Emulating EEPROM Using AT89LP On-Chip Flash Data Memory

AT89LP EEPROM Emulation

Application Note

1. Introduction

Many embedded systems rely on nonvolatile parameters that are preserved across reset or power-loss events. In some systems this static information is used to initialize the system to a correct state at start-up. In other systems it is used to log system history or accumulated data. Traditionally these tasks have been implemented using EEPROM; first with off-chip EEPROM and later in on-chip EEPROM as levels of system integration have increased.

This application note describes how to emulate the behavior of an on-chip EEPROM using the on-chip Flash data memory of Atmel's AT89LP series of microcontrollers. Flash data memory is an alternative to EEPROM that is well suited for large parameter sets. Flash access routines are provided in both MCS®51 assembly and the C programming language. These routines are meant to replace existing code; they do not take full advantage of Flash memory performance and are not recommended for new designs.

2. Theory of Operation

Microcontrollers with nonvolatile memories have traditionally used Flash memory for program storage and EEPROM for data storage. Some members of the AT89LP family of microcontrollers include an on-chip Flash memory for nonvolatile data storage. Using this memory is not quite as simple as accessing standard internal RAM. This section details basic information about Flash memory operation and constraints.

2.1 Flash Memory Basics

Flash memory consists of independent cells each representing a single data bit. The flash cells are based on floating gate transistor technology: an electrical charge "trapped" on the floating gate determines the logic value of the cell. "Erasing" a cell charges the floating gate, allowing the cell to read as logic one. "Programming" a cell discharges the floating gate, bringing the logic value to zero. Therefore it is only possible to program (discharge) a cell that was previously erased (charged).

Bit cells are grouped into data bytes, but bits within the byte can be programmed individually. Since only the cells being programmed are discharged, the remaining unprogrammed cells remain charged. Any unprogrammed cell can be programmed at a later stage. Therefore programming a byte that is already programmed, without erasing it in between, will result in a bit-wise AND between the old value and the new value. If the byte is not erased in advance, it may not be possible to program it to the intended value. For example, assuming that a byte was FEh and was then programmed to 01h; the result would be 00h since the LSB cannot be changed from zero to one by a program operation.

3707A–MICRO–10/09

Flash memory is arranged in pages of multiple bytes. An erase operation acts on an entire page; that is, all the bits of all the bytes in the page are charged at one time. A program operation *can* be performed on the entire page; that is, one or more bytes, up to the maximum page size, can have some or all of their bits discharged at one time. If a single bit in the page must change from zero to one, the entire page must be erased and all bytes reprogrammed.

Traditional EEPROM memory is similar to Flash memory except that the “Erase” and “Program” operations are merged into a single atomic “Write” operation that acts on a single byte. The “Write” operation first erases (charges) all bits in a byte and then programs (discharges) those bits that must be zero. Therefore an EEPROM can update a single byte without regard to its previous value or the value of its neighbors. However, most EEPROMs cannot update multiple bytes simultaneously.

A comparison of EEPROM and Flash memories is summarized in Table 2-1.

Table 2-1. Comparison of EEPROM and Flash Memories

Feature	EEPROM	FLASH
Minimum Erase Size	single byte	one page (multiple bytes)
Minimum Write Size	single byte	single bit
Maximum Write Size	single byte	one page (multiple bytes)
Read Size	byte	byte
Read Speed	fast	fast
Write Speed	very slow (milliseconds)	slow (10s of microseconds)

2.2 Data Constraints

The byte-wide Flash data memory easily supports 8-bit scalar values such as the “char” type. Wider data types can also be supported with some considerations detailed below. Data arrays are beyond the scope of this document. Please see the application note “AT89LP Flash Data Memory” for more detailed information on generic data storage in Flash memory.

One advantage Flash data memory has over traditional EEPROM is the ability to program multiple bytes at one time. This makes support for larger data types much more efficient than in EEPROM. However, to achieve the highest efficiency some constraints on the data size and location must be observed. Within the scope of this document we will assume that all data obeys the following constraints:

1. All data types are scalar values or structures of scalars (no arrays).
2. The size of the largest data type is equal to or less than the page size (AT89LP828) or half page size (AT89LP6440).
3. Data is aligned in memory such that an entire data value resides within a single page (AT89LP828) or half page (AT89LP6440), i.e. multiple byte data types must not straddle page/half-page boundaries.

By following these constraints we ensure that each individual data unit can be written with a single programming operation. Larger or unaligned data types are also supported by Flash data memory but their implementation is left as an exercise for the reader.

3. Architectural Overview

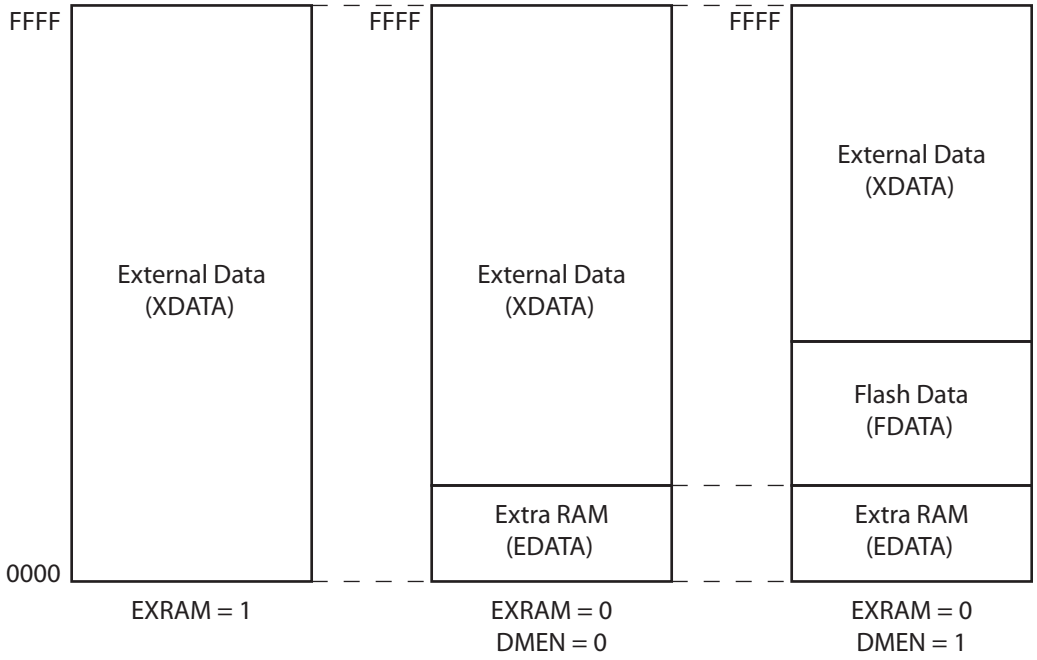
The following section provides a general overview of the architectural details of the Flash data memory on AT89LP microcontrollers. For more information see a specific device's datasheet.

