

Join the Stack Overflow Community

Stack Overflow is a community of 4.7 million programmers, just like you, helping each other.
Join them; it only takes a minute:

[Sign up](#)

Is there a general algorithm for microcontroller EEPROM wear leveling?

I'm working on an Arduino library that will maximize the life of the AVR's EEPROM. It takes the number of variables you want to store and does the rest. This is my attempt, which does not work in all cases.

Background information

Atmel says each memory cell is rated for 100,000 write/erase cycles. They also provide an [application note](#), which describes how to perform wear levelling. Here is a summary of the application note.

By alternating writes across two memory addresses, we can increase the erase/write to 200,000 cycles. Three memory addresses gives you 300,000 erase/write cycles and so on. To automate this process, a status buffer is used to keep track of where the next write should be. The status buffer must also be the same length as the parameter buffer because wear leveling must be performed on it as well. Since we can't store the index of the next write we increment the corresponding index in the status buffer.

Here is an example.

```
<----- EEPROM ----->
0                                     N
-----
Parameter Buffer | Status Buffer |
-----

Initial state.
[ 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 ]

First write is a 7. The corresponding position
in the status buffer is changed to previous value + 1.
Both buffers are circular.
[ 7 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 ]

A second value, 4, is written to the parameter buffer
and the second element in the status buffer becomes
the previous element, 1 in this case, plus 1.
[ 7 | 4 | 0 | 0 | 0 | 0 | 1 | 2 | 0 | 0 | 0 ]

And so on
[ 7 | 4 | 9 | 0 | 0 | 0 | 1 | 2 | 3 | 0 | 0 ]
```

To determine where the next write should occur, we look at the difference between elements. If the previous element + 1 does NOT equal the next element then that is where the next write should occur. For example:

```
Compute the differences by starting at the first
element in the status buffer and wrapping around.
General algo: previous element + 1 = current element
1st element: 0 + 1 = 1 = 1st element (move on)
2nd element: 1 + 1 = 2 = 2nd element (move on)
3rd element: 2 + 1 = 3 = 3rd element (move on)
4th element: 3 + 1 = 4 != 4th element (next write occurs here)

[ 7 | 4 | 9 | 0 | 0 | 0 | 1 | 2 | 3 | 0 | 0 ]
      ^               ^
      |               |
```

Another edge case to consider is when the incrementing values wrap around at 256 because we are writing bytes. With the following buffer we know the next write is at the 3rd element because $255 + 1 = 0 \neq 250$ assuming we are using byte arithmetic.

```
[ x | x | x | x | x | x | 254 | 255 | 250 | 251 | 252 | 253 ]
                        ^
                        |

After we write at element 3 the status buffer is incremented
to the next value using byte arithmetic and looks like this.
255 + 1 = 0 (wrap around)
0 + 1 != 251 (next write is here)

[ x | x | x | x | x | x | 254 | 255 | 0 | 251 | 252 | 253 ]
                        ^
                        |
```

These examples above show how to extend the life of the EEPROM for one variable. For multiple variables imagine segmenting the EEPROM into multiple segments with the same data structure, but smaller buffers.

Problem

I have working code for what I described above. My problem is the algorithm does not work when the buffer length ≥ 256 . Here is what happens

```
Buffer length of 256. The last zero is from
the initial state of the buffer. At every index
previous element + 1 == next element. No way to
know where the next write should be.

<----- Status Buffer ----->
[ 1 | 2 | ... | 253 | 254 | 255 | 0 ]

A similar problem occurs when the buffer length
```

```


is greater than 256. A lookup for a read will think
the next element is at the first 0 when it should be at
255.
<----- Status Buffer ----->
[ 1 | 2 | ... | 253 | 254 | 255 | 0 | 0 | 0 ]

```

Question

How can I solve the above problem? Is there a better way to track where the next element should be written?

