AN2540
Application note

EEPROM emulation in STR91xFxx devices

Introduction

Replacing external EEPROM with emulated EEPROM from the embedded-Flash memory of the microcontroller is a complex development. This application note is aimed at readers that are already familiar with the techniques used to secure the content of evolutive information in the external EEPROM of embedded applications.This application note explains the differences between external/internal EEPROMs and embedded-Flash memory. It also gives advice on how to replace external EEPROM with emulated-EEPROM using the on-chip Flash memory of STR91xFxx devices.

This document also focuses on some embedded aspects in emulated-EEPROM data storage, that are assumed to be known by the reader.

Overview

Electrically erasable and programmable read-only memory (EEPROM) is a key component in many embedded applications requiring non-volatile storage of data that are updated at a byte, half-word or word granularity during run time.

On the other hand, the microcontrollers used in those systems are more and more based on embedded-Flash memory. To eliminate components, save silicon area and reduce system cost, the STR91xFxx Flash memory could eventually replace the external EEPROM for simultaneous code and data storage.

However unlike Flash memory, external EEPROM does not require a block erase operation to free up space before data can be rewritten. A special software management is required to store data into Flash memory.

Obviously the emulation software scheme depends on many factors including the EEPROM reliability, Flash memory architecture and product requirements. Two approaches to implementation are described in detail in this application note using the on-chip Flash memory of the STR91xFxx microcontrollers.

Contents

List of tables

List of figures

1 Embedded Flash memory vs. EEPROM: main differences

Before describing the proposed concept for EEPROM emulation, it is important to remember the main differences between the embedded Flash memory of a microcontroller and serial external EEPROMs. These differences are the same for any microcontroller (that is they are not specific to STR91xFxx products). They are summarized in *Table 1*.

Table 1. Differences between Embedded Flash memory and EEPROM

Feature	External EEPROM	Emulated EEPROM using on-chip Flash memory
Write time	A few ms – random byte: 5 to 10 ms – page: equivalent to a hundred µs per word (5 to 10 ms per page)	A few µs (e.g.: 20 µs per 16-bit word)
Erase time	N/A	seconds (e.g.: 1.5 s)
Write method	Once started, is not CPU-dependent, needs only proper supply.	Once started, is CPU-dependent: a CPU reset will stop the write process even if the supply stays within specifications.
Read access	Serial: a hundred µs random word: 92 µs page: 22.5 µs/byte	Parallel: a hundred ns very few CPU cycles per 16-bit word.

1.1 Difference in write access time

As the Flash memory has a shorter write access time, critical parameters can be stored faster into the emulated EEPROM than into an external serial EEPROM. The use of the Flash memory therefore improves the system robustness.

1.2 Difference in writing method

One of the major differences between external and emulated EEPROM for embedded applications is the writing method.

- Standalone external EEPROM: once started by the CPU, the writing of a word cannot be interrupted by a CPU reset. Only a power supply failure can interrupt the writing process, so properly sizing the decoupling capacitors can secure the write process to a standalone EEPROM.
- Emulated EEPROM from an embedded Flash memory: once started by the CPU, the writing can be interrupted by a power failure and by a CPU reset. This difference should be analyzed by system designers to understand the possible impact(s) in their applications, and to determine a proper method to handle them.

1.3 Difference in erase time

The difference in erase time is the other major difference between standalone EEPROM and emulated EEPROM with embedded Flash memory. Unlike Flash memory, EEPROM does not require a block erase operation to free up space before writing. Moreover, as the erase process of a block in the Flash memory takes a few seconds, power shut-down and other spurious events that may interrupt the erase process (e.g.: reset) should be considered when designing the Flash memory management software. This means that to design a robust Flash memory management software it is necessary to have a good understanding of the Flash memory erase process.

2 Appropriate solution for Emulated EEPROM in the STR91xFxx

The STR91xFxx microcontrollers support the hardware and software architecture necessary to emulate EEPROM memory using the on-chip Flash memory.

2.1 STR91xFxx on-chip Flash memory features

- The STR91x internal Flash memory consists of two banks: Main Flash memory (Bank 0) and Secondary Flash memory (Bank 1). The Main Flash memory is up to 512 Kbytes in size and includes up to eight 64-Kbyte sectors. The Secondary Flash memory is 32 Kbytes in size and consists of four 8-Kbyte sectors, it can be useful for the wear-leveling feature (refer to *Section 4.2*).
- One of the STR9 embedded Flash memory features is Read-while-Write (RWW) Dual Bank operations. This means that the Main Flash memory (Bank0) can be used for code storage and the smaller Secondary Flash memory, for data storage (EEPROM emulation).
- The Flash memory can be erased on a sector or bank basis, and programmed on a 16-bit half-word basis.
- Each bank can be programmed and erased over 100 000 cycles.
- 20-year data retention.
- Each sector can be individually protected and unprotected against program and erase operations.
- As the Flash memory has a shorter write access time, critical parameters can be stored faster in the emulated EEPROM than in an external serial EEPROM.
- Interrupt servicing during program/erase is possible.
- CPU program does not need to be copied into RAM during program/erase: RAM less used to perform EEPROM emulation.
- Program/Erase Suspend and Resume commands supported. That is, Flash memory sector erase may be suspended while data is read from other sectors in the same Flash memory bank, and then resumed after reading.

2.2 STR91xFxx Flash memory library

The Flash memory programming library is a set of optimized C routines. It contains all that is needed to program the Flash memory embedded in STR9 devices.

