Persistent and reliable FLASH-based data container for embedded systems

Michal Bližňák, Tomáš Dulík, Tomáš Juřena and Peter Janků

Abstracts— NOR-FLASH memories often available as an integral part of modern MCUs and SoC microprocessors can be used as an ideal storage container for persistent application data in embedded systems design. Unfortunately, a technological background of these memory chips avoids users from simple and efficient multiple writes into identical memory locations which is very common use case scenario when, for example, the application's configuration is updated. This paper introduces simple and elegant solution including its reference software implementation which allow users to write configuration data structures into the FLASH RAM in highly efficient and reliable way. Discussed approach minimizes number of needed write and erase cycles and maximizes reliability of the overall write process.

Keywords—FLASH, RAM, write, erase, persistent, data, efficiency, MCU, embedded

I. Introduction

MODERN microcontrollers used for embedded system design are often equipped with built-in FLASH RAM which can be used as a persistent storage container for various run-time or configuration data. The motivation for implementation of any type of persistent data storage can vary from need of simple saving of small amount of runtime data to storing complex data structures used for description of highly modular and configurable embedded system [1] or embedded databases with real-time access as introduced in [9]. For example, STM32F4 family of 32-bit microcontroller integrated circuits by STMicroelectronics based on ARM Cortex-M4 technology uses up to 2 MB of integrated FLASH memory used for both program instructions and data [7]. However, the nature of FLASH RAM avoids users from simple and efficient multiple writes to identical memory locations so a periodic update of specific data structure cells could be difficult to implement. This paper introduces new methods and software implementations which allow users to write various data structures into the FLASH RAM in highly efficient and safe way and minimizes number of needed write and erase cycles and maximizes reliability of the overall write process. The new software library can be used as lightweight and elegant alternative to existing similar solutions like presented in [2].

The work was performed with the financial support of the research project NPU I No. MSMT-7778/2014 by the Ministry of Education of the Czech Republic and also by the European Regional Development Fund under the project CEBIA-Tech No. CZ.1.05/2.1.00/03.0089.

Michal Bližňák is with the Tomas Bata University, Faculty of Applied Informatics, Department of Informatics and Artificial Intelligence, Nad Stranemi 4511, 76005 Zlin, Czech Republic (corresponding author to provide phone: 00420-576035187; e-mail: bliznak@fai.utb.cz).

Tomáš Dulík, Tomáš Juřena and Peter Janků are with the Tomas Bata University, Faculty of Applied InformaticsDefault form of internal data structure, Department of Informatics and Artificial Intelligence, Nad Stranemi 4511, 76005 Zlin, Czech Republic (e-mails: dulik@fai.utb.cz, jurena@fai.utb.cz and janku@fai.utb.cz).

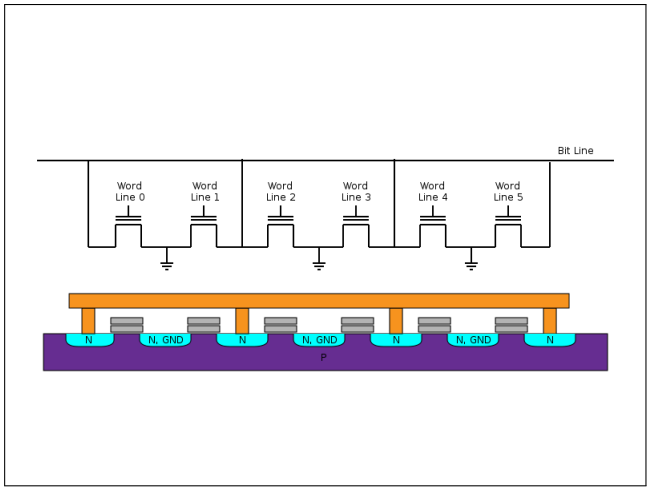

Fig. 1 Wiring of NOR flash memory [6]

Before discussing the proposed methods let us to examine basic FLASH memory characteristics first.

II. FLASH memories basics

FLASH memory (also called FLASH RAM) is a type of non-volatile memory device where stored data exists even when memory device is not electrically powered. It is an improved version of electrically erasable programmable read-only memory called EEPROM. The differences between FLASH memory and EEPROM are, EEPROM erases and rewrites its content one byte at a time, i.e at byte level. On the other hand, FLASH memory erases or writes its data in entire blocks, which makes it a very fast memory compared to EEPROM [5].

