Using the EEPROM memory in AVR-GCC

Dean Camera

April 11, 2012

****

Contents

Chapter 1

Introduction

1.1 What is the EEPROM memory and why would I use it?

Most of the 8-bit AVRs in Atmel's product line contain at least some internal EEPROM memory. EEPROM, short for **E**lectronically **E**rasable **R**ead-**O**nly **M**emory, is a form of non-volatile memory with a reasonably long lifespan. Because it is non-volatile, it will retain its information during periods of no power and thus is a great place for storing data such as device parameters and system configuration at runtime so that it can persist between resets of the application processor.

One important fact to note is that the AVR's internal EEPROM memory has a limited lifespan of 100,000 writes per EEPROM page — reads are unlimited. Keep this in mind in your application; try to keep writes to a minimum, so that you only write the least amount of information required for your application each time you update the EEPROM.

1.2 How is is accessed?

The AVR's internal EEPROM is accessed via special registers inside the AVR, which control the address to be written to (EEPROM uses byte addressing), the data to be written (or the data which has been read) as well as the flags to instruct the EEPROM controller to perform the requested read or write operation.

The C language does not have any standards mandating how memory other than a single flat model (SRAM in AVRs) is accessed or addressed. Because of this, just like storing arbitrary data into flash memory via your program, every compiler's standard library has a unique implementation based on what the author believed was the most logical system.

This tutorial will focus specifically on the AVR-GCC compiler; other compilers may have alternative syntax and APIs.

Chapter 2

The avr-libc EEPROM functions

2.1 Including the avr-libc EEPROM header

The *avr-libc* library, included with WinAVR and (more recently) the Atmel AVR Toolchain, contains prebuilt library routines for EEPROM access and manipulation to simplify its use in user applications. Before we can make use of those routines, we need to include the EEPROM library header `<avr/eeprom.h>` with the preprocessor directive `#include <avr/eeprom.h>`.

Once added to your application, you now have access to the EEPROM memory, via the routines now provided by the `<avr/eeprom.h>` module of avr-libc. There are three main types of EEPROM access: byte, word and block. Each type has three types of functions; a write, an update, and a read variant. The names of the routines exposed by our new headers are:

```
uint8_t eeprom_read_byte (const uint8_t *addr)
void eeprom_write_byte (uint8_t *addr, uint8_t value)
void eeprom_update_byte (uint8_t *addr, uint8_t value)

uint16_t eeprom_read_word (const uint16_t *addr)
void eeprom_write_word (uint16_t *addr, uint16_t value)
void eeprom_update_word (uint16_t *addr, uint16_t value)

void eeprom_read_block (void *pointer_ram, const void *pointer_eeprom, size_t n)
void eeprom_write_block (void *pointer_ram, const void *pointer_eeprom, size_t n)
void eeprom_update_block (void *pointer_ram, const void *pointer_eeprom, size_t n)
```

In AVR-GCC, a word is two bytes long, while a block is an arbitrary number of bytes which you supply (think string buffers, or customer data types you create).

2.2 The update vs. write functions

Historically, there were only two types of EEPROM functions per data type; a write function, and a read function. In the case of the EEPROM write functions, these functions simply wrote out the requested data to the EEPROM without any checking performed; this resulted in a reduced EEPROM lifespan if the data to be written already matches the current contents of the EEPROM cell. To reduce the wear on the AVR's limited lifespan EEPROM, the new update functions were added which only perform an EEPROM write if the data differs from the current cell contents. The old write functions are still kept around for compatibility with older applications, however **the new update functions should be used instead of the old write functions in all new applications**.