The Flash memory library contains the following source files:

- 91x_fmi.c, that contains the function codes
- 91x_fmi.h, that contains the function prototypes

To use the functions provided by the Flash memory library, these two files must be added to the project. With the STR9 software (SW) library (FMI driver) it is easy to implement the EEPROM emulation software.

3 Implementing the EEPROM emulation

3.1 Principle

This emulation is performed in various ways by considering the Flash memory limitations and the product requirements. Two approaches are described below in detail. Both require a minimum of two Flash memory sectors of identical size, that are allocated to non-volatile data. One that is initially erased and can be programmed byte by byte, and the other that is ready to take over when the first sector needs to be garbage-collected.

Since the STR91xFxx on-chip Flash memory can be programmed on a 16-bit half-word basis, the data granularity in this implementation is 16 bits.

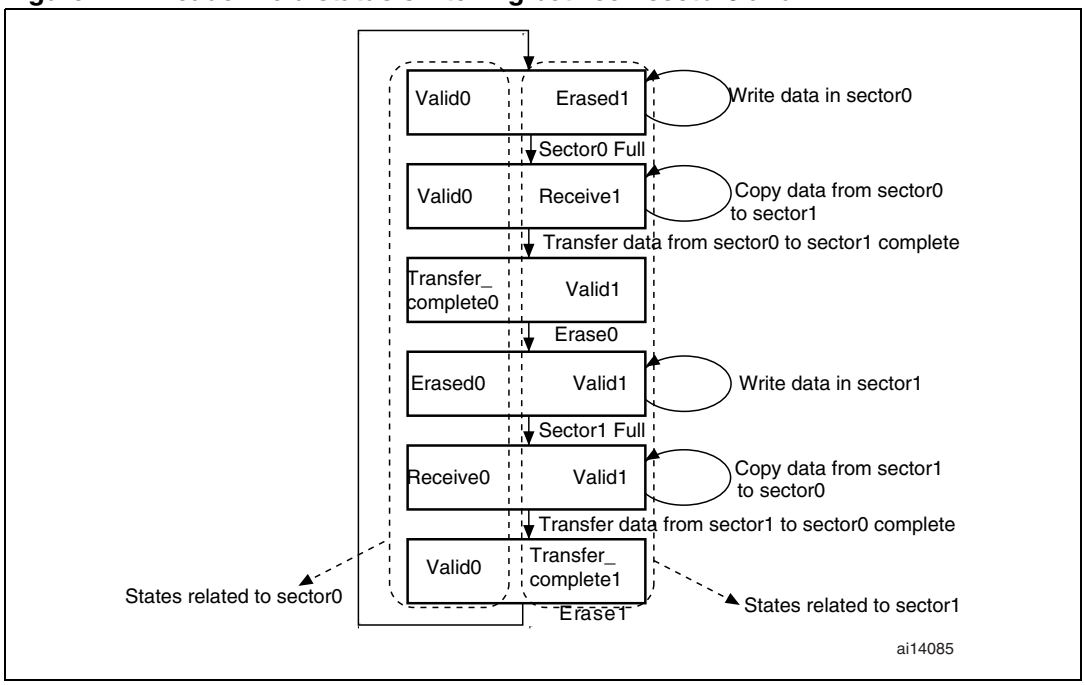
A header field that occupies the first 16-bit half word of each sector indicates the sector status.

Each sector has four possible states:

- **ERASED**: The sector is empty.
- **RECEIVE_DATA**: The sector is receiving data from the full sector.
- **VALID_SECTOR**: The sector contains valid data and its state will not change until valid data are completely transferred to the erased sector.
- **TRANSFER_COMPLETE:** Transfer of data to the other sector is finished and this sector is no longer in use. The system can then erase it and prepare it for future data.

Figure 1. shows how to switch from one state to another for both sectors.

Figure 1. Header field status switching between sector0 and 1



3.2 1st method

Parameter records stored in EEPROM vary in size and update frequency. Users using this method would usually know the update frequency in advance.

In this method, sector0 and sector1 in Bank1 are used. These Flash memory sectors are write-accessed in order to store several non-volatile variables. For this purpose, they have to be divided identically into several parts, one per variable. The size of the memory space allocated to each variable depends on the variable update frequency. The first 16-bit value of each variable is stored at the base address of the memory space allocated to the variable. When the variable is updated, the new value is stored at the next available address: Base address + 2, base address + 4 and so on until no room remains in the allocated memory space.

The 1st method Emulation driver meets the following requirements:

- At least two boot Flash memory sectors have to be used, more if possible for wear leveling (refer to *Section 4.2*)
- Minimum use of SRAM
- Simple and easily updatable code model
- User API consisting of EepromFormat, FindValidSector, WriteVariable, ReadVariable.
- Clean-up and internal data management transparent to the user
- Code in Main Flash memory, data storage in Secondary Flash memory