Basically, there are two main types of FLASH memory: NAND and NOR types. NAND type flash memory may be written and read in blocks which are generally much smaller than the entire device. NOR type flash allows a single machine word to be written to an erased location or read independently. Another differences between NAND and NOR FLASH memories are the following:

NOR-flash NOR-flash is slower in erase-operation and write-operation compared to NAND-flash. That means the NAND-flash has faster erase and write times. More over NAND has smaller erase units. So fewer erases are needed. NOR-flash can read data slightly faster than NAND.

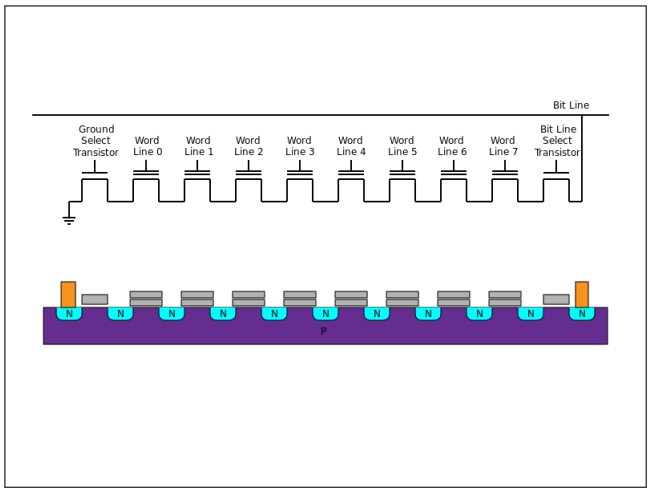

Fig. 2 Wiring of NAND flash memory [6]

NOR offers complete address and data buses to randomly access any of its memory location (addressable to every byte). This makes it a suitable replacement for older ROM BIOS/firmware chips, which rarely needs to be updated. Its endurance is 10,000 to 1,000,000 erase cycles. NOR is highly suitable for storing code in embedded systems thus most of the today's microcontrollers comes with built in flash memory [4].

In NOR gate flash, each cell has one end connected directly to ground, and the other end connected directly to a bit line. This arrangement is called "NOR flash" because it acts like a NOR gate: when one of the word lines (connected to the cell's CG) is brought high, the corresponding storage transistor acts to pull the output bit line low. NOR flash continues to be the technology of choice for embedded applications requiring a discrete nonvolatile memory device. The low read latencies characteristic of NOR devices allow for both direct code execution and data storage in a single memory product [**?**].

A single-level NOR flash cell in its default state is logically equivalent to a binary "1" value, because current will flow through the channel under application of an appropriate voltage to the control gate, so that the bit line voltage is pulled down. A NOR flash cell can be programmed or set to a binary "0" value by several possibles techniques [6].

NAND-flash NAND-flash occupies smaller chip area per cell. This maker NAND available in greater storage densities and at lower costs per bit than NOR-flash. It also has up to ten times the endurance of NOR-flash. NAND is more fit as storage media for large files including video and audio. The USB thumb drives, SD cards and MMC cards are of NAND type.

NAND-flash does not provide a random-access external address bus so the data must be read on a block-wise basis (also known as page access), where each block holds hundreds to thousands of bits, resembling to a kind of sequential data access. This is one of the main reasons why the NAND-flash is unsuitable to replace the ROM, because most of the microprocessors and microcontrollers require byte-level random access [5].

III. Main Limitations of FLASH memory

One limitation of FLASH memory is that, although it can be read or programmed a byte or a word at a time in a random access fashion, it can only be erased a "block" at a time. This generally sets all bits in the block to 1. Starting with a freshly erased block, any location within that block can be programmed. However, once a bit has been set to 0, only by erasing the entire block can it be changed back to 1. In other words, flash memory (specifically NOR flash) offers random-access read and programming operations, but does not offer arbitrary random-access rewrite or erase operations. A location can, however, be rewritten as long as the new value's 0 bits are a superset of the over-written values. For example, a nibble value may be erased to 1111, then written as 1110. Successive writes to that nibble can change it to 1010, then 0010, and finally 0000. Essentially, erasure sets all bits to 1, and programming can only clear bits to 0 [6].

