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, but EEPROM cannot match the speed of Flash memory when multiple bytes need to be stored at the same time.

Flash memory is more efficient for larger parameter sets because of page programming, which decreases programming time. The average programming time per byte is lower for Flash than for EEPROM when storing multi-byte parameter sets. Furthermore, Flash memory’s separate write and erase operations allow better management of nonvolatile memory endurance.

This application note describes how to use the on-chip Flash data memory of Atmel’s AT89LP series of microcontrollers. Low-level Flash access routines are provided in assembly with C programming language wrappers for the Keil® C51 compiler. Special attention is paid to achieving high endurance storage.

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 Operation

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.
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

<table>
<thead>
<tr>
<th>Feature</th>
<th>EEPROM</th>
<th>FLASH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Erase Size</td>
<td>single byte</td>
<td>one page (multiple bytes)</td>
</tr>
<tr>
<td>Minimum Write Size</td>
<td>single byte</td>
<td>single bit</td>
</tr>
<tr>
<td>Maximum Write Size</td>
<td>single byte</td>
<td>one page (multiple bytes)</td>
</tr>
<tr>
<td>Read Size</td>
<td>byte</td>
<td>byte</td>
</tr>
<tr>
<td>Read Speed</td>
<td>fast</td>
<td>fast</td>
</tr>
<tr>
<td>Write Speed (per byte)</td>
<td>very slow (milliseconds)</td>
<td>slow (10s of microseconds)</td>
</tr>
</tbody>
</table>

### 2.2 Flash Data Retention and Endurance

The duration of time that a programmable nonvolatile memory can preserve the correct values is referred to as the data retention time (or just data retention). The reason why the memory does not have infinite data retention is that the memory cells are leaking, albeit very slowly. This means that the charge stored in the cells is weakened over time and at some instance the charge no longer represents the logic level that it should have. Both the erased and the programmed cells are leaking towards an unpredictable state. If the data retention time is exceeded, the contents of the Flash memory become unreliable. Retention times of 10 to 20 years are common.

When erasing and writing the Flash cells they are physically worn. As the number of erase/write cycles increases, the leakage from the cells increases. The consequence is that the charge is weakened faster and that the data therefore will become invalid sooner – in other words, the data retention time is decreased. Flash devices specify a maximum number of erase/write cycles for which cell performance will remain within acceptable tolerances. This maximum number of programming cycles is referred to as the endurance. Endurance limits generally vary from 10,000 to 1 million cycles depending on the technology. For this purpose one erase followed by one write to the same location counts as one cycle. Read accesses do not affect data retention and are not included in the endurance limit.

In relation to the endurance/retention of Flash storage it is important to know that the Flash pages are independent of each other; erase/write operations performed on one page do not affect the endurance/retention of other pages in the memory. However, due to the page erase structure the endurance of bytes within a page are dependent on each other because bytes that must maintain their value must be erased and reprogrammed when updating other byte values.
For example if a single byte in a page is updated many times while the other bytes are held constant, all bytes in the page will see reduced endurance, not just the affected byte.

### 2.2.1 Increasing Endurance

Some applications require more erase/write cycles than are allowed by the endurance limit. To achieve higher endurance levels, a software routine is required to evenly spread the wear across a multitude of cells. This is generally implemented as a circular buffer where N copies of the parameter are stored. For example, instead of writing a parameter to the same location every time, which requires an erase/write cycle, space is allocated for 10 copies of the parameter and at each update a write occurs at a new location. An erase is only required every tenth update when the buffer is fully occupied and must be cleared. Therefore the effective endurance is increased by a factor of 10. If the base endurance is 10,000 cycles then the effective endurance becomes 100,000 cycles.

There are two drawbacks to this method. First, to achieve an N factor increase in endurance requires N times more memory. Second, some type of pointer, also stored in nonvolatile memory, is required to find the last valid location in the buffer. Implementing such a system is beyond the scope of this application note.

### 2.3 Power Loss Considerations

One must assume that any embedded system can be exposed to power failures. This is one of the reasons for using nonvolatile memory types for parameters; it will then be possible to recover the parameters after power cycling. However, due to the length of Flash programming operations, there is a greater chance of a power failure occurring while programming is in progress. In this case the programmed data is not reliable and must be discarded.