3.2.1 Application example

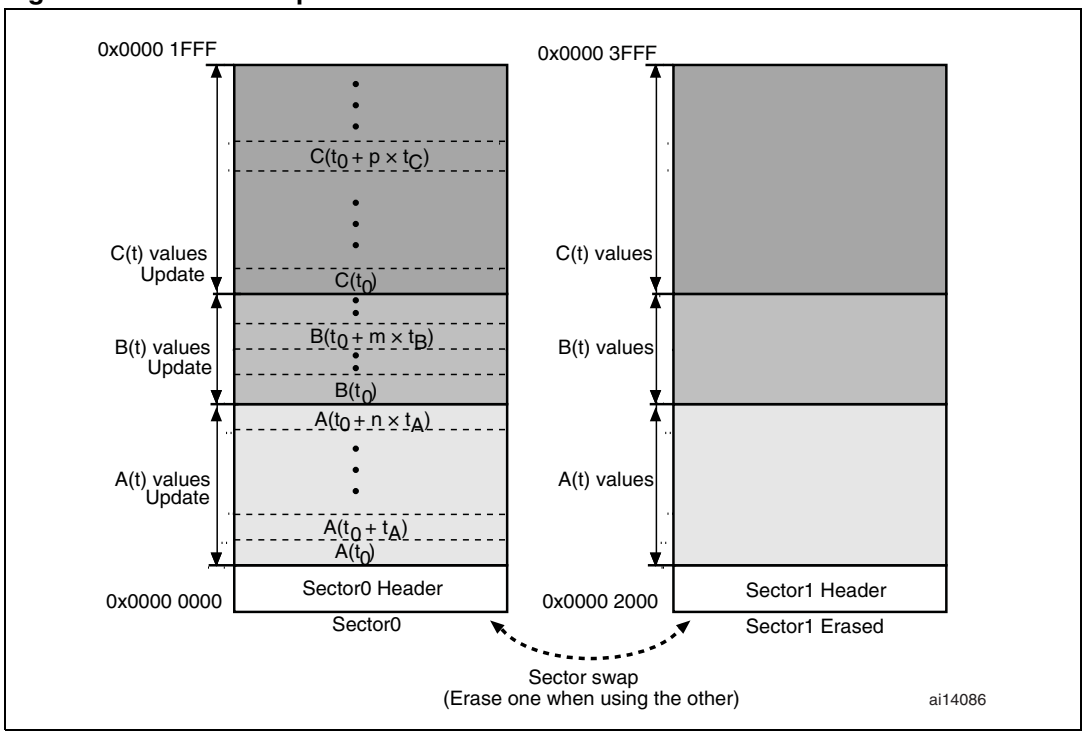
Let us assume that in sector0, three variables: A, B and C, will be stored and updated.

- The first variable A value is stored at t_0 and variable A(t) is updated every t_A.
- The second variable B value is stored at t_1 and variable B(t) is updated every t_B.
- The third variable C value is stored at t_2 and variable C(t) is updated every t_C.

In a typical application, the majority of non-volatile data are seldom updated, only a few data are updated more frequently.

Let us consider $t_C < t_A < t_B$. This means that C is updated more often than A and B, and that A is updated more often than B. So more memory space should be allocated to C than to A and B, and A should have more space than B (refer to *Figure 2*).

Figure 2. Sector swap scheme: Sector1 erased



In this application, sector0 is divided up as follows:

- 0x0000 0002 - 0x0000 04FF memory space is allocated to variable A
- 0x0000 0500 - 0x0000 0510 memory space is allocated to variable B
- 0x0000 0511 - 0x0000 1FFF memory space is allocated to variable C

In sector0, variables are stored until there the memory space left is not large enough to be allocated to another variable (case of variable A in the following example).

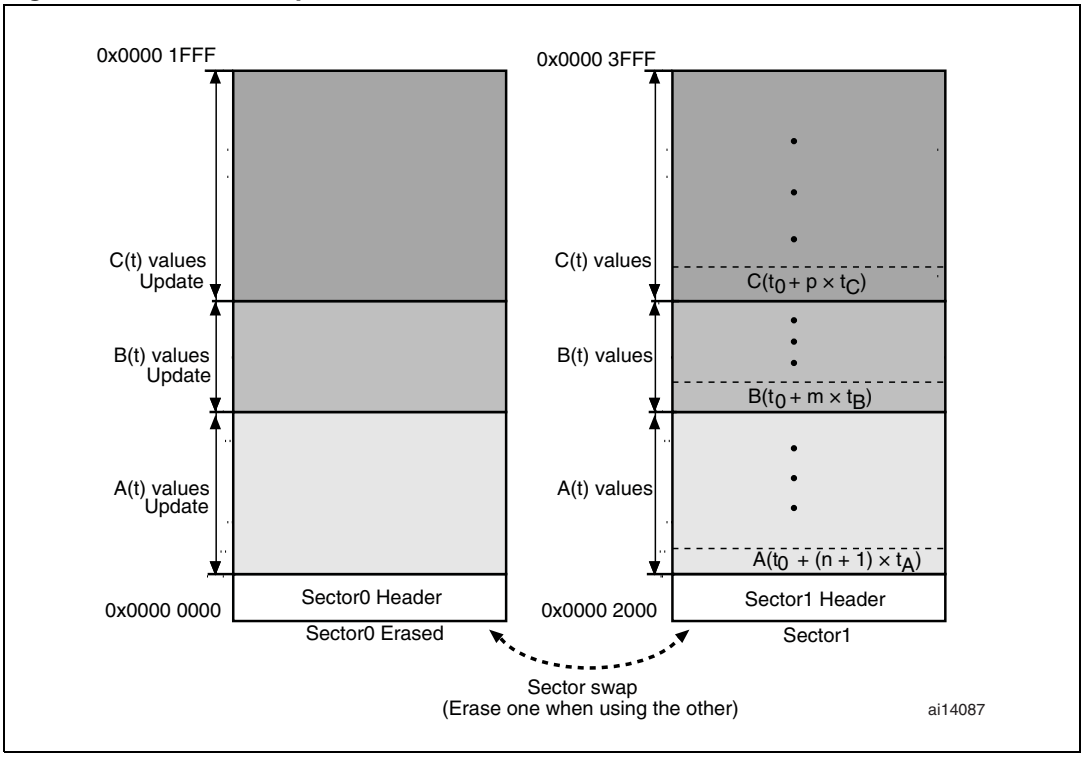
Example: Let us assuming that variable A is to be updated at time t0 + (n+1) × tA to A(t0 + (n+1) × tA). Addresses from 0x0000 0002 to 0x0000 04FF are all full, the next value is then stored in the base address of the space allocated to variable A in sector1.