Another limitation is that flash memory has a finite number of program-erase cycles (typically written as P/E cycles). Most commercially available flash products are guaranteed to withstand around 100,000 P/E cycles before the wear begins to deteriorate the integrity of the storage [3]. Micron Technology and Sun Microsystems announced an SLC NAND flash memory chip rated for 1,000,000 P/E cycles in 2008.

IV. Efficient and Safe FLASH Memory I/O Access

To eliminate the limitations mentioned in chapter III. a new software library aimed to STM32 microcontrollers called *PersistentData* was developed at Tomas Bata University. The library uses new approaches for accessing data written to FLASH memory integrated on the MCU's chip.

The main idea covered by the library is to store the data in internal structure similar to linked list where just real differences between old and modified data are written to the FLASH memory. Also, the library checks bit states of word to be written to determine whether the new data can be re-programmed at the current memory address location (thanks to the technology background explained in chapters II. and III.) or whether it should be stored to new empty (erased) location.

Now, let us to discuss the new write methods in more details.

A. Internal Data Structure

Main block of integrated FLASH memory available on STM32F2 and F4 families is organized into several (typically 12) sectors with variable sizes as shown in Table 1 [8].

The *PersistentData* library uses two sectors with identical sizes to store managed data structures. The reason why two dedicated FLASH sectors are used at the same time is due to mirroring of stored data which allows the library to minimize possibility of data loss when the MCU is powered down during a data write process accidentally as discussed later in this article.

Maximum size of data stored in the FLASH by using the library is calculated in 1.

$$maxSize = \frac{sectorSize - 8}{2} \quad (1)$$

The additional memory overhead is caused by a way how the library tries to keep managed data in valid state. The validity of the data is verified by a *CRC checksum* placed behind the data

Table 1: STM32's FLASH module organization

Name	Block base addresses	Size
Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbyte
Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbyte
Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbyte
Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbyte
Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbyte
Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbyte
...	...	...
Sector 11	0x080E 0000 - 0x080F FFFF	128 Kbyte

section and by validity words following raw data chunks as can be seen in Figure 3.

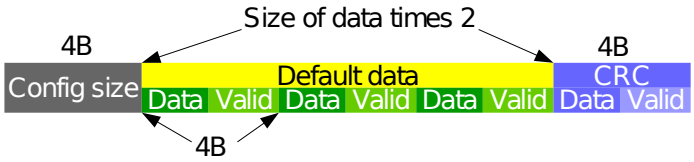

Fig. 3 default form of internal data structure

The raw data stored in the FLASH memory by using the library is divided into 16-bit words (*data chunks*) organized in the following way.

The overall length of internal data structure written as 32-bit double word is located at the beginning of the FLASH sector. The raw data interlaced by *validity words* follows. Size of the data section is doubled size of stored raw data because each data chunk is followed by its validity word with the same size (16 bits). At the end of the internal structure 16-bit *CRC checksum* of the data section followed by its validity word is placed.

The validity word's value is `0x7fff` by default which means the data chunk placed before is valid. Different validity value means relative offset in the FLASH memory where changed data chunk is stored.

B. *Writing of Data*

When the raw data structure is modified and the library is asked to write the modifications to FLASH memory, differences in the raw data are found. Then, the modified raw data chunks (i.e. modified 16-bits words) are written to empty (erased) FLASH memory locations after the last known (and valid) CRC checksum and the validity words associated with their previous values are updated so they point to new memory locations occupied by the new data chunks. The new data chunks are interlaced with default validity words like in the previous state. Finally, new CRC checksum is calculated and placed after the last used memory location (followed by its own validity word). Obviously, the validity word of previous CRC checksum is updated similarly to data chunk's validity word.

For minimizing of data writes into new memory locations, the library checks bit states of modified raw data chunk and writes the new value to the new memory location only when the old value cannot be re-written at the place. As mentioned in the chapter III., a content of FLASH memory can be re-written at identical memory location when bit states of the new value change just from "1" to "0". In this case, the validity word associated with the in-place modified data chunk remains unchanged (i.e. in default state) and just CRC checksum is modified (in the similar way).

Modified data structure written in FLASH memory is illustrated in Figure 4.