3.1 Memory Organization

The on-chip Flash data memory is mapped into the 16-bit external memory address space (XDATA) of an AT89LP microcontroller as shown in Figure 3-1. The Flash data memory is referred to as the FDATA memory space and is accessed with the MOVX @DPTR instructions. Note that MOVX @Ri instructions will not access FDATA. MOVX instructions to FDATA require a minimum of 4 clock cycles.

By default FDATA is not mapped on to the XDATA space. The EXRAM bit in the AUXR SFR forces all XDATA address to access external memory and must be cleared before accessing FDATA. To enable FDATA, the Data Memory Enable bit (DMEN) in the MEMCON SFR must be set to "1". When DMEN = 0, FDATA is not accessible. When DMEN = 1, FDATA will be mapped at the bottom of XDATA, above any internal Extra RAM (EDATA). The IAP bit in MEMCON enables the self programming feature for the CODE memory and must also be cleared before accessing FDATA.

Figure 3-1. External Data Memory Map



FDATA is organized by pages. For example, FDATA on AT89LP828 has 16 pages of 64 bytes each, mapped from 0200h–03FFh, while AT89LP6440 has 64 pages of 128 bytes, mapped from 1000h–2FFFh. To facilitate page programming, AT89LP devices include a temporary page buffer to store data to be written to a page. The size of the page buffer determines the maximum number of bytes that may be programmed at one time. AT89LP828 has a full-page buffer of 64 bytes as shown in Figure 3-2. AT89LP6440, on the other hand, has only a half-page buffer of 64 bytes. Therefore, two write cycles are required to fill the entire 128-byte page, one for the low half page (00H–3FH) and one for the high half page (40H–7FH) as shown in Figure 3-3.

Figure 3-2. Page Programming Structure (EX: AT89LP828)

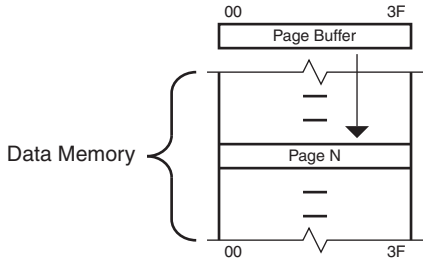
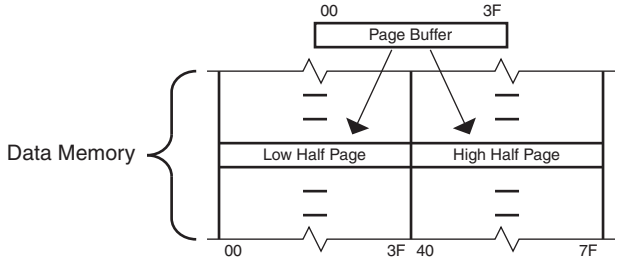


Figure 3-3. Half Page Programming Structure (EX: AT89LP6440)



The page buffer is reset to the all 0xFF state after any programming operation. Therefore any unloaded locations will not be programmed. The buffer obeys the same rules as the memory cells in that only zeros may be loaded. Loading the same location with different data will result in a bitwise AND between the old and new values. Loading 0xFF to any buffer location leaves the buffer unchanged. The provided routines make use of this behavior to optimize the buffer insertion routines. Note that due to architectural differences, bitwise ANDing is not allowed on the AT89LP6440; however, 0xFF can still be loaded to any location.

3.2 Access Protocol

The FDATA address space accesses an internal nonvolatile data memory. This address space can be read just like XDATA by issuing a MOVX A,@DPTR; however, writes to FDATA require a more complex protocol and take several milliseconds to complete. The AT89LP828 and AT89LP6440 use an *idle-while-write* architecture where the CPU is placed in an idle state while the write occurs. When the write completes, the CPU will continue executing with the instruction after the MOVX @DPTR,A instruction that started the write. All peripherals will continue to function during the write cycle; however, interrupts will not be serviced until the write completes.

3.2.1 Read Operation

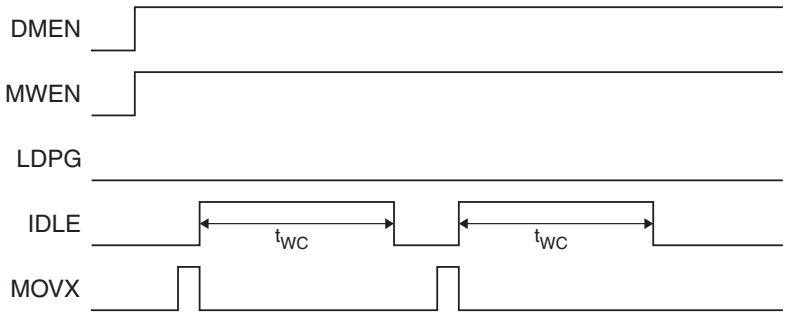
To enable read access to the Flash data memory, the DMEN bit (MEMCON.3) must be set to one, IAP (MEMCON.7) must be cleared to zero, and EXRAM (AUXR.1) must be cleared to zero. IAP and EXRAM are zero by default after reset. Then any MOVX A,@DPTR instruction targeting the FDATA address range will return a byte from the data memory.

```
; Flash Data Read Example
MOV  MEMCON, #08h          ; DMEN=1, IAP=0
MOV  DPTR, #SOME_ADDR      ; load pointer to FDATA
MOVX A, @DPTR              ; fetch byte
```