The latest stored values of all other variables (B and C) are transferred from their current location in sector0 to the base address of their allocated space in sector1. After the transfer of all variables is complete, sector0 is erased.

Variables are then stored and updated in sector1 in the same way as described for sector0.

When there is not enough memory space left for one of these variables in sector1, the variables are transferred back to sector0 (now empty) as described above and so on. This process is illustrated in *Figure 3*.

Figure 3. Sector swap scheme: Sector0 erased



3.2.2 EEPROM software description

This section describes the driver implemented for EEPROM emulation using the STR91xFxx Flash library provided by STMicroelectronics.

A demonstration program is also supplied to demonstrate and test the EEPROM Emulation driver using the three variables A, B and C already defined.

The project contains three source files in addition to the FMI library source files:

- eeprom.c: containing C code for the following routines:

```
EepromFormat()
WriteVariable()
ReadVariable()
FindValidSector()
WriteVerifyVariableFull()
EepromSectorTransfer()
```

- eeprom.h: Containing the routines' prototypes and some declarations.
- main.c: This application program is an example using the routines described in eeprom.c.

3.2.2.1 User API definition

Table 2. EepromFormat

Function Name	EepromFormat
Function Prototype	`u8 EepromFormat(void);`
Behavior Description	This function erases sector0 and sector1 and writes a VALID_SECTOR header to sector0.
Input Parameter	None
Return Parameter	Status of the operation.
Called functions	FMI_EraseSector,FMI_WaitForLastOperation,FMI_WriteHalfWord

Table 3. FindValidSector

Function Name	FindValidSector
Function Prototype	`u8 FindValidSector(u8 operation);`
Behavior Description	This function reads both sector's headers and returns the sector number which contains valid data.
Input Parameter	A byte indicating that we are looking for a valid sector for write or read operation(READ_FROM_VALID_SECTOR or WRITE_IN_VALID_SECTOR
Return Parameter	Sector number.
Called functions	FMI_ReadWord.

Table 4. ReadVariable

Function Name	ReadVariable
Function Prototype	`u16 ReadVariable(u8 index, u32 *T);`
Behavior Description	This function reads variable data. Only last update is read. The function enters a loop in which it reads the variable entries until the last one is found. Finally, the data is returned.
Input Parameter	– index: variable identifier – T: variable array
Return Parameter	Returns 16-bit read data on success or error code on failure.
Called functions	FindValidSector.

Table 5. WriteVariable

Function Name	WriteVariable
Function Prototype	`u8 WriteVariable(u8 index, u32 *T, u16 data);`
Behavior Description	This function is called by the user application to update a variable
Input Parameter	– index: variable identifier – T: variable array – data: 16-bit data to be written
Return Parameter	Returns 1 on success or error code on failure
Called functions	WriteVerifyVariableFull, EepromSectorTransfer.

The procedure of updating a variable entry in the EEPROM is shown in *Figure 4*.