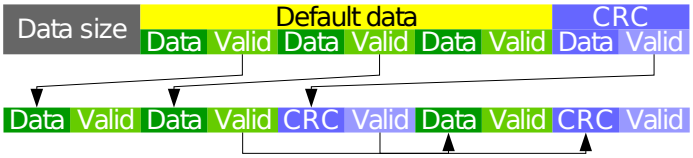

Fig. 4 modified internal data structure

Remember that the write changes are mirrored in both used FLASH memory sectors so the written data can be read successfully even when one of the dedicated sectors is corrupted. All tasks done during writing of the data by using `PersistentData_Set()` API function declared in the library are shown in Figure 5.

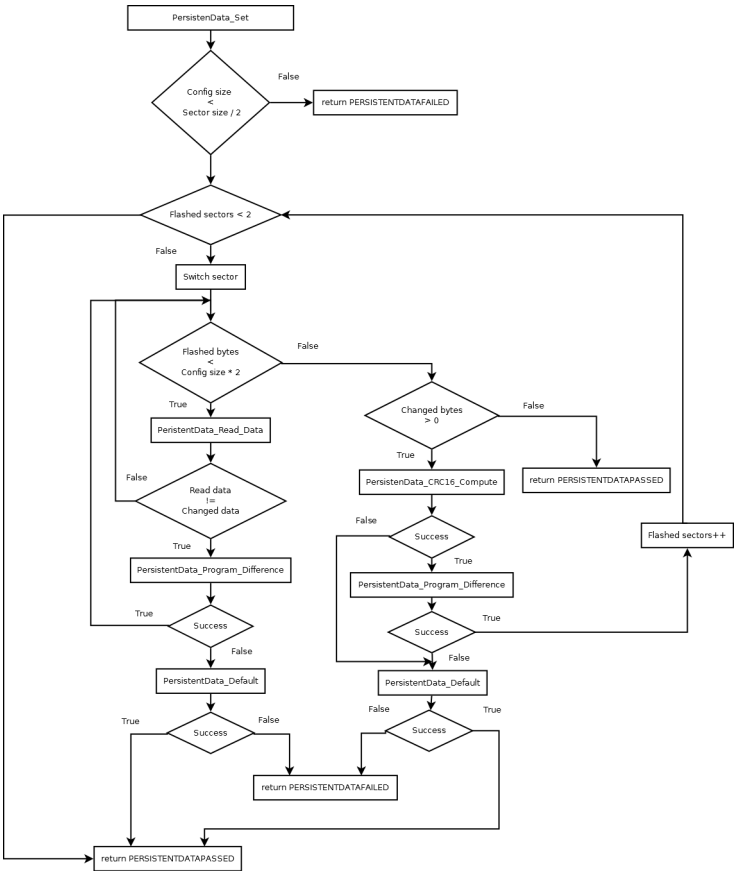

Fig. 5 activity diagram of `PersistentData_Set()` function

C. *Reading of Data*

When reading the stored raw data, the internal data structure written in FLASH memory is scanned by using content of the validity words and the data is reconstructed by joining the raw data chunks. During the scanning, the library reads values from validity words appended to data chunks and when found non-default

value which means that the data chunk is outdated (as mentioned above, the default value is `0x7fff`), the library jumps to new memory location calculated as current address of examined data chunk plus offset read from its validity word. Prior to that, a validity of whole (currently active) FLASH sector is checked by reading of the last know CRC checksum. When the CRC checksum mismatches or it cannot be read, the active FLASH sector is switched to the backup one and the data reading process is repeated. If the data cannot be read successfully even from the backup sector, the reading process fails and is aborted.

Tasks performed by the library during the data reading process are illustrated in Figure 6.