[arduino](#)
[avr](#)
[eeprom](#)



edited Jun 17 '12 at 10:45

Peter Mortensen

10.2k1369107



asked May 19 '12 at 17:42

Nabil

14827

Please add a link to your library to the "[Libraries for Arduino](#)" page. Thank you. – David Cary Aug 29 '14 at 22:55

1 Answer

Some thoughts about general EEPROM lifetime extension:

- EEPROM cells are usually written (by the hardware) in a two step operation: first, the cell is erased, that is, set to all ones (0b11111111 = 0xff), then the bits to be written (effectively only those that are 0) are actually written. Bits can only be set to 0 by an actual write operation. Changing a bit from 0 to 1 requires the whole cell to be erased and then re-writing the new value.
- If an EEPROM cell already contains the same value that is to be written to it - which may be the case for more or less of the data to be (re-)written - there is no need to write to the cell at all, reducing wear for that write operation to 0. One might want to check the cells content to decide if it needs to be written at all instead of always writing a new value to it.
- The combination of the above leads to an approach where a cell is only erased before a write, if there are any 1 bits in the new value where the stored value has a 0 bit (that is, if `StoredValue & NewValue != NewValue`). There is no need to erase the cell if there are no 0 -> 1 bit transitions required for the new value (`StoredValue & NewValue == NewValue`).
- The AVR provides separate instructions for erasing and writing to, respectively, an EEPROM cell to support the mechanisms mentioned above.
- The worst-case speed of a data transfer to EEPROM will of course drop when performing a read-compare-erase-write instead of just an erase-write operation. However, this has the potential to completely skip erase-write operations for some/most cells which may reduce the relative speed penalty.

For your present problem, think about the above points: Why not use single bits to store your next write position?

Example:

Status buffer is initialized to all ones:

```

Bit number: 0123 4567 89AB CDEF
Value:      1111 1111 1111 1111

```

Before accessing a value in EEPROM, find the first 1 bit in your status buffer. The number of that bit represents the address of the current "head" of your (circular) parameter buffer.

Each time you advance the parameter buffer, set the next bit in your status buffer to 0:

```

Bit number: 0123 4567 89AB CDEF
Value:      0111 1111 1111 1111

```

then

```

Value:      0011 1111 1111 1111

```

then

```

Value:      0001 1111 1111 1111

```

and so on.

This can be done **without** erasing the whole cell and will thus only "wear" a single bit of your status buffer for every update - if the data written has only a single 0 bit, too. To turn, for instance, a stored value of 0111 to the new value 0011 the data to write should be 1011 (`data = (newValue XOR oldValue) XOR 0xff`), leaving all bits untouched except for the single one we actually want to change.

Once the status buffer is exhausted (all 0) it is erased in full, and it all starts over again.

A definite plus here is that only a single bit of status needs to be maintained per unit of the parameter buffer, which consumes only 1/8 of the memory compared to the Atmel application note. ~~Besides, finding the next write location will also be much faster since only 1/8 of the read operations on the status buffer will be needed.~~ (Edit: Not true because EEPROM reads come at almost zero cost performance-wise, while the required bit-shifting may take some dozens of cycles.)

Another note, though: Do you think it is actually useful to use 256+ parameter buffer units? The units would become quite small when dealing with, for example, 1024 bytes of total available EEPROM on the device. - And 100000 cycles multiplied by 256 is quite a huge number of write operations and if that large a number seems to be required, there probably is something wrong in the algorithm or EEPROM should not be used for the purpose at all. As an alternative, external [NVRAM](#) would be a good choice in some scenarios.

Access time may be an aspect here, too: When trying to look up and read an element of, say, 3 bytes size in the parameter buffer with a 256 byte status buffer, **256** (+3) read operations will be


needed in the worst case - a tremendous overhead!

There is a very illustrative document about the workings of EEPROM cells, including the how-and-why of deterioration:

[STMicroelectronics: "How a designer can make the most of STMicroelectronics Serial EEPROMs", application note AN2014](#)

edited May 5 '14 at 9:42

answered May 19 '12 at 23:34



JimmyB

7,401 1 9 30

Thanks for the comprehensive answer. One follow up question. You said, "This can be done without erasing the whole cell and will thus only "wear" a single bit of your status buffer for every update." Does a bit "wear" if you write a 0? I also agree that the 256+ parameter buffer is large, but I want to give users of the library that option. Most people will use more than 1 variable I'm sure. – Nabil May 20 '12 at 16:55

3 Yes and no: Each bit basically gets worn when it *changes* its state. So, writing a 0 to a location where there already is a 0 or writing a 1 to where a 1 was does not produce significant wear. When writing 0011 to a cell previously containing 0111 only a single bit has to change from 1 to 0. Changes from 1 to 0 are performed by "writes", changes from 0 to 1 can only be performed by "erases"; erase always affects the whole byte. If no erase is needed *and* none is actually performed this alone saves -best case- 50%+ of the total wear for an erase-write cycle. – JimmyB May 20 '12 at 19:13