}
}
```

We have declared one EEPROM variable:

```
uint8_t EEMEM eechar;
```

This will automatically allocate one byte in EEPROM memory with an initial zero value. The second two variables have some defined values:

```
uint16_t EEMEM eeword=0x1234;
uint8_t EEMEM eestring[] = "Write to EEPROM";
```



When the chip is being flashed, *the .eep* file must also be uploaded to have initial EEPROM values. Otherwise, your program will fail to try to use them.

This is the end of another tutorial. Comments and corrections are always welcome.

TAGGED AVR EEPROM memory, AVR Tutorial, EEMEM, write eeprom.

« Controlling 4 servos with Arduino-Python interface

Small fully featured atxmega32 scope »

4 Comments

This should be BLKSIZE instead of 15 else you'll get a buffer overflow.

admin
JANUARY 5, 2011 AT 2:06 AM

Oops missed that. Thanks.

bud
MARCH 27, 2011 AT 9:17 AM

thank you this is the most complete eeprom tut i have found yet

FAITH + 1
JUNE 10, 2016 AT 8:30 PM

nice tutorial.

SEARCH

CATEGORIES

AVR Projects Arduino PIC Projects Misc Business Technology Other MCU Projects Linux board projects FPGA Projects ARM Cortex Internet MSP430 Projects Games Education Software PIC32 ARM7 Projects AVR Tutorial MSC-51 Projects PCB Health Handy Circuits Guide Lifestyle 68HC Projects ARM Cortex Tutorial BASIC Stamp Finances Casino ChipKIT Projects MSP430 Tutorial Raspberry Pi ZiLOG Marketing

RECENT COMMENTS

Ed Woodrick on Is Investing in High-End Ethernet Cable Worth it for Programmers?

admin on Order PCBs from PCBWay and get a free mask to fight COVID-19

Ed on Order PCBs from PCBWay and get a free mask to fight COVID-19

John Jennings on Programming STM32-Discovery using GNU tools. Linker script

Ashok on Programming STM32-Discovery using GNU tools. Startup code

RECENT TUTORIALS

Programming AVR I2C interface

Using Direct Memory Access (DMA) in STM32 projects

RECENT POSTS

6 Benefits of Using Solar Inverter for Residential Energy

How to make pancakes healthier with Kratom?

The Top 5 Photo Editing Software Choices

CRM vs. ERP — Do You Know the Difference?

Five Tips To Be Successful In Data Science

TOP POSTS & PAGES

TFTLCD Menu interface for Arduino

Programming AVR I2C interface

Programming STM32-Discovery using GNU tools. Linker script

Using Direct Memory Access (DMA) in STM32 projects

Accessing AVR EEPROM memory in AVRGCC