3.2.2 Write Operation

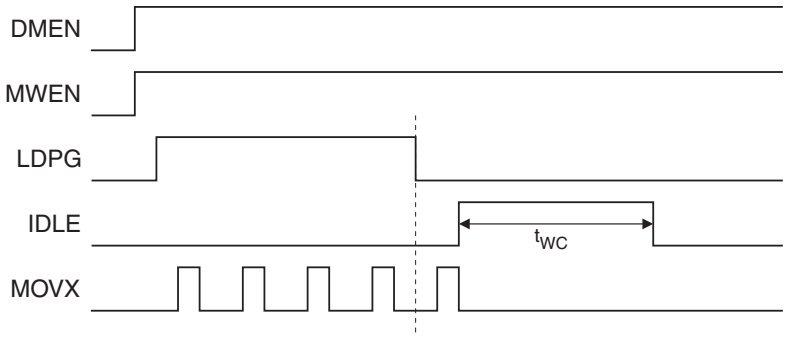
To enable write access to the nonvolatile data memory, the MWEN bit (MEMCON.4) must also be set to one. When MWEN = 1, DMEN = 1, IAP = 0 and EXRAM = 0, MOVX @DPTR,A may be used to write to FDATA. The LDPG bit (MEMCON.5) allows multiple data bytes to be loaded to the temporary page buffer. While LDPG = 1, MOVX @DPTR,A instructions will load data to the page buffer, but will not start a write sequence. Note that a previously loaded byte must not be reloaded prior to the write sequence as a bit-wise AND will occur between the data values. To write the buffer into the memory, LDPG must first be cleared and then a MOVX @DPTR,A with the final data byte is issued. The address of the final MOVX determines which page will be written. If a MOVX @DPTR,A instruction is issued while LDPG = 0 without loading any previous bytes, only a single byte will be written. The page buffer is reset after each write operation. Figures 3-4 and Figure 3-5 show the difference between byte writes and page writes (not to scale).

Figure 3-4. FDATA Byte Write




```
; Flash Data Byte Write Example (Write two bytes)
MOV  MEMCON, #18h          ; DMEN=1,MWEN=1,IAP=0,LDPG=0
MOV  DPTR, #SOME_ADDR      ; load pointer to FDATA
MOV  A, #SOME_DATA         ; load data to be written
MOVX @DPTR, A              ; write byte
MOV  DPTR, #OTHER_ADDR     ; load another pointer to FDATA
MOV  A, #OTHER_DATA        ; load data to be written
MOVX @DPTR, A              ; write byte
```

Figure 3-5. FDATA Page Write




```
; Flash Data Page Write Example (Write five bytes)
MOV  MEMCON, #38h          ; DMEN=1,MWEN=1,IAP=0,LDPG=1
MOV  DPTR, #SOME_ADDR      ; load pointer to FDATA
MOV  A, #DATA1             ; load data to be written
MOVX @DPTR, A              ; load byte
MOV  A, #DATA2             ; load data to be written
INC  DPTR                  ; next location
```

```
MOVX @DPTR, A              ; load byte
MOV  A, #DATA3             ; load data to be written
INC  DPTR                  ; next location
MOVX @DPTR, A              ; load byte
MOV  A, #DATA4             ; load data to be written
INC  DPTR                  ; next location
MOVX @DPTR, A              ; load byte
ANL  MEMCON, #0DFh         ; LDPG=0
MOV  A, #DATA5             ; load data to be written
INC  DPTR                  ; next location
MOVX @DPTR, A              ; load byte and write
```

3.2.3 Erase Operation

The auto-erase bit AERS (MEMCON.6) can be set to one to perform a page erase automatically at the beginning of any write sequence. The page erase will erase the entire page. On AT89LP6440 this means both the low and high half pages are erased. Since the write operation paired with the auto-erase can only program one of the half pages, a second write cycle without auto-erase is required to update the other half page.

A page erase operation, without writing any data, can be performed by setting AERS and writing a dummy byte of FFh to any byte in the page of interest. Remember than write operations only program zeroes, ones leave the data untouched.

```
; Flash Page Erase Example
MOV  MEMCON, #58h          ; DMEN=1,MWEN=1,IAP=0,LDPG=0,AERS=1
MOV  DPTR, #SOME_ADDR      ; load pointer to FDATA
MOV  A, #0FFh              ; load dummy data to be written
MOVX @DPTR, A              ; start erase
```

Table 3-1. MEMCON – Memory Control Register

MEMCON = 96H — Reset Value = 0000 00XXB

Not Bit Addressable

	IAP	AERS	LDPG	MWEN	DMEN	ERR	–	$\overline{\text{WRTINH}}$
Bit	7	6	5	4	3	2	1	0

Symbol	Function
IAP	In-Application Programming Enable. When IAP = 1 and the IAP Fuse is enabled, programming of the CODE/SIG space is enabled and MOVX @DPTR instructions will access CODE/SIG instead of EDATA or FDATA. Clear IAP to disable programming of CODE/SIG and allow access to EDATA and FDATA.
AERS	Auto-Erase Enable. Set to perform an auto-erase of a Flash memory page (CODE, SIG or FDATA) during the next write sequence. Clear to perform write without erase.
LDPG	Load Page Enable. Set to this bit to load multiple bytes to the temporary page buffer. Byte locations may not be loaded more than once before a write. LDPG must be cleared before writing.
MWEN	Memory Write Enable. Set to enable programming of a nonvolatile memory location (CODE, SIG or FDATA). Clear to disable programming of all nonvolatile memories.
DMEN	Data Memory Enable. Set to enable nonvolatile data memory and map it into the FDATA space. Clear to disable nonvolatile data memory.
ERR	Error Flag. Set by hardware if an error occurred during the last programming sequence due to a brownout condition (low voltage on VCC). Must be cleared by software.
$\overline{\text{WRTINH}}$	Write Inhibit Flag. Cleared by hardware when the voltage on VCC has fallen below the minimum programming voltage. Set by hardware when the voltage on VCC is above the minimum programming voltage.

4. Implementation

This application note provides four routines in both assembly and C:

- **write_eeprom_byte** – Update one data byte on a full-page device (AT89LP828).
- **write_eeprom_byte2** – Update one data byte on a half-page device (AT89LP6440).
- **write_eeprom_word** – Update one data word (2 bytes) on a full-page device
- **write_eeprom_word2** – Update one data word (2 bytes) on a half-page device .