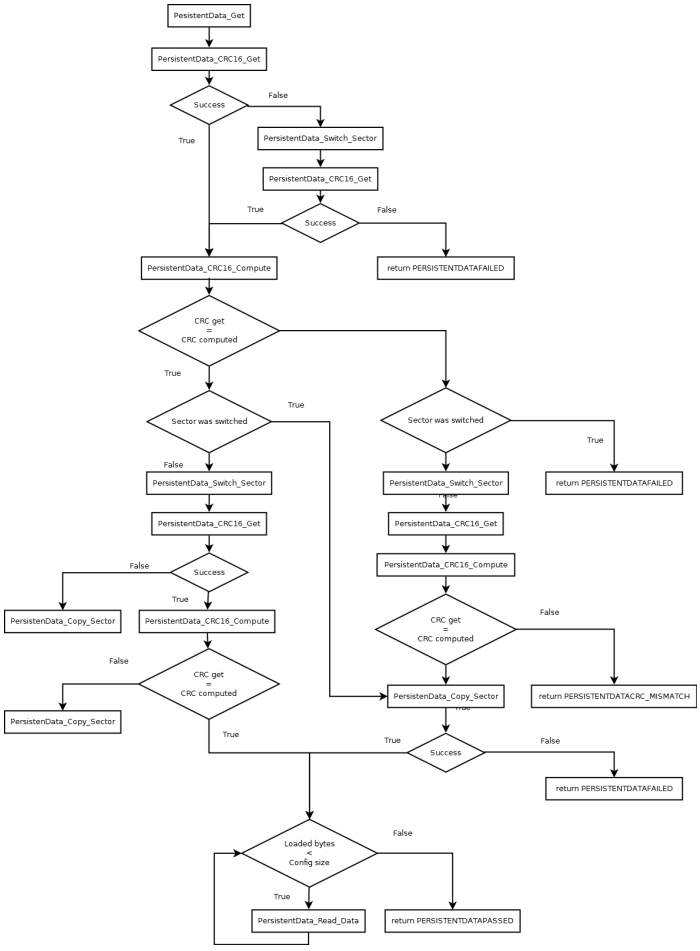

Fig. 6 activity diagram of `PersistentData_Get()` function

D. *Validation of Stored Data*

The validity of stored data is ensured by using CRC checksum which is calculated from raw data stored in the FLASH memory and it is compared with CRC value appended to the data section during the writing process as shown in Figure 3. Memory address of the first available CRC checksum can be calculated as in 2.

$$crcAddress = (2 \times dataSize) + 4 \tag{2}$$

The CRC checksum is regarded as valid itself when its assigned validity word contains the default value (`0x7fff`). If not, the up to date CRC checksum is searched in the same way like reading of data chunks described in chapter C.. During the scanning of CRCs validity words the calculated absolute memory addresses are checked to unveil possible data corruption in the following way: when the address exceeds bounds of active FLASH memory sector or when new calculated address is lower or equal to the current one, then the CRC is declared as invalid, otherwise the CRC value is regarded as valid. By the way, an integrity of validity words appended to the data chunks is ensured in the same way.

V. Solving of Possible Error States

Unfortunately, a data loss may occur when MCU's electric power supply gets down during the writing process so both data and its validation components could remain incomplete. Due to this, the CRC checksums are calculated to ensure correct writing process and to detect random errors caused by the data loss. Let us to examine several critical scenarios which may occur and let us to see how they are handled by the *PersistenData* library.

A. *Accidental Power-down During the Write Process*

In case of MCU's electric power supply malfunction a data being written to FLASH memory could be damaged. If happened, the CRC checksum calculated from this data mismatches CRC value stored on this affected sector. Obviously, the writing process can be aborted in any phase so the errors can involve various data structure's parts. Here are four possible critical scenarios which may occur:

- The CRC checksum is not written after the last valid data chunks followed by its validity word

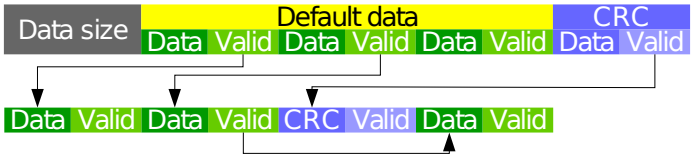

Fig. 7 CRC is not written after valid data

- Updated data chunk is not written after the update of validity word assigned to the outdated data chunk.

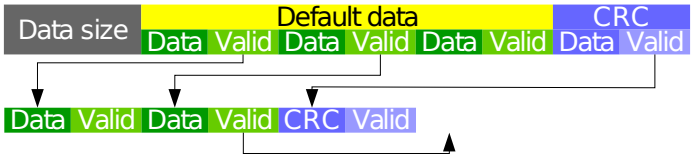

Fig. 8 new data are not written after validity word update

- Not all data chunks are written, but all the written are associated with correct validity words.