Figure 4. 1st method: WriteVariable flowchart

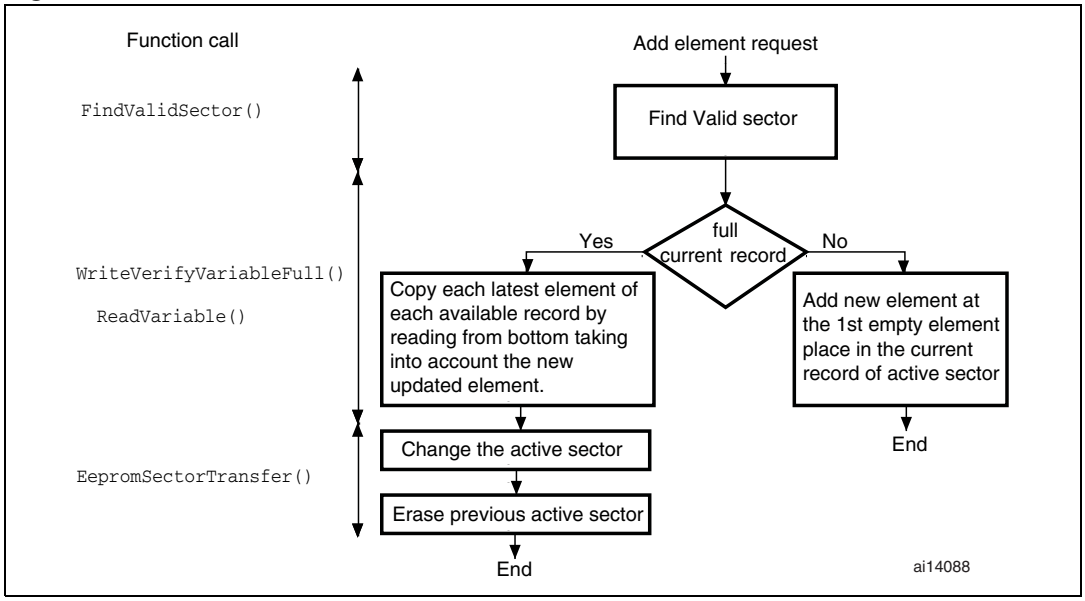


Table 6. WriteVerifyVariableFull

Function Name	WriteVerifyVariableFull
Function Prototype	`u16 WriteVerifyVariableFull(u8 index, u32 *T, u16Data);`
Behavior Description	If a write operation takes place, the write process must either update a variable, or create the first instance of a variable.
Input Parameter	– variable index, in the demonstration example provided: (0 for A, 1 for B or 2 for C). – T: variable Array – 16-bit data to be written
Return Parameter	Returns 1 on success or 0x80 if variable is full or Flash error code
Called functions	FindValidSector,FMI_ReadWord,FMI_WriteHalfWord

If the data to be written is equal to 0xFFFF, it is necessary to find the last data equal to 0xFFFF and to write 0x0000 into the next location to indicate that 0xFFFF represents data and not a blank location. If the data to be written is different from 0xFFFF, it is then written to the last location that contains 0xFFFFh (the following location should not contain 0x0000). This function returns 0 on success, VARIABLE_FULL if there is not enough memory space for a variable update, or a Flash error code indicating an operation failure (erase or program).

Table 7. EepromSectorTransfer

Function Name	EepromSectorTransfer
Function Prototype	`u8 EepromSectorTransfer(u8 index, u32 *T, u16 data);`
Behavior Description	It transfers the most recent data (Last variable updates) plus the new data from a full sector to an empty one.
Input Parameter	– index: variable identifier (0, 1, 2...) – T: variable array – data: 16-bit data to be written
Return Parameter	Returns 1 on success or error code on failure
Called functions	FindValidSector,FMI_WriteHalfWord, WriteVerifyVariableFull, ReadVariable, FMI_EraseSector.

At the beginning, the function determines the active sector which is the sector to be transferred from. The new sector header field is defined and written (new sector status is RECEIVE_DATA given that it is in the process of receiving data). When the data transfer is complete, the new sector header is marked VALID_SECTOR and the old one TRANSFER_COMPLETE. At the end, the old sector is erased.

3.2.2.2 Key features of the 1st method

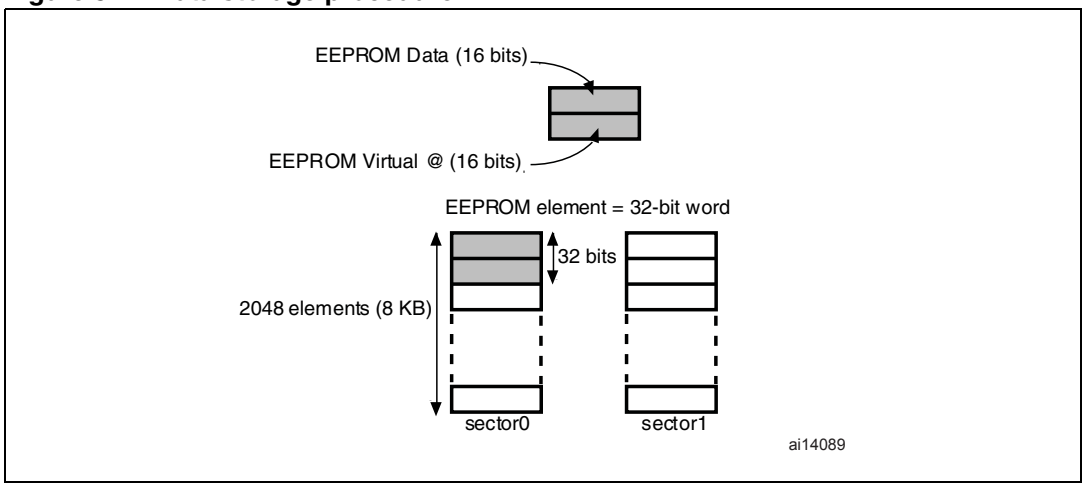
- User-configured emulated EEPROM size.
- The number of Flash program/erase cycles used can be minimized by permanently allocating a large memory space to the most frequently updated non-volatile data variables.
- ReadVariable and WriteVariable functions to access variables.
- The whole available memory space is used to store data: no need for virtual addresses.
- Fast read access to any variable since this simply implies going to the corresponding allocated memory space.
- Interrupt servicing during program/erase is possible.

3.3 2nd method

Generally when using this method, the user does not known in advance the update frequency of the variables. To emulate the EEPROM, two sector data structures are used.

Each data element is defined by a virtual address and its value to be stored into Flash memory locations for subsequent retrieval or update. When data is modified, the data associated with the earlier virtual address is stored into a new Flash memory location. During data retrieval, the modified data, in the latest Flash memory location is returned.