4.1 Firmware Description

The provided routines emulate the behavior of a traditional EEPROM, that is they modify a memory location without affecting the other values in the memory. To create this behavior in a Flash memory, the entire page containing the location of interest must be saved to a buffer, the page is erased, and then the saved values with the new value inserted are written back to the page. To increase the Flash endurance and save execution time, data checks are performed so that unnecessary write or erase operations are skipped.

Each routine follows the same basic flow (See Figure 4-1). First the old and new value are compared. If they are equal then no update is required. If they are not equal then a bit-wise AND is performed to determine if an erase is required. If an erase is not required (ANDing old and new is equal to new) the new value is programmed (this optimization does not apply to AT89LP6440). Otherwise the temporary page buffer is loaded with the old contents of the page, inserting the new value at its proper location. An autoerase-and-program sequence is initiated on the correct page by writing a dummy 0xFF byte to the correct address. Afterwards the new value is verified and the status is returned. The saving and restoring of registers/SFRs at the start/end of the routines are left as exercises for the reader.

The main difference between the full-page and half-page versions (See Figure 4-2) is the handling of the page reload for the autoerase sequence. While the full-page version requires only the temporary buffer and a single write for this purpose, the half-page version must complete two write sequences (one for each half page) and requires a buffer allocated in one of the volatile memories to store one half-page while programming the other. This buffer can be located in the IDATA, EDATA or XDATA spaces.

Routines for 16-bit word values are provided as examples of larger data types. These routines are easily extensible to other data types. As the data type of every location in a page may be unknown, the page is cast to an 8-bit data type for the reload process.

4.2 Requirements

The following requirements must be considered for this application:

- The memory page size is limited to 256 bytes or less (Assembly routines only).
- If an interrupt service routine that also writes to FDATA can interrupt the provided routines, then that interrupt, or global interrupts, must be disabled prior to executing the update to prevent collisions in the temporary page buffer.
- If an interrupt service routine that uses MOVX @DPTR,A to access other memories can interrupt the provided routine it must save and restore the MEMCON and AUXR registers to preserve the state of EXRAM, DMEN, MWEN, LDPG, AERS and IAP.
- The erase-write endurance of an entire memory page is limited by the total number of updates occurring per page, not necessarily its most frequently changing member. If

endurance is a concern, see the application note “AT89LP Flash Data Memory” for methods of managing endurance.

Figure 4-1. Write EEPROM Byte Flowchart (AT89LP828)

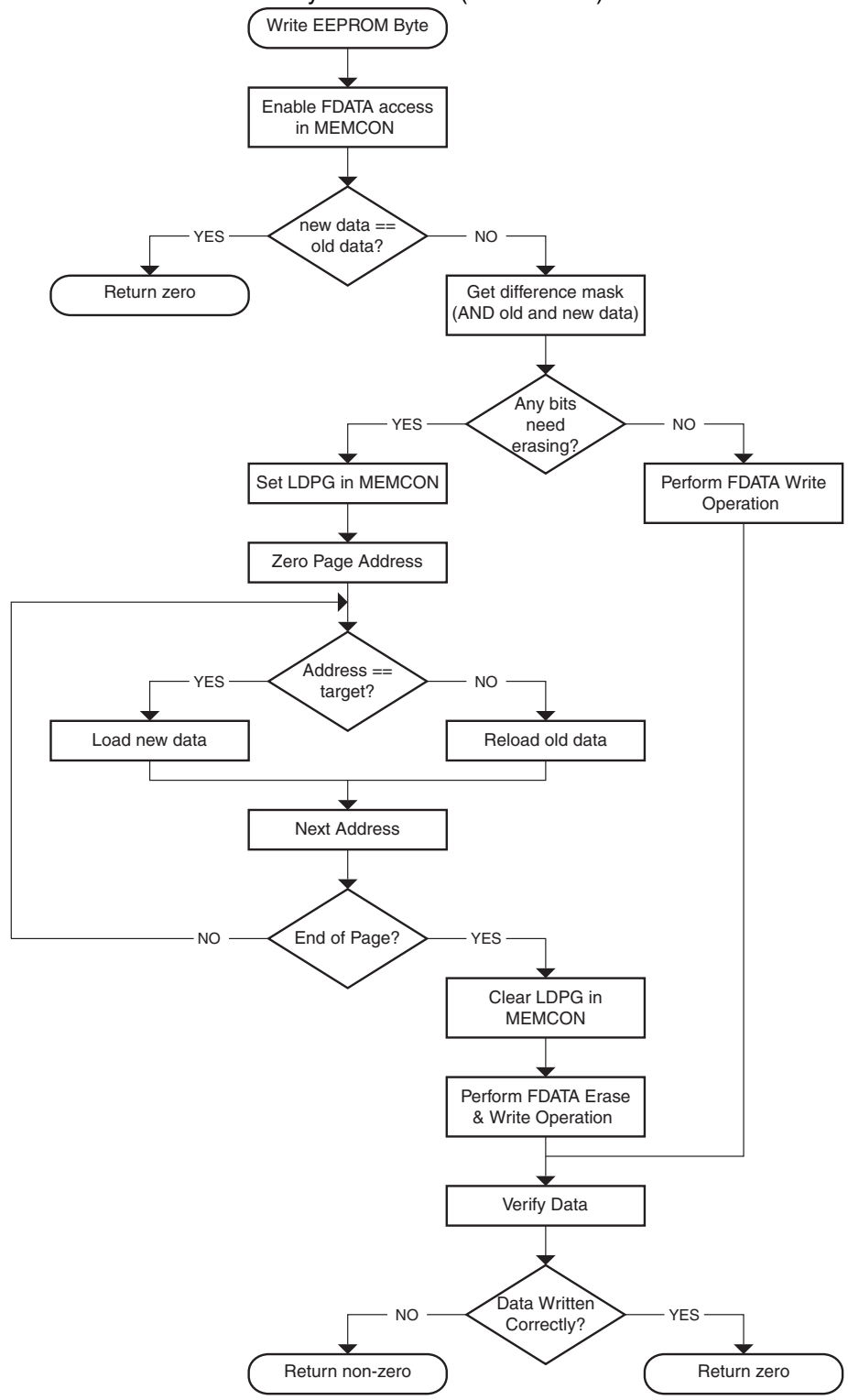
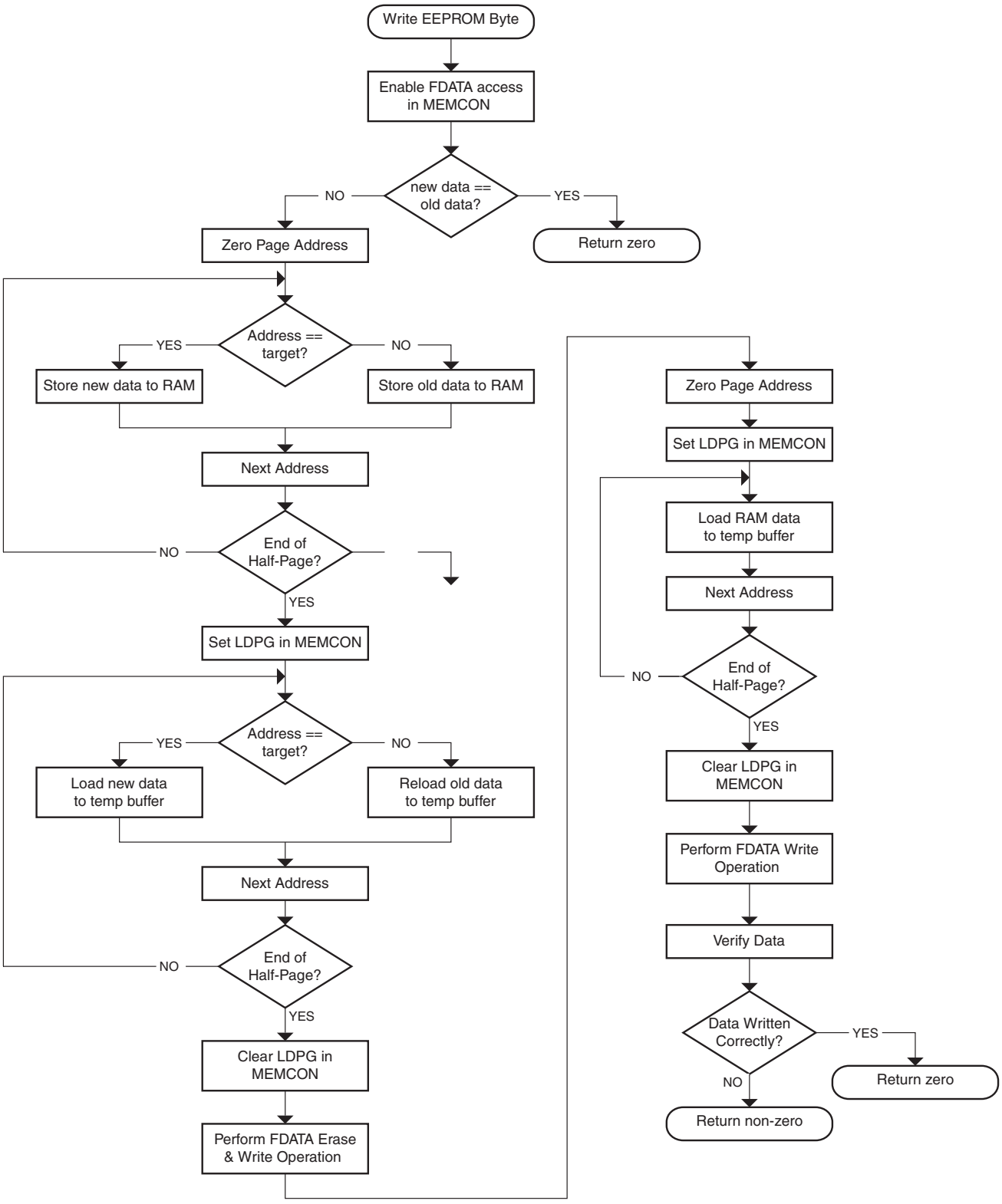


