AVR104: Buffered Interrupt Controlled EEPROM Writes

Features

- **Flexible Multi-byte EEPROM Buffer**
- **Power Efficient EEPROM Access**
- **Access Control on Buffers**
- **EEPROM Buffer Rewrite**

Introduction

Many applications use the built-in EEPROM of the AVR to preserve and hence restore system information when power is removed from the system. The programming time for storing a single byte of data in the On-chip EEPROM between 3 and 8.5 ms and write access is therefore constrained by this write time. Typically, implementations writing to EEPROM utilize a polling method to determine when an EEPROM write is completed. This application note present a buffered interrupt driven approach, which significantly increases general performance and decreases power consumption compared to a polling implementation.

The improved performance and reduced power consumption are directly related to the decrease in execution overhead required when implementing an interrupt driven EEPROM routine versus a polling routine. When implementing polling EEPROM write accesses all processing resources (except interrupt driven ones) are occupied by the polling. An interrupt driven approach leaves the MCU core free to execute any code, while “waiting” for the EEPROM Interrupt to trigger the service routine once the previous write has completed. The interrupt driver EEPROM write therefore frees up to 8.5 ms of processing time per byte written compared to an implementation using polling – dependt on the programming time for the device used and the system clock frequency.

8-bit AVR® Microcontroller

Application Note

Rev. 2540A–AVR–04/03

Theory of Operation

The Atmel AVR devices has the capability of writing to the On-chip EEPROM through either an interrupt driven approach or by polling. Both methods have their advantages, but as far as overall processing performance and/or throughput is concerned the interrupt driven approach is the method of choice.

Polling Method

The polling subroutines, both read and write, poll the EEWE status bit to verify that the previous write-cycle has completed. If the write-cycle is still in progress, the MCU waits in a loop constantly polling the status bit waiting for clearance to proceed. It is also necessary to check whether or not a Self-Programming operation is active. If so the routine needs to wait until the SPM operation has been completed. If Self-Programming is not in use, this step may be omitted. Once clearance has been granted the next EEPROM operation may begin. The advantage to a polling implementation is the compact code footprint while the major disadvantage is the overhead, or the time the MCU spends waiting for an EEPROM write to be viable. A typical single byte write subroutine is listed below for reference.

```
EEPROM_WR:           ;EEPROM Write Sub-Routine
  sbic  EECR, EEWE   ;If EEWE Not Clear
  rjmp  EEPROM_WR    ;Wait Longer

SPM_BUSY:            ;(Omit if Self-Programming is Not Used)
  sbic  SPMCR, SPMEN ;If SPMEN Not Clear
  rjmp  SPM_BUSY     ;Wait Longer

  out   EEARH, r16   ;Output Address Byte (High)
  out   EEARL, r17   ;Output Address Byte (Low)
  out   EEDR, r18    ;Output Data Byte

  cli                ;Disable Global Interrupts

  sbi   EECR, EEMWE  ;Set Master Write Enable
  sbi   EECR, EEWE   ;Set EEPROM Write Strobe
                     ;This instruction takes four clock
                     ;cycles.

  sei                ;Enable Global Interrupts
  ret                ;Return From Sub-Routine
```

Interrupt Method

In the interrupt driven approach, there is no need to poll the EEWE status bit to verify whether the previous write cycle has completed. The EEPROM Ready Interrupt is constantly triggered when the EEWE status bit is cleared. It is however still necessary to poll the SPMEN status bit if Self-Programming is used, to make sure a Self-Programming operation is not currently active. The primary advantage of an interrupt driven approach is that dedicated hardware can request processing power when needed; this decreases the processor load.

Interrupt driven EEPROM access is made more efficient if a buffer is used: The buffer holds the value(s) that should be written to the EEPROM and the Interrupt Routine fetches data from the buffer.