Chapter 3

Reading data from the EEPROM

To start, lets try a simple example and try to read a single byte of EEPROM memory, let's say at location 46. Our code might look like this:

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

void main(void)
{
    uint8_t ByteOfData;

    ByteOfData = eeprom_read_byte((uint8_t*)46);
}
```

This will read out location 46 of the EEPROM and put it into our new variable named `ByteOfData`. How does it work? Firstly, we declare our byte variable, which I'm sure you're familiar with:

```
uint8_t ByteOfData;
```

Now, we then call our `eeprom_read_byte()` routine, which expects a pointer to a byte in the EEPROM space. We're working with a constant and known address value of 46, so we add in the typecast to transform that number 46 into a pointer for the `eeprom_read_byte()` function to make the compiler happy.

EEPROM words (two bytes in size) can be written and read in much the same way, except they require a pointer to an int:

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

void main(void)
{
    uint16_t WordOfData;

    WordOfData = eeprom_read_word((uint16_t*)46);
}
```

Chapter 4

Writing data to the EEPROM

Alright, so now we know how to read bytes and words from the EEPROM, but how do we *write* data? The same principals as reading EEPROM data apply, except now we need to supply the data to write as a second argument to the EEPROM functions. As explained earlier, we will use the EEPROM update functions rather than the old write functions to preserve the EEPROM lifespan where possible.

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

void main(void)
{
    uint8_t ByteOfData;

    ByteOfData = 0x12;

    eeprom_update_byte((uint8_t*)46, ByteOfData);
}
```

Unsuprisingly, the process for writing words of data to the EEPROM follows the same pattern:

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

void main(void)
{
    uint16_t WordOfData;

    WordOfData = 0x1232;

    eeprom_update_word((uint8_t*)46, WordOfData);
}
```

Chapter 5

EEPROM Block Access

Now we can read and write bytes and words to EEPROM, but what about larger datatypes, or strings? This is where the block commands come in.

The block commands of the avr-libc `<avr/eeprom.h>` header differ from the word and byte functions shown above. Unlike it's smaller cousins, the block commands are both routines which return nothing, and thus operate on a variable you supply as a parameter. Let's look at the function declaration for the block reading command.

```
void eeprom_read_block (void *pointer_ram, const void *pointer_eeprom, size_t n)
```

Wow! It looks hard, but in practise it's not. It may be using a concept you're not familiar with however, `void`-type pointers.

Normal pointers specify the size of the data type in their declaration (or typecast), for example `uint8_t*` is a pointer to an unsigned byte and `int16_t*` is a pointer to a signed int. But `void` is not a variable type we can create, so what does it mean?

A `void` type pointer is useful when the exact type of the data being pointed to is not known by a function, or is unimportant. A `void` is mearly a pointer to a memory location, with the data stored *at* that location being important but the *type* of data stored at that location not being important. Why is a `void` pointer used?

Using a `void` pointer means that the block read/write functions can deal with any datatype you like, since all the function does is copy data from one address space to another. The function doesn't care what data it copies, only that it copies the correct number of bytes.

Ok, that's enough of a derail — back to our function. The block commands expect a `void*` pointer to a RAM location, a `void*` pointer to an EEPROM location and a value specifying the number of bytes to be written or read from the buffers. In our first example let's try to read out 10 bytes of memory starting from EEPROM address 12 into a string.

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

void main(void)
{
    uint8_t StringOfData[10];

    eeprom_read_block((void*)&StringOfData, (const void*)12, 10);
}
```

Again, looks hard doesn't it! But it isn't; let's break down the arguments we're sending to the `eeprom_read_block()` function.

```
(void*)&StringOfData
```

The first argument is the RAM pointer. The `eeprom_read_block()` routine modifes our RAM buffer and so the pointer type it's expecting is a `void*` pointer. Our buffer is of the type `uint8_t`, so we need to typecast its address to the necessary `void` type.

```
(const void*)12
```

Reading the EEPROM won't change it, so the second parameter is a pointer of the type `const void`. We're currently using a fixed constant as an address, so we need to typecast that constant to a pointer just like with the byte and word reading examples.

```
10
```

Obviously, this the number of bytes to be read. We're using a constant but a variable could be supplied instead.

The block update function is used in the same manner, except for the update routine the RAM pointer is of the type `const void` and the EEPROM is just a plain `void` pointer, as the data is sourced from the RAM location and written to the EEPROM location:

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

void main(void)
{
    uint8_t StringOfData[10] = "TEST";

    eeprom_update_block((void*)&StringOfData, (const void*)12, 10);
}
```

Chapter 6

Using the EEMEM attribute

You should now know how to read and write to the EEPROM using fixed addresses. But that's not very practical! In a large application, maintaining all the fixed addresses is an absolute nightmare at best. But there's a solution to push this work onto the compiler — the `EEMEM` attribute.