Figure 4-2. Write EEPROM Byte 2 Flowchart (AT89LP6440)



5. Appendix A – Assembly Routines

Note that most third party assemblers currently do not support the special MOVX A, @/DPTR and MOVX @/DPTR, A instructions. These instructions may be emulated with the following substitutions:

```
;; for MOVX A, @/DPTR use:
DB   0A5h
MOVX A, DPTR

;; for MOVX @/DPTR, A use:
DB   0A5h
MOVX @DPTR, A
```

5.1 write_eeprom_byte()

```
;---------------------------------------------------------------------------
;- EEPROM Byte Write Emulation (Full Page Version)
;-
;- Arguments:
;-     ACC - data to be written (modified)
;-   DPTR0 - address to write in FDATA (not modified)
;- Return value:
;-     ACC - 0=success, !0=failure
;- Register usage:
;-    B - temp variable
;-   R7 - loop counter
;-   R6 - save DPL0
;-   R5 - save DPH0
;- Modifies: PSW, MEMCON
;---------------------------------------------------------------------------


write_eeprom_byte:
   MOV  MEMCON, #18h       ; set DMEN, MWEN ; clear IAP, LDPG
   MOVB, A
   MOVX A, @DPTR           ; fetch old value
   CJNE A, B, bitwise      ; if (old != new) bitwise
   CLR  A                  ; else return(SUCCESS)
   RET
bitwise:
   ANL  A, B               ; try AND function between old and new
   CJNE A, B, erase_byte   ; 0->1 needs erase
   SJMP write_byte         ; else just write
erase_byte:
   ORL  MEMCON, #60h       ; set LDPG, AERS
   MOV  R5, DPH0           ; save high address
   MOV  R6, DPL0           ; save low address
   ANL  DPL0, #(0FFh-(PAGESIZE-1)); zero back page address
   MOV  R7, #PAGESIZE
load_byte:
   MOV  A, R6
   CJNE A, DPL0, reload_byte  ; is this the byte of interest?
   MOV  A, B
   SJMP next_byte
reload_byte:
   MOVX A, @DPTR
```

```
next_byte:
    MOVX @DPTR, A
    INC  DPTR
    DJNZ R7, load_byte
    ANL  MEMCON, #0DFh          ; clear LDPG
    MOV  DPH0, R5
    MOV  DPL0, R6
    MOV  A, #0FFh
write_byte:
    MOVX @DPTR, A
verify_byte:
    MOVX A, @DPTR
    CLR  C
    SUBB A, B
    RET
```