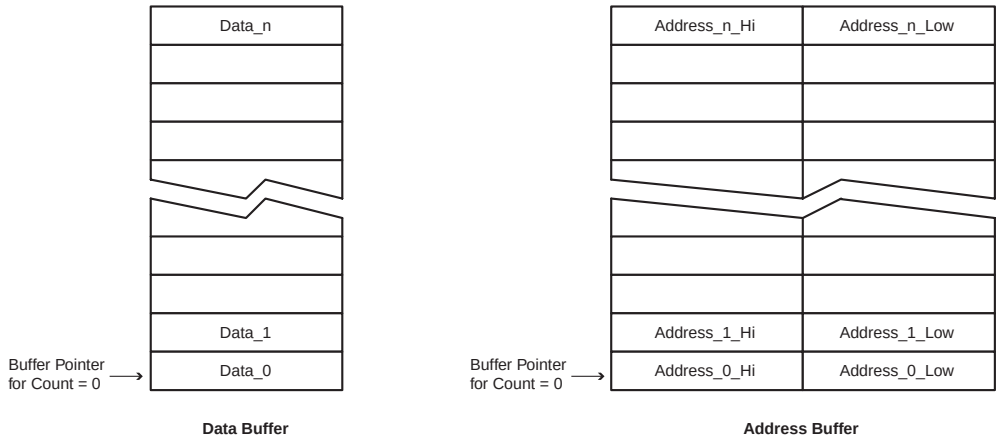
The complexity of the Interrupt Service Routine (ISR) is related to the number of bytes being written to the EEPROM. This routine is rather simple for a single byte buffer, but

when multiple bytes need to be written the routine becomes more complex. In order to accommodate multiple bytes a buffer will need to be constructed along with a counter variable. The counter will be responsible for keeping track of the number of locations currently in use. The counter therefore operates as a buffer index pointer.

Buffer Construction

In an effort to simplify and enhance the interrupt driven approach for multiple-byte writes, two buffers residing in the On-chip SRAM will be constructed. The two buffers are responsible for maintaining the address and data bytes prior to their placement in the On-chip EEPROM. The smallest EEPROM in the ATmega family is 512 bytes, thus requiring a 2-byte address, therefore the total size for a 1-byte EEPROM buffer is three-bytes of On-chip SRAM.

Figure 1. LIFO Buffers for Data and Address for EEPROM Acces



Buffer Size

One of the primary considerations of implementing Buffered EEPROM Writes is the size of the corresponding buffer. The buffer size affects performance because it is necessary to traverse the buffer to see if the desired EEPROM address is already located in the buffer; If so, it is necessary to update that location on a write or return the contents of that location on a read operation.

The EEPROM also plays a role in determining the optimum buffer size. Considerations must be made including the role of the EEPROM and the number of EEPROM bytes in use and/or updated. If a number of bytes are being written to the EEPROM in a short period of time, the buffer should be large enough to accommodate these requests without having to actually purge locations by writing their contents to EEPROM.

It is outside the scope of this applicaiton note to provide a specific method to determine the optimum buffer size. The buffer size needs to be assessed on an application-by-application basis. The guidelines presented above do provide enough insight to help determine in what range the optimum buffer size will be.

Buffering Consequences

When buffering the contents being written to the On-chip EEPROM, a couple of special conditions must be considered. Primarily, it needs to be determined what happens when a read or write of the EEPROM is needed but an updated value is contained in the buffer and has not been written to the EEPROM memory.

Separate read and write routines need to be created which first traverses the buffer to examine whether the desired location is contained in the buffer. If the desired location resides in the buffer, the data would be returned or updated if a read or write command was issued, respectively. If the location does not reside in the buffer, then a read instruction would access the EEPROM and return the requested data. However, if a write command were issued, the data would be placed in an unused buffer location and queued for writing to the EEPROM.

General Concerns Using EEPROM

The primary danger of a buffered interrupt driven approach to EEPROM writing resides in power failures. If a system is utilizing a buffered interrupt driven approach and a power loss occurs, the entire buffer will be lost completely since the SRAM, and thereby the EEPROM data and address buffers, is cleared by a power loss. Therefore, the consequences of a power loss should be analyzed carefully and considerations of how to avoid loss of non-volatile data must be made.

The AVR sleep modes will not affect the contents of the buffer, as the contents of the Register File and SRAM are unaltered when the device wakes up from sleep. However, the EE_RDY ISR must be given additional considerations when using the AVR sleep modes: The EE_RDY Interrupt Service Routine will wake-up the device when the MCU is in either the Idle or ADC Noise Reduction mode, but not from other sleep modes. To add possibility to use other sleep modes in the application in general, the sleep mode is modified if data is placed in the EEPROM buffer. The sleep mode is returned to its previous state when the buffer is emptied. This naturally implies that the sleep mode should not be modified while the EEPROM buffer contains data. This allows the rest of the application to call the SLEEP instruction without considering if the mode should be changed to meet the requirements of the EEPROM writes.

Precautions must be taken determining when to perform the actual EEPROM write operations. It is not possible to write to the EEPROM while Self-Programming is active! The EEPROM write should therefore be detained until Self-Programming has finished. Furthermore, to avoid EEPROM corruption it is recommended to avoid writes during periods of low V_{CC} (See device datasheet for further details).