There are many different methods to verify that parameters have been written correctly. The preferred choice depends on the time and code space available to do the verification. A safe method that uses very little memory space and time, is keeping a write complete flag in a static part of the memory. This flag can be used when recovering from a power failure to determine if the last write was completed correctly. If not, appropriate actions can be taken.

AT89LP microcontrollers use an on-chip Write Inhibit circuit to block Flash programming at low voltages. This circuit will prevent unwanted programming operations below the minimum required programming voltage. However, unwanted programming operations can still occur due to illegal execution if the minimum operating voltage for the operating frequency is above this level. To avoid Flash corruption due to power loss in these cases it is recommended to use an external Brown-out Detection circuit to reset the device at the appropriate supply level.

### 2.4 Data Considerations

The size and placement of parameters is crucial in achieving the highest efficiency from Flash Data memory. The minimum programming time for a page in Flash memory is on the order of a few milliseconds (comparable to an EEPROM byte operation). If only a single byte is written there is no speed advantage over EEPROM. As many bytes as possible should be programmed at one time to reduce the average write time per byte. If practical, smaller-sized parameters can be lumped together and updated at the same time. Depending on the requirements of the application, parameters can be cached in volatile RAM until a larger percentage of a page needs updating.

Reducing the number of page operations required for an update is also beneficial. Small parameters (less than the page size) should be aligned within the page boundaries such that only a
single page operation is required to update them. Larger parameters should straddle as few pages as possible to reduce the total required update time.

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)

```
<table>
<thead>
<tr>
<th>Page Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Memory</td>
</tr>
<tr>
<td>Page N</td>
</tr>
</tbody>
</table>
```

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

```
<table>
<thead>
<tr>
<th>Page Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Memory</td>
</tr>
<tr>
<td>Low Half Page</td>
</tr>
<tr>
<td>High Half Page</td>
</tr>
</tbody>
</table>
```

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.

```asm
; 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
```
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>
<thead>
<tr>
<th>Table 3-1. MEMCON – Memory Control Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMCON = 96H</td>
</tr>
<tr>
<td>Not Bit Addressable</td>
</tr>
<tr>
<td>Bit</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAP</td>
<td>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.</td>
</tr>
<tr>
<td>AERS</td>
<td>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.</td>
</tr>
<tr>
<td>LDPG</td>
<td>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.</td>
</tr>
<tr>
<td>MWEN</td>
<td>Memory Write Enable. Set to enable programming of a nonvolatile memory location (CODE, SIG or FDATA). Clear to disable programming of all nonvolatile memories.</td>
</tr>
<tr>
<td>DMEN</td>
<td>Data Memory Enable. Set to enable nonvolatile data memory and map it into the FDATA space. Clear to disable nonvolatile data memory.</td>
</tr>
<tr>
<td>ERR</td>
<td>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.</td>
</tr>
<tr>
<td>WRTINH</td>
<td>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.</td>
</tr>
</tbody>
</table>
4. Software

This application note provides two source files:

- fdata_api.c
- fdata_lib.a51

and two header files:

- fdata_api.h
- fdata_lib.h

Files are provided for both the AT89LP828 and the AT89LP6440. To use the provided routines, simply add the source files “flash_api.c” and “flash_lib.a51” to your project and include “flash_api.h” in all C source files that use the library. You also need to include a standard device-specific header file such as “at89lp828.h” or “at89lp6440.h”. Assembly projects need only to add and link “flash_lib.a51”.

In addition an example calling program “test_api.c” and a Keil μVision 4 project are included. The source files may be adapted to other compilers with slight modifications.

4.1 Function Descriptions

The following C functions are provided:

Table 4-1. Functions in flash_api.c

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>__api_rd_data_byte</td>
<td>adr</td>
<td>value</td>
</tr>
<tr>
<td>__api_wr_data_byte</td>
<td>address, value</td>
<td>none</td>
</tr>
<tr>
<td>__api_wr_data_page</td>
<td>add_flash, add_xram, nb_data</td>
<td>none</td>
</tr>
<tr>
<td>__api_ers_data_page</td>
<td>adr</td>
<td>none</td>
</tr>
<tr>
<td>__api_read_data_bytes</td>
<td>add_flash, add_xram, nb_data</td>
<td>none</td>
</tr>
</tbody>
</table>

4.1.1 __api_rd_data_byte

This function reads a byte in Flash Data memory for a given address.