5.2 write_eeprom_word()

```
;-------------------------------------------------------------------------
;- EEPROM Word Write Emulation (Full Page Version)
;-
;- Arguments:
;-   R2,R3 - data to be written (not modified)
;-   DPTR0 - address to write in FDATA (not modified)
;- Return value:
;-     ACC - 0=success, !0=failure
;- Register usage:
;-    B - temp variable
;-   R7 - loop counter
;-   R6 - save DPL0
;-   R5 - save DPH0
;-   R4 - temp variable
;- Modifies: PSW, MEMCON
;-------------------------------------------------------------------------
write_eeprom_word:
    MOV  MEMCON, #38h           ; set DMEN, MWEN, LDPG ; clear IAP
    MOV  R5, DPH0               ; save high address
    MOV  R6, DPL0               ; save low address
    MOVX A, @DPTR               ; fetch old low value
    MOV  R4, A
    MOV  B, R2
    CJNE A, B, bitwise1         ; if (old != new) bitwise
    INC  DPTR
    MOVX A, @DPTR               ; fetch old high value
    MOV  B, R3
    CJNE A, B, bitwise2         ; if (old != new) bitwise
    CLR  A                      ; else return(SUCCESS)
    RET
bitwise1:
    ANL  A, R2                  ; try AND function between old and new high
    CJNE A, B, erase_word       ; 0->1 needs erase
    INC  DPTR
    MOVX A, @DPTR               ; fetch old high value
    MOV  B, R3
bitwise2:
```

```
    ANL  A, R3
    CJNE A, B, erase_word
    MOV  DPL0, R6
    MOV  A, R2
    MOVX @DPTR, A
    INC  DPTR
    MOV  A, R3
    MOVX @DPTR, A
    SJMP write_word            ; else just write
erase_word:
    ORL  MEMCON, #60h          ; set LDPG, AERS
    ANL  DPL0, #(0FFh-(PAGESIZE-1)); zero back page address
    MOV  R7, #PAGESIZE
load_word:
    MOV  A, R6
    CJNE A, DPL0, reload_word  ; is this the byte of interest?
    MOV  A, R2
    MOVX @DPTR, A
    INC  DPTR
    DEC  R7
    MOV  A, R3
    SJMP next_word
reload_word:
    MOVX A, @DPTR
next_word:
    MOVX @DPTR, A
    INC  DPTR
    DJNZ R7, load_word
write_word:
    ANL  MEMCON, #0DFh         ;  clear LDPG
    MOV  DPH0, R5
    MOV  DPL0, R6
    MOV  A, #0FFh
    MOVX @DPTR, A
verify_lo:
    MOVX A, @DPTR
    CLR  C
    SUBB A, R2
    JZ   verify_hi
    RET
verify_hi:
    INC  DPTR
    MOVX A, @DPTR
    CLR  C
    SUBB A, R3
    MOV  DPL0, R6
    RET
```

5.3 write_eeprom_byte2()

```
;------------------------------------------------------------------------
;- EEPROM Byte Write Emulation Routine (Half Page Version)
;-
;- Arguments:
;-     ACC - Data to write (modified)
;-   DPTR0 - Address to write (pointer not modified)
;-   DPTR1 - Address of temp buffer (pointer not modified)
;- Return value:
;-     ACC - 0=success, !0=failure
;- Register usage:
;-    B - temp variable
;-   R7 - loop counter
;-   R6 - save DPL0
;-   R5 - save DPH0
;-   R4 - save DPL1
;-   R3 - save DPH1
;-   R2 - DPL0 for high half page
;- Modifies: PSW, DPCF, MEMCON
;------------------------------------------------------------------------


write_eeprom_byte2:
    MOV  MEMCON, #18h          ; set DMEN, MWEN ; clear IAP,LDPG
    MOV  B, A                  ; save new value
    MOVX A, @DPTR              ; fetch old value
    CJNE A, B, erased_byte     ; if (old != new) then erase
    CLR  A                     ; else return(SUCCESS)
    RET
erased_byte:
    CJNE A, #0FFh, erase_byte  ; already erased?
    SJMP write_byte            ; then just write
erase_byte:
    MOV  DPCF, #0C0h           ; config post increment dptrs
    MOV  R3, DPH1              ; save high address
    MOV  R4, DPL1              ; save low address
    MOV  R5, DPH0              ; save high address
    MOV  R6, DPL0              ; save low address
    ANL  DPL0, #PAGEMASK       ; zero back page address
    MOV  R7, #(PAGESIZE/2)     ; load half page loop counter
save_byte:
    MOV  A, R6                 ; get original pointer low byte
    CJNE A, DPL0, save_new_byte ; is this the byte of interest?
    MOV  A, B                  ; get new value
    INC  DPTR                  ; update pointer
    SJMP save_next_byte
save_new_byte:
    MOVX A, @DPTR              ; retrieve existing value
save_next_byte:
    MOVX @/DPTR, A             ; store value to buffer
    DJNZ R7, save_byte         ; next byte
    ORL  MEMCON, #60h          ; set LDPG, AERS
    MOV  DPCF, #0              ; disable post increment
    MOV  R1, DPH0              ; save start of upper half page
    MOV  R2, DPL0
    MOV  R7, #(PAGESIZE/2)     ; load half page loop counter
```

```
load_byte:
    MOV  A, R6                 ; get original pointer low byte
    CJNE A, DPL0, reload_byte  ; is this the byte of interest?
    MOV  A, B                  ; get new value
    SJMP load_next_byte
reload_byte:
    MOVX A, @DPTR              ; retrieve existing value
load_next_byte:
    MOVX @DPTR, A              ; store value to temp buffer
    INC  DPTR                  ; update pointer
    DJNZ R7, load_byte         ; next byte
    ANL  MEMCON, #0DFh         ;  clear LDPG
    MOV  DPH0, R5              ; restore upper half pointer
    MOV  DPL0, R2
    MOV  A, #0FFh              ; dummy data
    MOVX @DPTR, A              ; initiate write to upper half page
    ORL  MEMCON, #20h          ; set LDPG
    MOV  R7, #(PAGESIZE/2)
    MOV  DPH1,R3               ; restore high address
    MOV  DPL1,R4               ; restore low address
    MOV  DPCF, #0C0h           ; post increment
restore_byte:
    MOVX A, @/DPTR
    MOVX @DPTR, A
    DJNZ R7, restore_byte
    MOV  DPH1,R3               ; restore high address
    MOVD PL1,R4                ; restore low address
    MOV  DPCF, #0
    ANL  MEMCON, #09Fh         ;  clear LDPG, AERS
    MOV  DPH0, R5
    MOV  DPL0, R6
    MOV  A, #0FFh
write_byte:
    MOVX @DPTR, A
verify_byte:
    MOVX A, @DPTR
    CLR  C
    SUBB A, B
    RET
```