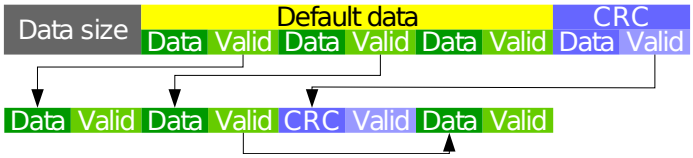

Fig. 9 data are written partially

- Not all data chunks are written; some of the written are without assigned validity word.

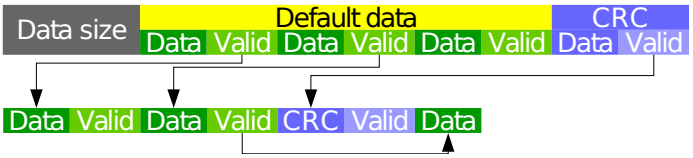

Fig. 10 data written partially, validity word is missing

All these scenarios are detected by the library and can by solved by common way: After the detection the library tries to recover the data from the backup FLASH memory sector. Of course, it is possible only when the backup sector contains valid, undamaged data.

B. *Reading of Corrupted Data*

During the reading, the CRC checksum could be missing or couldn't be found. Also, the calculated CRC value can mismatch the stored one. The first case occurs when the CRC's validation word is corrupted or it is missing. When both calculated and stored CRC checksums are present but they don't match then the data is corrupted or the CRC checksum hasn't been updated correctly. In this case the library tries to read requested data from the backup FLASH sector.

VI. PersistenData Library's API

Based on methods described in the previous chapters, a software library called *PersistentData* was developed at Tomas Bata University in Zlin. The library is written in ANSI C programming language and has been successfully deployed and tested on STM32F2 ad F4 chip families in conjunction with low level peripheral drives for on-chip FLASH memory provided by STMicroelectronics.

The library defines set of API functions aimed to save/load user-defined data structures to/from FLASH memory easily without need of any programming overhead. The API functions available in the library can be divided into two groups: the public ones defining API function intended to be used directly by the library's user and the private ones used by the library internally. During the reading or the writing of given data all the methods discussed in this article are used.

A. *Public API*

The public API defines following functions:

- `ERROR_T` **PersistentData_Get**(`uint16_t *data, uint16_t size`);
  The function reads data from FLASH memory to buffer specified by the user. Function's argument `data` points to the beginning of the buffer and `size` argument specifies its size. The function returns `ERR_OK` value on success, `ERR_FAIL` when it fails and `ERR_PD_CRC_MISSMATCH` when the CRC checkums don't match.

- `ERROR_T` **PersistentData_Set**(`const void *data, uint16_t size`);
  The function updates/writes user-defined data in/to FLASH memory. The `def_data` argument points to the beginning of user's data buffer, `size` argument specifies length of the data and `start_sector` argument determines target FLASH memory sector used as the primary one (the second one is regarded as the backup sector). The function returns `ERR_OK` value on success and `ERR_FAIL` when it fails.

- `ERROR_T` **PersistentData_Default**(`const void *def_data, uint16_t size, PD_SECTORS_T start_sector`);
  The function writes default user-defined data to the FLASH memory (i.e. initializes the memory with default data). The first function's argument `def_data` is pointer to the default, user-defined data bufer, the second argument `size` specifies size of the data and the last function argument called `start_sector` determines FLASH sector where the data will be stored to. Return values are `ERR_OK` on success and `ERR_FAIL` when the function fails.

- `bool` **PersistentData_Empty**(`void`);
  The last public function takes no argument and returns logical flag telling whether the FLASH memory contains any useful data written by the library.

B. *Sample Souce Code*

The following source code shows typical usage of the library. Assume there exists user-defined data structure called `CONFIGURATION_T` containing specific application's configuration data. Also assume two global instances of the structure defined in the user's source code called `config_global` containing current configuration data and `config_defaults` containing constant default data.

The first sample shown in Listing 1 soure code shows how a function suitable for reset of the application's configuration could look like.

Listing 1: Reset of an application's configuration data

```
1  /*!
2   * \brief Reset the global application configuration to
        its factory state.
3   */
4  void Config_Reset(void)
5  {
6      // copy default data to the current configuration
         instance
7    memcpy( &config_global, &config_defaults, sizeof(
       CONFIGURATION_T) );
8      // write the default data to FLASH memory into first
         data sector (specified by
9      // the user in the library's header file).
10   PersistentData_Default( &config_global, sizeof(
       CONFIGURATION_T), PD_SECTOR_1 );
11 }
```

The second sample illustrates how read data stored in FLASH memory. In the code dedicated FLASH memory sectors are examined first to determine whether they contain useful data previously written be the library. Next, the data are read by using `PersistentData_Get()` function. Also, the configuration's version number is checked to determine whether the data read from FLASH memory matches expected structure of `CONFIGURATION_T` data type.

