NVM Tag Design

Copyright © 2019, Bernie Woodland

All rights reserved.

Redistribution and use, with or without

modification, are permitted provided that the following conditions are met:

1. Redistributions must retain the above copyright notice, this

list of conditions and the following disclaimer.

THIS DOCUMENTATION IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE

DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;

LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENTATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Arm® is a trademark of Arm Holdings

Arm® Cortex® is a trademark of Arm HoldingsTable of Contents

[History 4](#_Toc9090687)

[This design is based on an open-source design from Intel back in the ‘90’s, 4](#_Toc9090688)

[Layout 4](#_Toc9090689)

[Tags 4](#_Toc9090690)

[Sector Reserved Area & Marking 5](#_Toc9090691)

[Sector View of a Dynamic System 5](#_Toc9090692)

[Operation 6](#_Toc9090693)

[Analysis 7](#_Toc9090694)

[System Description 8](#_Toc9090695)

[Tag Spaces 8](#_Toc9090696)

[Sector Reserved Space 8](#_Toc9090697)

[Tag and SectorTerminology 9](#_Toc9090698)

[Tags 9](#_Toc9090699)

[Tag Offsets 10](#_Toc9090700)

[Tag Header Field Descriptions 11](#_Toc9090701)

[Tag Writing Sequence 12](#_Toc9090702)

[Tag Rewrite Steps 12](#_Toc9090703)

[Garbage Collection Steps 12](#_Toc9090704)

[Reclaim Scoring Algorithm 13](#_Toc9090705)

[API’s 14](#_Toc9090706)

[Peformance Considerations 14](#_Toc9090707)

[Tag Space Sizing and Wear Life 14](#_Toc9090708)

[Tag Write Times 15](#_Toc9090709)

[Tag Read Times 16](#_Toc9090710)

History

This design is based on an open-source design from Intel back in the ‘90’s, , and was implemented in Ericsson’s TDMA phones (> 10 million produced). The design was called a *FLASH Tagging System*. Non-volatile storage is divided into units called tags, and these are optimized for storage.

# Analysis

This system of non-volatile data storage uses FLASH memory efficiently, is flexible in that it allows for tags of various sizes to coexist in the same system. Data integrity is extremely good—there are no unrecoverable ‘holes’ that can result from a random crash or power loss at any time in the processing of tag storage. Tags can be rewritten several times before an erase is needed. Boot time is extremely fast1. The RAM pointers allow for direct reads of tag contents without the need for semaphores or context switching—the RAM pointers can be cast to actual “C” structs and data can be read directly out of the tags.

The reclaim trigger algorithm is flexible; it can be made simple or complex. A sophisticated reclaim algorithm will keep track of the number of times each sector has been erased (storing this data in dedicated tags, for example), and attempt to spread the data storage, and thus reclaim cycles, in such a way that the erase loading is evenly distributed among the sectors, so as to minimize the erase cycles per sector, in the attempt to maximize the lifespan of the FLASH part, as every FLASH device has a finite number of erasures before it wears out.

A tag storage system is the ideal underlying layer for a true FLASH file system. A real file system dynamically creates and deletes files, which can be of various lengths, and the file system has a file name structure, complete with directories. What a tag storage system does is to insure the data integrity of the FLASH system, while it presents to the file system layer above it a block storage facility. Block storage on a tag system is accomplished by allocating uniform sized tags to a file system in the same way that sectors appear on a hard drive. Generic file systems typically mount block storage in this manner.

In more complex systems, the FLASH part or parts can be partitioned so that the tagging software controls 1 to N tag spaces, where each tag space consists of 2 to M FLASH sectors. In this way, simple non-volatile parameters can be stored in one partition, while a true file system resides in another partition. With the data parameters physically isolated from the file system, the tagging software can very quickly initialize the data parameter tag partition, making the crucial non-volatile parameters available in just a few milliseconds after powerup, while deferring the file system initialization to a later time. Also, for convenience, the file system can be wiped clean by erasing all the sectors dedicated to the file system partition.

# **Layout**

The use of FLASH parts to store non-volatile chunks of data is one of its primary functions. Typically, the data consists of a contiguous string of 1 – N bytes. The nature of FLASH memory is that a write operation can only change bits that are 1’s to 0’s. An erase operation is necessary to change 0’s back to 1’s, and while a write can be done on a single byte, an erase must be done on an entire sector. This nature has profound implications for FLASH data storage software designs.

## Tags

Using a tag storage scheme means that non-volatile data is stored in discrete units called tags. The stored data item is mapped to a single tag ID. Tags can be of various length, the practical maximum dictated by the sector size (usually a tag maximum length is, say, <= 10% of the sector size).

A tag consists of the actual data contents prepended with a header:

Data

Length

2 bytes

Version

2 bytes

Tag ID

2 bytes

Status

1 byte

Magic No.