Figure 5. Data storage procedure



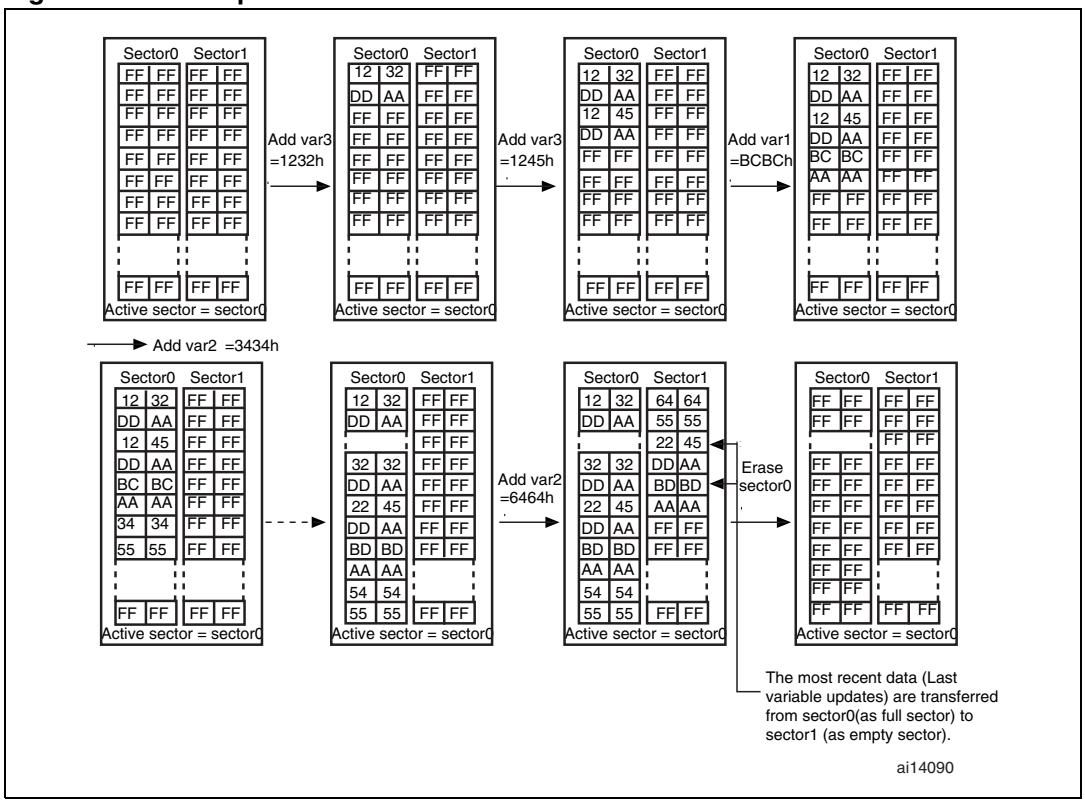
3.3.1 Application example

This example shows three EEPROM variables with the following virtual addresses:

- var1: AAAAh
- var2: 5555h
- var3: DDAAh

The data update flow is shown in *Figure 6.*

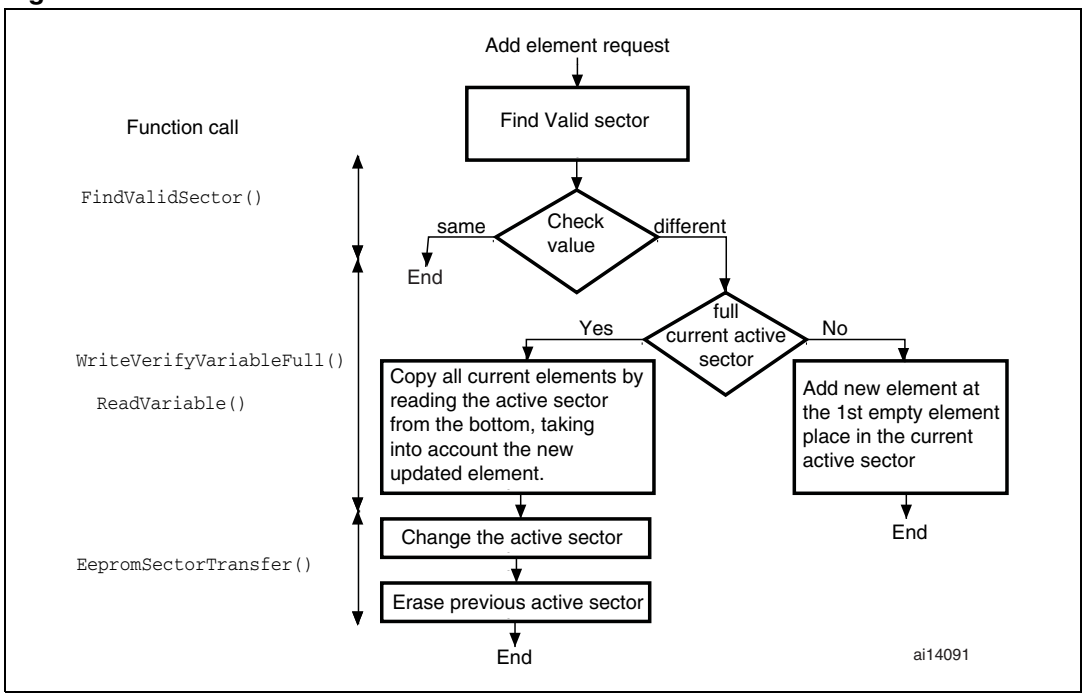
Figure 6. Data update flow



3.3.2 EEPROM software description

The user API function naming is the same as in the 1st method routines, the main differences are shown in the WriteVariable flowchart given in *Figure 7.*

Figure 7. 2nd method: WriteVariable flowchart



3.3.2.1 Key features of the 2nd method

- User-configured emulated EEPROM size.
- Increased Flash memory endurance: Sector erased only once it is full.
- Non-volatile data variables can be updated infrequently
- No need to perform a write operation if the updated value is the same.
- Interrupt servicing during program/erase is possible.

