Using the EEPROM memory in AVR-GCC

Tutorial (c) Dean Camera, 2006.

dean_camera@hotmail.com

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

Most of the AVRs in Atmel's product line contain at least *some* internal EEPROM memory. EEPROM, short for *Electronically Erasable Read-Only memory*, is a form of non-volatile memory with a reasonably long lifespan. Because it is non-volatile, it will retain it's information during periods of no AVR power and thus is a great place for storing sparingly changing data such as device parameters.

The AVR internal EEPROM memory has a limited lifespan of 1,000,000 writes - reads are unlimited.

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 a write or a read.

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 data into program memory via your program, every compiler has a unique implementation based on what the author believed was the most logical system.

This tutorial will focus on the GCC compiler.

Using the AVRLibC EEPROM library routines:

The AVRLibC, included with WinAVR, contains prebuilt library routines for EEPROM access and manipulation. Before we can make use of those routines, we need to include the eeprom library header:

Code:

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

At the moment, we now have access to the eeprom memory, via the routines now proveded by eeprom.h. There are three main types of EEPROM access: byte, word and block. Each type has both

a write and a read variant, for obvious reasons. The names of the routines exposed by our new headers are:

Quote:

• uint8_t eeprom_read_byte (const uint8_t *addr)
• void eeprom_write_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_read_block (void *pointer_ram, const void *pointer_eeprom, size_t n)
• void eeprom_write_block (const void *pointer_ram, void *pointer_eeprom, size_t n)

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

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:

Code:

```
#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:

Code:

```
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.

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

Code:

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

void main(void)
{
  uint16_t WordOfData;
```

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

But what about larger datatypes, or strings? This is where the block commands come in.

EEPROM Block Access

The block commands of the AVRLibC 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 internally. Let's look at the function declaration for the block reading command.

Code:

```
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 familar with though, 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 data type, so what does it mean?

Void pointers are 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.

Code:

```
#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.

Code:

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

The first argument is the RAM pointer. The eeprom_read_block routine modifies 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 it's address to the necessary void type.

Code:

```
(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.

Code:

```
10
```

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

The block write function is used in the same manner, except for the write routine the RAM pointer is of the type const void and the EEPROM is just a plain void pointer.

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 - the EEMEM attribute.

Defined by the same eeprom.h header, 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 allocate addresses for several variables, like so:

Code:

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

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

Although the compiler allocates addresses to your EEMEM variables, it cannot directly access them. For example, the following code will **NOT** function as intended:

Code:

```
#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 (see tutorial here) we need to still use the eeprom read and write routines discussed above.

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:

Code:

```
#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:

Code:

```
#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:

Code:

```
#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;

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

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:

Code:

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

Note that if the EEP file isn't loaded, your program will most likely run with the entire EEPROM being set to 0xFF, or at least unknown data. You should devise a way to ensure that no problems will arise via some sort of protection scheme (eg, loading in several values and checking against known start values).