- Prototype:
  - unsigned char __api_rd_data_byte(int adr)

- Inputs:
  - adr: Address in Flash Data memory of byte to read

- Outputs:
  - Return value read

- Example:

```c
unsigned char read_value;
read_value = __api_rd_data_byte(0x1000);
```

4.1.2 __api_wr_data_byte

This function writes a byte to Flash Data memory at a given address. If an erase is required, the entire page is erased and all the remaining bytes are reprogrammed with their previous values.

- Prototype:
void __api_wr_data_byte(int address, unsigned char value)

Inputs:
- address: Address in Flash Data memory of byte to write
- value: Value of byte to be written

Outputs:
- None

Example:
__api_wr_data_byte(0x1000, 0x55);

4.1.3 __api_wr_data_page

This function writes up to one page size of bytes from XRAM to Flash Data memory at a given
starting address. All data must be in the same page, i.e. the starting address plus the number
of bytes must still be in the same page range as the starting address.

If an erase is required, the behavior depends on defines in “flash_lib.a51”. By default if the num-
ber of bytes is less than the page size, the entire page is erased and the remaining bytes are
reprogrammed with their previous values. If WRITE_NO_READ_BACK is defined, the remaining
bytes are erased to 0xFF whenever an erase is required, but remain constant when an erase is
not required. If WRITE_FORCE_ERASE is defined, a page erase is performed on every page
write and any unspecified bytes with be erased to 0xFF.

Prototype:
- void __api_wr_data_page(int add_flash, int add_xram, unsigned char nb_data)

Inputs:
- add_flash: Starting address in Flash Data memory where data is to be written
- add_xram: Starting address in XRAM (EDATA or XDATA) of buffer for write values.
  The buffer must have allocated space for an entire page even if fewer bytes are
  written and must map to an XRAM range that is not mapped in FDATA.
- nb_data: Number of bytes to write, up to the page size (64 for AT89LP828, 128 for
  AT89LP6440)

Outputs:
- None

Example:
xdata unsigned char buffer[128] = {0x55,...,0x55};
__api_wr_data_page(0x1000, &buffer[], 128);

4.1.4 __api_ers_data_page

This function erases a page in Flash Data memory at a given address. The address may be any
byte within the desired page.

Prototype:
- void __api_ers_data_page(int adr)

Inputs:
- address: Address in Flash Data memory of page to erase

Outputs:
- None
4.1.5 __api_rd_data_bytes

This function reads (copies) up to 256 bytes from Flash Data memory at a given starting address to XRAM. Data do not need to be in the same page.

- Prototype:
  - void __api_rd_data_bytes(int add_flash, int add_xram, unsigned char nb_data)
- Inputs:
  - add_flash: Starting address in Flash Data memory where data is to be read
  - add_xram: Starting address in XRAM (EDATA or XDATA) to write values. The buffer must map to an XRAM range that is not mapped in FDATA.
  - nb_data: Number of bytes to read, up to a maximum of 256
- Outputs:
  - None
- Example:
  ```c
  xdata unsigned char buffer[128];
  __api_rd_data_bytes(0x1000, &buffer[], 128);
  ```

4.2 Requirements/Restrictions

The following requirements or restrictions must be considered for this application:

- As written, the memory page size is limited to 256 bytes or less. The page size is 64 bytes for AT89LP828 and 128 bytes for AT89LP6440.
- As written, the XRAM locations at the same address range as FDATA are not available for buffering read/write data.
- Write and erase routines disable interrupts during execution and re-enable them upon completion. Interrupts will not be serviced during their operation.
- Interrupts are not disabled during read routines. If an interrupt service routine modifies the data pointers, AUXR, AUXR/DPCF or MEMCON registers, interrupts must also be disabled during reads or the interrupt service routine must save and restore these values.
- The buffer in XRAM for holding write data for __api_wr_data_page() must be allocated to hold an entire page-worth of data even if fewer bytes will be written. The low-level routines will use this buffer for temporary storage should an erase and write back be required. This is not a requirement if either WRITE_NO_READ_BACK or WRITE_FORCE_ERASE is defined.

5. Revision History

| Revision A – January 2010 | Initial Release |