3.4 Program execution

Independently of the implementation scheme, the STR91xFxx internal Flash memory offers the ability to update a bank while code is executed from the other bank. Therefore, there is no need to transfer Flash memory operations (Program/Erase) to RAM. However when the CPU frequency is higher then 25 MHZ it is recommended to copy some routines into RAM This means that before erasing and programming, a few software routines have to be copied from the Flash memory into the on-chip RAM. Should be copied at least the routines used for the erase and program operations, and the routines that run while the erase or program operation is ongoing.

4 Embedded application aspects

This section gives some advice on how to overcome software limitations in embedded applications and to fulfill the needs of different applications.

4.1 Data granularity management

The emulated EEPROM can be used in embedded applications where non-volatile storage of data updated with a byte, half-word or word granularity is required. It generally depends on the user requirements and Flash architecture, such as stored data length, write access, etc.

The STR91xFxx on-chip Flash memory allows 16-bit, half-word programming. Data can however be programmed by bytes or words by using some software techniques.

4.1.1 Programming on a word-by-word basis

To write 32 bits of data "FMI_Data" to the desired Flash memory address "FMI_Address" in Bank1 for example, the process is the following:

- Write the LSB part of the data (the first two bytes) to the desired Flash memory address.
- Write the MSB part of the data (the last two bytes) to the Flash memory address incremented twice.

```
u16 LSB=FMI_Data;

u16  MSB= FMI_Data >>16;

FMI_WriteHalfWord ( FMI_Address,  LSB);
FMI_WaitForLastOperation(FMI_BANK_1);

FMI_WriteHalfWord ( FMI_Address+2,
MSB);FMI_WaitForLastOperation(FMI_BANK_1);
```

4.1.2 Programming on a byte-by-byte basis

Writing by bytes offers the user the possibility to cover the entire memory space and, therefore, to store more data. The performance may however be reduced.

A simple example can be useful to understand how to implement such a feature.

Let us assume for example that we want to write one byte of data "0xDD" into Bank1 at the FMI_Address. This can be achieved as follows:

If FMI_Address is even, then 0xFFDD is written. If FMI_Address is odd, 0xDDFF is written. The generic scheme is shown in *Figure 8.*

Figure 8. WriteOnebyte function description

```
void WriteVariable(u32 FMI_Address,u8 FMI_Data)
{
if (FMI_Address &1)
  {
    FMI_WriteHalfWord(FMI_Address,((FMI_Data << 8) | 0xFF));
    FMI_WaitForLastOperation(FMI_BANK_1);
  }
  else
  {
    FMI_WriteHalfWord(FMI_Address,(0xFF00 |FMI_Data));
    FMI_WaitForLastOperation(FMI_BANK_1);
  }
}
```

The above function is used to write the "FMI_Data" data byte to the "FMI_Address" address.

4.2 Wear-leveling: Flash endurance improvement

In the STR91xFxx on-chip Flash memory, each sector can be programmed or erased reliably over 100 000 times.

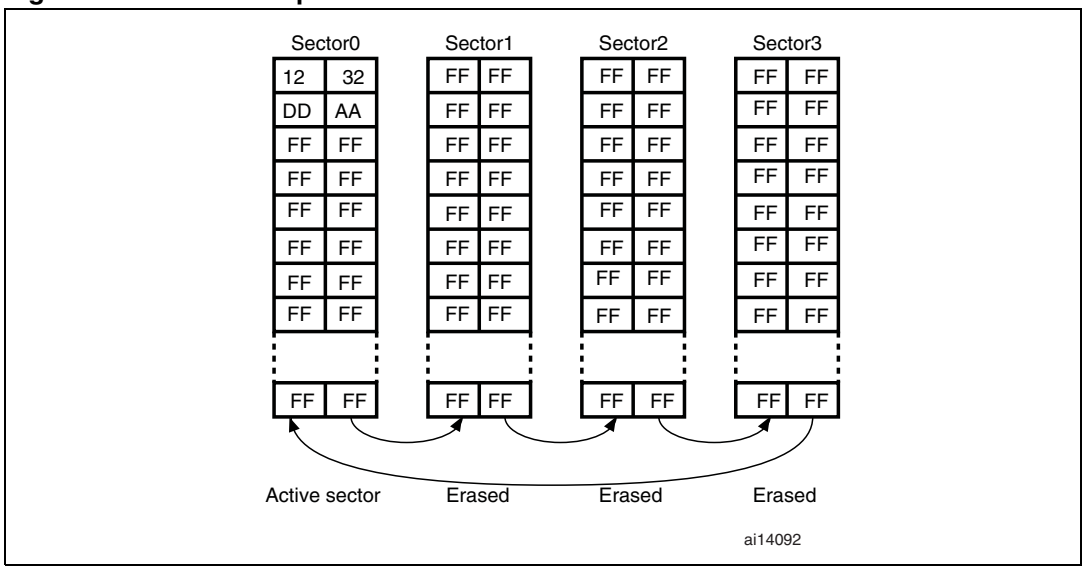
For write-intensive applications that use more than two sectors (3 or 4) for the emulated EEPROM, it is recommended to implement a wear-leveling algorithm to monitor and distribute the number of write cycles among the sectors.
When no wear-leveling algorithm is used, the sectors are not used at the same rate. Sectors with long-lived data do not endure as many write cycles as sectors that contain frequently updated data. The wear-leveling algorithm ensures that equal use is made of all the available write cycles for each sector.