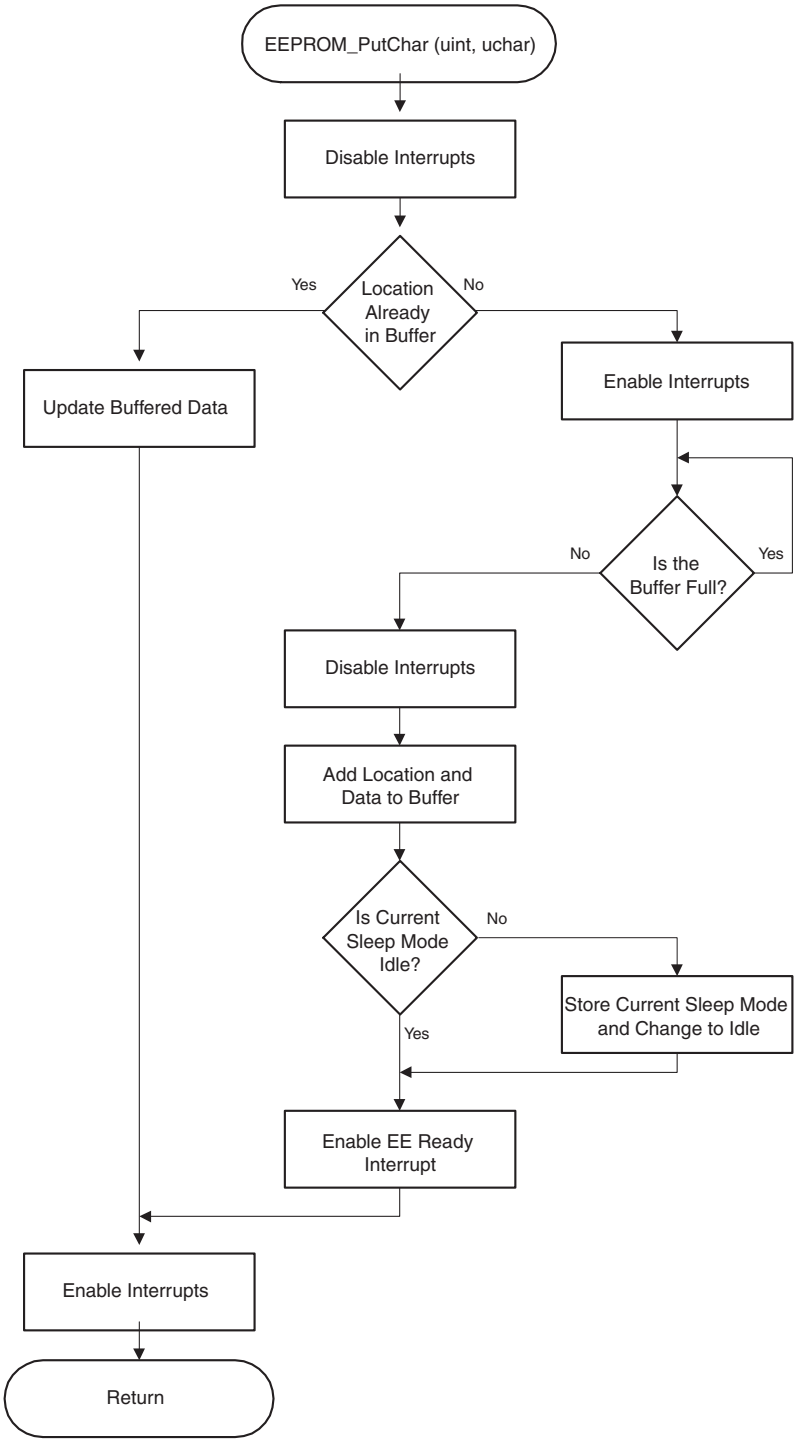
Implementation

The implementation of the Buffered Interrupt Controlled EEPROM writes example is targeted for the IAR Systems EWAVR v2.28A compiler. However, with only a few modifications, the source code would be re-targeted for ImageCraft, CodeVisionAVR, or any other C Compiler of choice.

The code example consists of two functions and an Interrupt Service Routine. The `EEPROM_PutChar()` places the contents desired to be written to the EEPROM in the buffer. The `EEPROM_GetChar()` function retrieves and returns the data at a desired EEPROM location. Finally, the `EE_RDY` Interrupt Service Routine is responsible for handling the EEPROM writes.

The following flowcharts outline the specific details of the `EEPROM_PutChar()` and `EEPROM_GetChar()` functions as well as the `EE_RDY` ISR.