Defined by the same `<avr/eeprom.h` header that gives us our access functions for the EEPROM, the `EEMEM` attribute instructs the GCC compiler to assign your variable into the EEPROM address space, instead of the SRAM address space like a normal variable. For example, we could add the following code to our application:

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

uint8_t  EEMEM NonVolatileChar;
uint16_t EEMEM NonVolatileInt;
uint8_t  EEMEM NonVolatileString[10];
```

And the compiler will automatically allocate addresses for each of the EEPROM variables within the EEPROM address space, starting from location 0. Although the compiler allocates addresses for the EEMEM variables you declare, it cannot directly access them. This means that the following code will **NOT** work as intended:

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

uint8_t  EEMEM NonVolatileChar;
uint16_t EEMEM NonVolatileInt;
uint8_t  EEMEM NonVolatileString[10];

int main(void)
{
    uint8_t SRAMchar;

    SRAMchar = NonVolatileChar;
}
```

That code will instead assign the RAM variable "SRAMchar" to whatever is located in SRAM at the address of "NonVolatileChar", and so the result will be incorrect. Like variables stored in program memory, we need to still use the eeprom read and write routines discussed above to access the seperate EEPROM memory space.

So lets try to fix our broken code. We're trying to read out a single byte of memory. Let's try to use the `eeprom_read_byte()` routine.

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

uint8_t  EEMEM NonVolatileChar;
uint16_t EEMEM NonVolatileInt;
uint8_t  EEMEM NonVolatileString[10];

int main(void)
{
```

```
    uint8_t SRAMchar;

    SRAMchar = eeprom_read_byte(&NonVolatileChar);
}
```

It works! Why don't we try to read out the other variables while we're at it. Our second variable is an int, so we need the `eeprom_read_word()` routine:

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

uint8_t  EEMEM NonVolatileChar;
uint16_t EEMEM NonVolatileInt;
uint8_t  EEMEM NonVolatileString[10];

int main(void)
{
    uint8_t  SRAMchar;
    uint16_t SRAMint;

    SRAMchar = eeprom_read_byte(&NonVolatileChar);
    SRAMint  = eeprom_read_word(&NonVolatileInt);
}
```

And our last one is a string, so we'll have to use the block read command:

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

uint8_t  EEMEM NonVolatileChar;
uint16_t EEMEM NonVolatileInt;
uint8_t  EEMEM NonVolatileString[10];

int main(void)
{
    uint8_t  SRAMchar;
    uint16_t SRAMint;
    uint8_t  SRAMstring[10];

    SRAMchar = eeprom_read_byte(&NonVolatileChar);
    SRAMint  = eeprom_read_word(&NonVolatileInt);
    eeprom_read_block((void*)&SRAMstring, (const void*)&NonVolatileString, 10);
}
```

Chapter 7

Setting initial values

Finally, I'll discuss the issue of setting an initial value to your EEPROM variables.

Upon compilation of your program with the default makefile, GCC will output two Intel-HEX format files. One will be your `.HEX` which contains your program data (and which is loaded into your AVR's memory), and the other will be a `.EEP` file. The `.EEP` file contains the default EEPROM values, which you can load into your AVR via your programmer's EEPROM programming functions.

To set a default EEPROM value in GCC, simply assign a value to your `EEMEM` variable, like thus:

```
uint8_t EEMEM SomeVariable = 12;
```

You can then program in the resulting `.EEP` file into the target's EEPROM using your chosen programming software. This is a seperate operation to programming in the application from the generated `.HEX` file – if you forget this, your AVR's EEPROM won't have the correct initial values! You should devise a way in your application to check for this possibility to ensure that no problems will arise if the initial values aren't loaded correctly, via some sort of protection scheme (eg, loading in several values and checking against known start values).