4.2.1 Application example

In this example, in order to enhance the emulated EEPROM capacity, four sectors will be used.

To implement the wear-leveling algorithm with the 2^{nd} method scheme, the procedure is the following: when sector n is full, switch to sector n+1. Sector n is then garbage-collected. Let us consider the four sectors of the example: when sector3 is full, the device goes back to sector0, then sector 3 is garbage-collected and so on (refer to *Figure 9.*).

Figure 9. Sector swap scheme with four sectors



The previous algorithm can be implemented in the FindValidSector(...) function.

4.3 Sector header recovery in case of power loss

Data or sector header corruption is possible in case of a power loss during a variable update, a sector erase or a transfer.

To detect this corruption and recover it, an initialization routine should be called immediately after power-up.

After power loss, this routine is used to check the sector header status and to perform repair if necessary. There are 16 possible status combinations, half of which are invalid. The following table shows the actions that should be taken based on the sector states upon power-up.

Table 8. Status combinations and actions to be taken

Sector 1	Sector 0			
	ERASED	**RECEIVE_DATA**	**VALID_DATA**	**TRANSFER COMPLETE**
ERASED	Invalid state so erase both sectors and format sector 0	Invalid state so erase both sectors and format sector 0	Use sector 0 as valid sector and erase sector 1	Invalid state: erase both sectors and format sector 0
RECEIVE_DATA	Invalid state: erase both sectors and format sector 0	Invalid state: erase both sectors and format sector 0	Use sector 0 as valid sector & erase sector1 & transfer data from sector0 to sector1	Use sector 1 as valid sector & erase sector 0
VALID_DATA	Use sector 1 as valid sector and erase sector 0	Use sector 1 as valid sector & erase sector 0 & transfer data from sector1 to sector0	Invalid state: erase both sectors and format sector0	Use sector 1 as valid sector and erase sector 0
TRANSFER COMPLETE	Invalid state: erase both sectors and format sector 0	Use sector 0 as valid sector and erase sector 1	Use sector 0 as valid sector and erase sector 1	Invalid state: erase both sectors and format sector 0

4.4 Emulated EEPROM parameters

4.4.1 Program/Erase parameter cycling

One program/erase cycle consists of one or more write accesses and one sector erase operation.

When the EEPROM technology is used, each byte can be programmed and erased a finite number of times, typically in the range of 10 000 to 100 000.

However, when the Flash memory is used, the minimum erase size is the sector and the number of Program/Erase cycles applied to a sector is the number of erase cycles. The STR91xFxx electrical characteristics guarantee 100 000 Program/Erase cycles per sector.

Generally the maximum life of the EEPROM is thereby limited to the update rate of the most frequently written parameter.

In the 1st scheme application example, two sectors of 8 Kbytes are used and programming is done on a 16-bit half-word basis. If we consider that the memory space allocated to variable C is always filled before the memory spaces allocated to variables A and B, the expected number of erase cycles depends on variable C. As 6896 bytes are allocated to C, the variable can be updated 3448 times before it is switched to the other sector and the first sector is erased. Two sectors are used that can be erased 100 000 times so the total number of cycles we can expect for C is:

$3448 \times 2 \times 100\,000 = 689\,600\,000$ cycles.

4.4.2 Program timing

The following table gives an idea about the emulated EEPROM write time related to the current implementation.

It is clear that updating a variable with the 1st method scheme takes less time than with the 2nd method scheme since only the corresponding allocated memory space is accessed in the first case and not the whole sector.

Table 9. Write time related to the current implementation

Implemented scheme	Parameter	
	Write time (typical)	Write time (max)(1)
1st method scheme	20 µs	300 ms
2nd method scheme	30 µs	300 ms

1. This is the time taken to update a variable that makes a call to the EepromSectorTransfer(..) function. It is nearly equal to the sector erase time.

5 Conclusion

The external EEPROM can be replaced by an emulated EEPROM using the on-chip Flash memory of STR91xFxx devices. With the shorter Flash memory write access time, critical parameters are stored faster into the emulated EEPROM than into an external serial EEPROM. However, because the Flash memory needs to be erased before being written, some form of software management is required to store data into the emulated EEPROM.

Two methods are used to implement an emulated EEPROM: both require a minimum of two Flash memory sectors of identical size, allocated to non-volatile data. One that is initially erased and can be programmed byte by byte, and the other that is ready to take over when the first sector needs to be garbage-collected.

The emulated EEPROM can be used in embedded applications where non-volatile data storage is required, with a byte, half-word or word granularity. The STR91xFxx on-chip Flash memory allows 16-bit, half-word programming. Data can however be programmed by bytes or words by using some software techniques.

For write-intensive applications that use more than two sectors (3 or 4) for the emulated EEPROM, it is recommended to implement a wear-leveling algorithm to monitor and distribute the number of write cycles among the sectors.

6 Revision history

Table 10. Document revision history

Date	Revision	Changes
01-Jun-2007	1	Initial release.
05-Apr-2013	2	Changed part number to STR91xFxx. Updated Disclaimer.