Figure 2. EEPROM_PutChar(uint, uchar) Flowchart

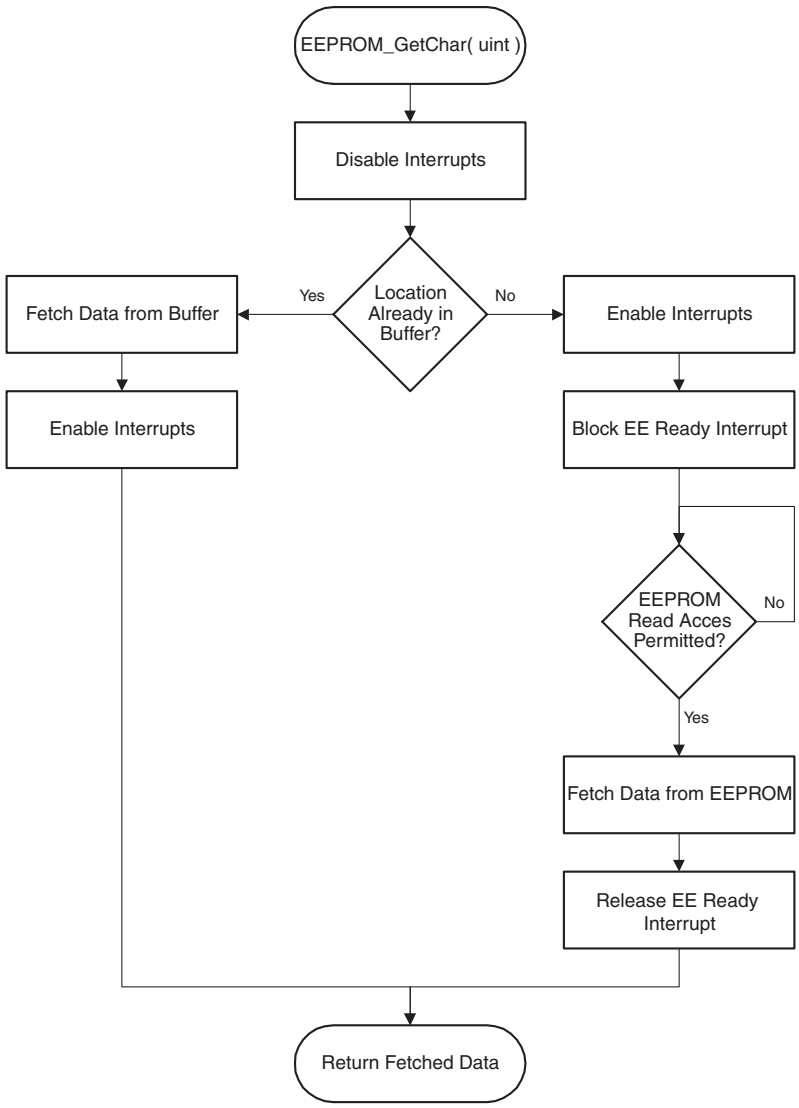


When the `EEPROM_PutChar()` routine is entered, it first traverses the buffer looking for the desired location. If the location already resides in the buffer, the location's data component is simply updated with the new value and returns to the main program. If the location is not contained in the buffer, both the target location and data component need to be placed in the buffer. In both cases the interrupts are disabled during the updating to ensure that the EEPROM ISR is not accessing the buffer while it is being modified.

Prior to placing a new location and data component in the buffer, the buffer needs to be checked for space. If space is available the contents may be added and the function returns to main program. However, if the buffer is full an interrupt driven EEPROM write is commenced. Once the write has completed and a buffer location is cleared, the desired content is added and the function returns to the main program.