5.4 write_eeprom_word2()

```
;------------------------------------------------------------------------
;- EEPROM Word Write Emulation Routine (Half Page Version)
;-
;- Arguments:
;-   R0,R1 - Data to be written (not modified)
;-   DPTR0 - Address to write
;-   DPTR1 - Address of 64-byte buffer
;- Return value:
;-     ACC - 0=success, !0=failure
;- Register usage:
;-    B - temp variable
;-   R7 - loop counter
;-   R6 - save DPL0
;-   R5 - save DPH0
```

```
;-   R4 - save DPL1
;-   R3 - save DPH1
;-   R2 - DPL0 for high half page
;- Modifies: PSW, DPCF, MEMCON
;-------------------------------------------------------------------------------

write_eeprom_word2:
    MOV  MEMCON, #38h          ; set DMEN, MWEN, LDPG ; clear IAP
    MOV  R5, DPH0
    MOV  R6, DPL0
    MOV  B, R0
    MOVX A, @DPTR              ; fetch old value
    CJNE A, B, erase_word      ; if (old != new) then erase
    MOV  B, R1
    INC  DPTR
    MOVX A, @DPTR              ; fetch old value
    CJNE A, B, erase_word      ; if (old != new) then erase
    CLR  A                     ; else return(SUCCESS)
    RET
erase_word:
    MOV  DPCF, #0C0h           ; post increment
    MOV  R3, DPH1              ; save high address
    MOV  R4, DPL1              ; save low address
    ANL  DPL0, #PAGEMASK       ; zero back page address
    MOV  R7, #(PAGESIZE/2)
save_word:
    MOV  A, R6
    CJNE A, DPL0, save_new_word ; is this the word of interest?
    MOV  A, R0
    MOVX @/DPTR, A
    INC  DPTR
    MOV  A, R1
    INC  DPTR
    SJMP save_next_word
save_new_word:
    MOVX A, @DPTR
save_next_word:
    MOVX @/DPTR, A
    DJNZ R7, save_word
    ORL  MEMCON, #60h          ; set LDPG, AERS
    MOV  DPCF, #0
    MOV  R1, DPH0
    MOV  R2, DPL0
    MOV  R7, #(PAGESIZE/2)
load_word:
    MOV  A, R6
    CJNE A, DPL0, reload_word  ; is this the word of interest?
    MOV  A, R0
    MOVX @DPTR, A
    INC  DPTR
    MOV  A, R1
    SJMP load_next_word
reload_word:
    MOVX A, @DPTR
```

```
load_next_word:
    MOVX @DPTR, A
    INC  DPTR
    DJNZ R7, load_word
    ANL  MEMCON, #0DFh          ;  clear LDPG
    MOV  DPH0, R5
    MOV  DPL0, R2
    MOV  A, #0FFh
    MOVX @DPTR, A
    ORL  MEMCON, #20h           ; set LDPG
    MOV  R7, #(PAGESIZE/2)
    MOV  DPH1,R3                ; save high address
    MOV  DPL1,R4                ; save low address
    MOV  DPCF, #0C0h            ; post increment
restore_word:
    MOVX A, @/DPTR
    MOVX @DPTR, A
    DJNZ R7, restore_word
    MOV  DPCF, #0
write_word:
    ANL  MEMCON, #09Fh          ;  clear LDPG, AERS
    MOV  DPH0, R5
    MOV  DPL0, R6
    MOV  A, #0FFh
    MOVX @DPTR, A
verify_word:
    MOVX A, @DPTR
    CLR  C
    SUBB A, B
    JNZ  done
    INC  DPTR
    MOVX A, @DPTR
    CLR  C
    SUBB A, B
    MOV  DPL0, R6
done:
    RET
```

6. Appendix B – C Routines

6.1 write_eeprom_byte()

```
/*---------------------------------------------------------------------------
 * EEPROM Byte Write Emulation (Full Page Version)
 *
 * ptr  - address of byte in FDATA to write
 * byte - value to write
 *---------------------------------------------------------------------------
 */
char write_eeprom_byte(unsigned char xdata *ptr, unsigned char byte) {
  unsigned char xdata *tmp;
  unsigned char i, tmp1;
  unsigned int adr;

  tmp1 = *ptr;         // fetch old data value
  adr = (int)ptr;      // cast pointer to int for arithmetic

  MEMCON = 0x18;       // DMEN=1 MWEN=1 LDPG=0
  if (byte == tmp1) {
    /* no update required */
    return(0);
  } else if (byte == (tmp1 & byte)) {
    /* erase not required, just write */
    *ptr = byte;
  } else {
    /* reload & erase page */
    MEMCON |= 0x60;    // LDPG=1 AERS=1
    tmp = (unsigned char xdata *)(adr & ~(PAGESIZE-1));// zero page
    for (i=PAGESIZE; i>0; i--) {
      if( tmp == ptr) { // is this the byte of interest?
        *tmp = byte;     // load new data
      } else {
        *tmp = *tmp;     // reload old data
      }
      tmp++;             // next byte
    }
    MEMCON &= 0xDF;    // LDPG=0
    *ptr = 0xff;       // initiate page write
  }
  /* verify value */
  tmp1 = *ptr;
  return(tmp1 != byte);
}
```