Listing 2: Load application's configuration data from FLASH memory

```
1  /*!
```

```
2    * \brief Initialize global configuration structure with
         values stored in the FLASH memory.
3    * \retval ERROR_OK Configuration has been successfully
         loaded from FLASH memory
4    * \retval ERROR_FAIL No valid configuration has been
         found in FLASH memory
5    */
6   ERROR_T Config_Load()
7   {
8       // check whether the FLASH memory contains useful data
9     if( ! PersistentData_Empty() ) {
10          // try to load the data from FLASH memory
11      if( PersistentData_Get( (uint16_t*)&config_global,
              sizeof(CONFIGURATION_T) ) == ERR_OK ) {
12            // verify configuration version number
13        if( Config_VerifyVersionNumber() ) {
14          return ERR_OK;
15        }
16      }
17    }
18
19    return ERR_FAIL;
20  }
```

The last example shows how to save modified configuration data. Assume that the `CONFIGURATION_T` structure contains also counter indicating number of made changes which could be used by the user later.

Listing 3: Save application's configuration data to FLASH memory

```
1   /*!
2    * \brief Store current values from global configuration
         structure to the FLASH memory.
3    * \param config Pointer to source configuration structure
         .
4    * \retval ERROR_OK Configuration has been saved to FLASH
         memory successfully
5    * \retval ERROR_FAIL Configuration couldn't be saved to
         FLASH memory
6    */
7   ERROR_T Config_Save()
8   {
9     // increment configuration's modification counter
10    config_global.cfg_modification_number++;
11      // write the configuration to FLASH memory
12    return PersistentData_Set( &config_global, sizeof(
          CONFIGURATION_T) );
13  }
```

VII. Conclusion

All the principles discussed in this article lead to efficient and safe implementation of persistent data storage based on FLASH memory suitable for embedded systems. Based on the discussed principles, new software library called *PersistentData* has been developed and tested on STMicroelectronic's STM32F2 and STM32F4 MCU's families. Set of unit tests covering all critical functionality of the library have been created and used for testing. Also, the library has been successfully used in two commercial projects developed at the Tomas Bata University which proved its maturity.

References

[1] M. Otesteanu, "Embedded systems with adaptive architecture ", *WSEAS Transactions on Electronics* , vol. 2, issue 3, July 2005, pp. 100-107

[2] M. Short, M. Schwarz, J. Boercsoek, "Efficient implementation of fault-tolerant data structures in embedded control software", *WSEAS Transactions on Electronics* , vol. 5, issue 1, January 2008, pp. 12-24

[3] J. Thatcher, T. Coughlin, J. Handy and N. Ekker. (2009). NAND Flash Solid State Storage for the Enterprise. *SNIA, Solid State Storage Initiative* [Online]. Available: http://www.snia.org/sites/default/files/SSSI_NAND_Reliability_White_Paper_0.pdf

[4] C. Zitlaw. (2011). The Future of NOR flash memory. *EETimes* [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1278751&page_number=1

[5] Electronics Design Magazine. (2006). Flash memory basics and its interface to a processor. *EE Herald* [Online]. Available: http://www.eeherald.com/section/design-guide/esmod16.html

[6] Wikipedia. (2015). Flash memory. [Online]. Available: http://en.wikipedia.org/wiki/Flash_memory

[7] *STM32F4 Series*, STMicroelectronics Website. (2015). [Online]. Available: http://www.st.com/web/en/catalog/mmc/FM141/SC1169/SS1577

[8] *STM32F2 Flash Programming Manual*, STMicroelectronics Website. (2015). [Online]. Available: http://www.st.com/web/en/resource/ technical/document/programming_manual/CD00233952.pdf

[9] W. Hu, T. Chen, Q. Shi, N. Jiang, "A data centered approach for cache partitioning in embedded real-time database system", *WSEAS Transactions on Computers* , vol. 7, issue 3, March 2008, pp. 140-146