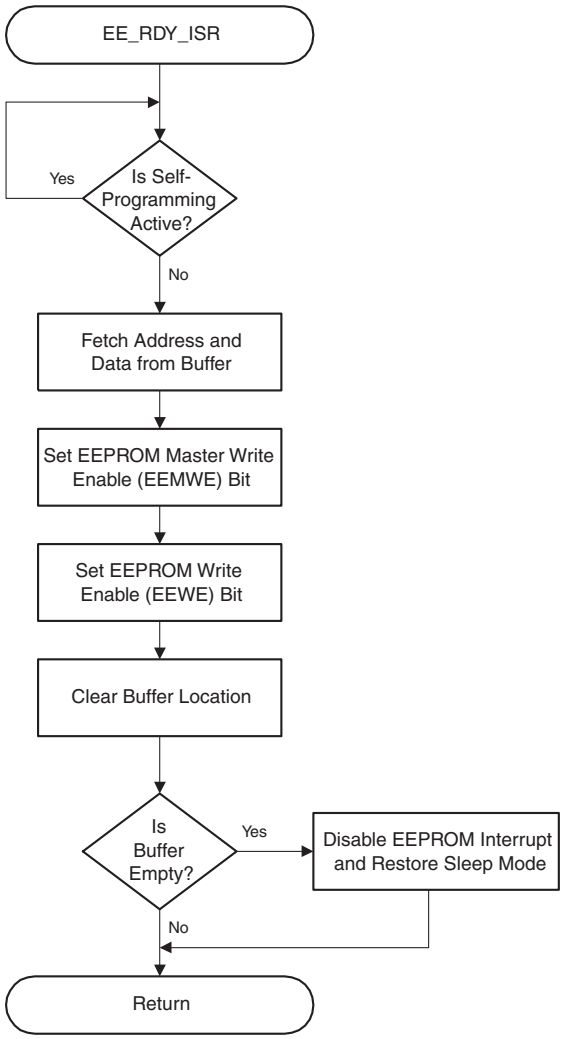
Further, the `EEPROM_PutChar()` function enables the EEPROM Ready Interrupt and configures Sleep to operate in Idle mode prior to returning to the main routine.

Figure 3. EEPROM_GetChar(uint) Flowchart



Once the `EEPROM_GetChar()` routine is entered, it first parses the buffer looking for the specified location. If the location is found in the buffer the data corresponding to that location is fetched and returned from the function. If the location isn't found in the buffer, the routine returns the fetched value from EEPROM for that specific address. The buffer is parsed first because the buffer contains updated EEPROM data that has not been committed to the EEPROM.

Figure 4. EE_RDY Interrupt Service Routine Flowchart



Prior to commencing any activity in the EE_RDY ISR, it is necessary to check if a Self-Programming operation is currently active. If so, an EEPROM write cannot commence until the Self-Programming operation has completed. (If your design does not make use of the Self-Programming feature this step may be omitted.)

After determining an EEPROM write can occur, the simple algorithm presented below (which is also found in the device datasheets) is followed to commence an EEPROM write.

1. Write the new EEPROM address to EEAR.
2. Write the new EEPROM data to EEDR.
3. Write a logical one to the EEMWE bit while writing a logical zero to the EEWE bit in EECR.
4. Within four clock cycles after setting EEMWE, write a logical one to EEWE.

When EEWE has been set, the MCU is halted for two cycles before the next instruction is executed. When the write access time has elapsed, the EEWE bit is cleared by hardware. Following the halt of the MCU, the ISR then empties the element of the EEPROM buffer just written by writing $FFFF to the address and $FF to the data locations prior to returning from the sub-routine.

If the buffer is emptied the EEPROM Ready Interrupt is disabled and the sleep mode is set back to the mode that was selected prior to the calling of the EEPROM_PutChar function when no data was present in the buffer.

Code Footprint

The footprint of the code for the Buffered Interrupt Controlled EEPROM write driver is specified in the table below.

Table 1. Memory Footprint of Code

Memory	No Optimization	Size Optimized	Speed Optimized
Code	440	388	418
Data	50(1)	50(1)	50(1)

Note: 1. Buffersize = 16

Potential Enhancements

The buffer used in this design is a Last In First Out (LIFO), commonly referred to as a Stack. In the Stack Buffer method it is possible for data to remain unwritten due to the LIFO method. The buffer could be modified creating a First In First Out (FIFO), otherwise known as a queue. Implementing a queue buffer would ensure the older data gets written first, however a significant increase in code overhead would be required.

Alternatively, by prioritizing the data, one could be assured that critical data gets programmed (which could aid in recovery of a power loss). A prioritization scheme could be implemented easily by utilizing either the upper bits of the Address word (as they are unused) or by using another type of bit-field containing the priority level. The interrupt sub-routine in charge of committing the data to EEPROM would then write all values with a high priority level first, following with values of decreasing priorities.

Atmel Corporation

2325 Orchard Parkway
San Jose, CA 95131
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

Regional Headquarters

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
Tel: (41) 26-426-5555
Fax: (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
Tel: (33) 2-40-18-18-18
Fax: (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
Tel: (33) 4-42-53-60-00
Fax: (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
Tel: (44) 1355-803-000
Fax: (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
Tel: (49) 71-31-67-0
Fax: (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
Tel: (33) 4-76-58-30-00
Fax: (33) 4-76-58-34-80

e-mail
literature@atmel.com

Web Site
http://www.atmel.com

2540A–AVR–04/03 0M