6.2 write_eeprom_word()

```
/*-------------------------------------------------------------------------
 * EEPROM Word Write Emulation (Full Page Version)
 *
 * ptr  - address of word in FDATA to write
 * word - value to write
 *-------------------------------------------------------------------------
 */
void write_eeprom_word(unsigned int xdata *ptr, unsigned int word) {
  unsigned char xdata *tmp;
  unsigned int xdata *wp;
  unsigned int i, tmp1;
  unsigned int adr;

  tmp1 = *ptr;         // fetch old value
  adr = (int)ptr;      // cast pointer to int for arithmetic

  MEMCON = 0x38;       // DMEN=1 MWEN=1 LDPG=0
  if (word == tmp1) {
    /* no update required */
    return(0);
  } else if (word == (tmp1 & word)) {
    /* erase not required, just write */
    *ptr = word;
  } else {
    /* reload & erase page */
    MEMCON |= 0x60;    // LDPG=1 AERS=1
    tmp = (unsigned char xdata *)(adr & ~(PAGESIZE-1));// zero page
    for (i=PAGESIZE; i>0; i--) {
      if( tmp == (unsigned char xdata *)ptr) { // is this the word of interest?
        wp = (unsigned int xdata *)tmp;
        *wp = word;    // load new data
        tmp = (unsigned char xdata *)(wp + 1);  // next word
      } else {
        *tmp = *tmp;                            // reload old data
        tmp++;                                  // next byte
      }
    }
    MEMCON &= 0xDF;    // LDPG=0
    *((unsigned char xdata *)ptr) = (unsigned char)0xff; // initiate page write
  }
  /*verify value */
  tmp1 = *ptr;
  return(tmp1 != word);
}
```

6.3 write_eeprom_byte2()

```
/*---------------------------------------------------------------------
 * EEPROM Byte Write Emulation (Half Page Version)
 *
 * ptr  - address of byte in FDATA to write
 * byte - value to write
 * buf  - address of half page buffer in XRAM
 *---------------------------------------------------------------------
 */
char write_eeprom_byte2(unsigned char xdata *ptr, unsigned char byte,
                        unsigned char xdata *buf) {
  unsigned char xdata *tmp;
  unsigned char xdata *sav, *lowp;
  unsigned char i, tmp1;
  unsigned int adr;

  tmp1 = *ptr;         // fetch old value
  adr = (int)ptr;      // cast pointer to int for arithmetic

  MEMCON = 0x18;       // DMEN=1 MWEN=1 LDPG=0
  if (byte == tmp1) {
    /* no update required */
    return(0);
  } else {
    /* reload & erase page */
    MEMCON |= 0x60;    // LDPG=1 AERS=1
    tmp = (unsigned char xdata *)(adr & ~(PAGESIZE-1));// zero page
    lowp = tmp;
    sav = buf;
    /* save low half page to RAM buffer */
    for (i=PAGESIZE/2; i>0; i--) {
      if( tmp == ptr) {
        *sav = byte;
      } else {
        *sav = *tmp;
      }
      tmp++;
      sav++;
    }
    sav = tmp;
    /* reload high half page to temp buffer */
    for (i=PAGESIZE/2; i>0; i--) {
      if( tmp == ptr) {
        *tmp = byte;
      } else {
        *tmp = *tmp;
      }
      tmp++;
    }
    MEMCON &= 0xDF;    // LDPG=0
    *sav = 0xff;       // write high half page
    MEMCON &= 0xBF;    // AERSG=0
    MEMCON |= 0x40;    // LDPG=1
    sav = buf;
```

```
    tmp = lowp;
    /* load low half page from RAM */
    for (i=PAGESIZE/2; i>0; i--) {
      *tmp = *sav;
      tmp++;
    }
    MEMCON &= 0xDF;    // LDPG=0
    *lowp = 0xff;      // write low half page
  }
  /* verify value */
  tmp1 = *ptr;
  return(tmp1 != byte);
}
```

6.4 write_eeprom_word2()

```
/*---------------------------------------------------------------------------
 * EEPROM Word Write Emulation (Half Page Version)
 *
 * ptr  - address of word in FDATA to write
 * word - value to write
 * buf  - address of half page buffer in XRAM
 *---------------------------------------------------------------------------
* /
void write_eeprom_word2(unsigned int xdata *ptr, unsigned int word,
                        unsigned char xdata *buf) {
  unsigned char xdata *tmp;
  unsigned char xdata *sav, *lowp;
  unsigned int xdata *wp;
  unsigned int i, tmp1;
  unsigned int adr;

  tmp1 = *ptr;         // fetch old value
  adr = (int)ptr;      // cast pointer to int for arithmetic

  MEMCON = 0x38;       // DMEN=1 MWEN=1 LDPG=0
  if (word == tmp1) {
    /* no update required */
    return(0);
  } else {
    /* reload & erase page */
    MEMCON |= 0x60;    // LDPG=1 AERS=1
    tmp = (unsigned char xdata *)(adr & ~(PAGESIZE-1));// zero page
    lowp = tmp;
    sav = buf;
    /* save low half page to RAM buffer */
    for (i=PAGESIZE/2; i>0; i--) {
      if( tmp == (unsigned char xdata*)ptr) {
        wp = (unsigned int xdata *)sav;
        *wp = word;
        sav = (unsigned char xdata *)(wp + 1); // next word
        wp = ((unsigned int xdata *)tmp ;
        tmp = (unsigned char xdata *)(tmp + 1); // next word
      } else {
        *sav = *tmp;
```

```
      tmp++;
      sav++;
    }
  }
  sav = tmp;
  /* reload high half page to temp buffer */
  for (i=PAGESIZE/2; i>0; i--) {
    if( tmp == (unsigned char xdata *)ptr) {
      wp = (unsigned int xdata *)tmp;
      *wp = word;
      tmp = (unsigned char xdata *)(wp + 1); // next word
    } else {
      *tmp = *tmp;
      tmp++;
    }
  }
  MEMCON &= 0xDF;    // LDPG=0
  *sav = 0xff;       // write high half page
  MEMCON &= 0xBF;    // AERSG=0
  MEMCON |= 0x40;    // LDPG=1
  sav = buf;
  tmp = lowp;
  /* load low half page from RAM */
  for (i=PAGESIZE/2; i>0; i--) {
    *tmp = *sav;
    tmp++;
  }
  MEMCON &= 0xDF;    // LDPG=0
  *lowp = 0xff;      // write low half page
 }
 /* verify value */
 tmp1 = *ptr;
 return(tmp1 != word);
}
```

7. Revision History

Revision A – October 2009	• Initial Release

Headquarters

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

International

Atmel Asia
Unit 1-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
Hong Kong
Tel: (852) 2245-6100
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel 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

Product Contact

Web Site
www.atmel.com

Technical Support
mcu@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Requests
www.atmel.com/literature