1 byte

Magic Number: A single byte of value 0xA5. Used for sanity purposes only

Status Bits: Each field in the header, plus the data, have a bit associated with it. When the field has been fully written, the value is set to ‘0’. One of the bits indicates that the tag is dirty/garbage/not used/superseded.

Each sector has a 4-byte fixed-field at the end address of the sector. This has:

Status Bits:

* Bit 0: if set to ‘0’, marked ‘dirty’…sector needs to be erased and refreshed
* Bit 1: erase started

Tag ID: A value indicating which tag is which

Version: which version of the tag this is. Each time tag is rewritten or moved, its version is incremented

Length: 0 to N length of ‘Data’, not including header

## Sector Reserved Area & Marking

In any sector, the top 16 bytes and the bottom 16 bytes are not used for tag storage, but are reserved for marking the sector. This is optional, but it’s a good idea to reserve this space anyways, in case it might be useful in the future.

Here’s a memory map of a flash sector (Sector 1) at address 0x80000000 (hypothetical address).

Top of sector: 0x8000FFFF0 to 0x8000FFFF reserved

Bottom of sector: 0x80000000 to 0x80000010 reserved

Sector 1

(64k bytes)

Base address = 0x80000000

The 32 bytes of reserved areas of the sector can be (and this is optional) used to:

* Mark sectors as ‘dirty’, needing erasing
* Mark sectors as ready for use

## Sector View of a Dynamic System

This is a hypothetical memory mapping of a few sectors of FLASH that are being used by a tagging system. The large, vertical rectangular blocks are sectors (let’s say they’re 64kbytes). Addresses decrease in value as one goes down the page, so the bottom of each sector block is actually the start address of the sector. The blue and grey blocks represent tags which have been written. The blue are tags currently in use; the grey are obsoleted tags. The white area is not yet used/unwritten/0xFF areas of the sector.

Sector 0

Sector 1

Sector 2

Sector 3

# Operation

A tag is written by adding its header and then the data. The ‘version’ starts at 1. If the value of the tag changes, then, instead of erasing the old tag, a new one is written. The new tag’s version will be 2. The old tag’s ‘dirty’ state bit is marked (i.e. cleared), so that it’ll be clearly identified as garbage, to be ignored. Every time a tag’s data changes, the old tag must be obsolete, and a new tag must be written. A sector erase is not necessarily (and seldom is) required when rewriting a new tag due to a data change.

The tagging software will select one sector as the target sector for writing new tags in. It will fill the sector with tag data starting at the lowest address and incrementing until the sector is filled. As tags are randomly changed, the sector will fill with a combination of currently-in-use tags to obsolete tags. Once the sector has been filled, a new sector is chosen as the current working sector, and that is filled in the same manner. Once many or all the available sectors have been filled, the sectors will have been populated with a mixture of in-use and obsolete tags. At this point, the tagging software will look to do garbage collection by reclaiming sectors.

Each tag has a pointer in RAM identifying the address of the latest version of the tag. When the tag is rewritten, a new version of the tag is created, so the RAM pointer must be updated to point to this new version. At power-up, the FLASH tag-space will be scanned for the latest version of each tag, and the RAM pointers will be set to point to these. For the sake of read efficiency, the RAM pointer does not point to the beginning of the tag, but skips over the header and points to the start address of the data portion of the tag (which address is aligned on a 4-byte boundary).

Algorithms will be established to determine when a sector should be “reclaimed”. Reclaiming a sector consists of the following:

* All the valid/current version tags are copied to a new sector. Version numbers are incremented for freshly copied tags
* For tags that have just been copied, their previous version is marked as garbage.
* In this way, all tags in the sector to be reclaimed become garbage; all data has been moved out of this garbage sector
* Sector is marked as ‘dirty’, made pending for erase
* Sector is erased at a time when the system permits erasures, so as not to interfere with system operation
* Likewise the tagging software can be proactive about reclaiming sectors, so when a client needs to change a tag, there’s always storage available to accommodate the request without the need for an erase

At power-up, the tagging software does a thorough audit of the FLASH data storage sectors under its control. The software assumes a worst-case shutdown scenario, and therefore does the following at bootup:

* All sectors are scanned for tags
* Basic integrity checks of the sectors are performed: sector state bits are checked to see if the sector was marked for erase, if erase started and didn’t complete. Furthermore, the sector is checked to see if it is full and has not been marked as full.
* For any give tag ID, all versions are checked to insure that the latest is found. Older revisions are marked as dirty if not already so.
* Tags that were part way through with a write, but didn’t complete when powerdown occurred, are marked as obsolete

# System Description

This flash tagging system has the following highlights and features:

## Tag Spaces

The tagging system is partitioned into 1 to N tag “spaces”. A *space* is a single tag storage repository. A space has dedicated to it 1 to N external sectors. The following restrictions apply to a tag space:

* There must be a minimum of two sectors in a space
* The sectors must be contiguous
* The sectors must be either 64k sectors or 8k sectors, but not a mixture of the two
* Sectors dedicated to a tag space cannot be used by any other application
* Tag spaces cannot share sectors

There are advantages in having multiple tag spaces:

* The data from one sector is partitioned from another
* A “total reset” is done per space; a space can be wiped clean while leaving another intact
* The 64k and 8k sectors can both be used
* Smaller spaces can be monitored more easily than large spaces, so having the ability to break tags into different spaces makes things easier to manage
* Certain multi-tasking scenarios are simplified by the use of multiple tag spaces

## Sector Reserved Space

Certain parts of the sector are not used for tag storage. These are:

A. Reserved1: first 16 bytes

B. Reserved2: last 16 bytes

C. Headroom: 16 additional bytes just before Reserved2

The Reserved1 and Reserved2 areas are in case future enhancements might need this space. The Headroom area is a cushion to insure that no tag overruns occur.

With these areas in place, the amount of storage that the tags themselves occupy is computed as follows:

Tag storage in a sector = (sector size) - 48

## Tag and SectorTerminology

The following terms are used throughout this design doc and the code itself:

fresh

A byte in a flash tagging sector is *fresh* if its value is 0xFF. This is the default value after a sector erasure.

unfresh

Likewise, a byte that is *not* 0xFF is *unfresh.* An unfresh byte can have a value from 0x00 to 0xFE.

dirty tag

A tag that has been superseded by a newer version of the same tag ID is a dirty tag. Also, an insane tag can be marked as dirty also.

sane tag

A tag is *sane* if all its header fields are within the allowable ranges, and they all pass sanity tests.

insane tag

A sector that fails sanity tests is *insane;* likewise, a tag can be marked insane. There are different levels of insane tags: recoverable and unrecoverable. An unrecoverable tag is corrupted to the point where it cannot be safely closed down and marked as dirty/insane.

## Tags

A *tag* is the fundamental storage element in the tagging system. Client applications store non-volatile data in tags. The purpose of the flash tagging system is to abstract from such clients the undesirable aspects of flash memory storage, allowing flexible and efficient reading and writing of non-volatile data.

The tags have the following features:

* A tag can be from length zero to length N, where N is limited by sector size and by practical write time limitations. Also, since a tag’s length field is 2 bytes, length cannot be greater than 65535.
* A RAM array (called Tag Pointers) is maintained at all times. This array contains a pointer to the most recent version of a tag. These pointers allow for quick retrieval of tag data.
* If a new version of a tag is written, the tagging system enforces a discrete transition from the older version of the tag to the newer version; a tag is not promoted to a newer version until that newer version has been completely written.
* If device power is lost while a tag is partly written, the tag system will recover from this without loss of data.
* Tag lengths are determined when a tag write occurs (i.e. at run-time), so that a tag can be resized on-the-fly.
* Zero length tags are permitted.

## Tag Offsets

The below diagram shows how tags are laid. The offsets are increasing memory locations.

Magic No. + Status

Tag ID

(2 bytes)

Length

(2 bytes)

Version

(2 bytes)

(Reserved)

Data

0, 1

Offsets

6

4

8

12

2

Each tag header is 12 bytes. The last 4 bytes are reserved, so currently only 8 bytes are used. The tag’s data starts immediately after the last reserved byte, extending to howsoever far the length deems. Since a tag must be aligned on a word (4-byte) boundary, the tag’s data is always aligned on a word boundary. This means that structures can be overlaid into tags without violating any alignment restrictions.

## Tag Header Field Descriptions

Note that all 2-byte fields (Tag ID, version, length) are big-endian. Also, these 2-byte fields purposely don’t use 0x0000 or 0xFFFF as a safety mechanism.

Magic Number

The magic number is one byte located at offset 0. It is a fixed value of 0xA5. This value is used to reinforce tag sanity assurance.

Status

This is a single byte at offset 1. Each bit in the status byte represents a single status boolean. Since a flash device defaults to 1 when erased, and is changed to 0 when written, each status bit can only be changed from a 1 to a 0, but not from a 0 to a 1.

Here are the following status bits. A value of 0 means that the field is active.

*Bit Number Description*

0 Header Written

1 Data Written

2 Dirty

3 Insane

*Header Written—*All fields in the header (except for the status bits, of course) have been written.

*Data Written—*All the bytes in the tag’s data string have been written.

*Dirty—*The tag is marked as obsolete, in normal cases having been replaced by a newer version of the same tag

*Insane—*The tag is marked as insane if it fails a sanity check on power up. The probable cause of an insane tag is that the tag was only partially written when an outage occurred. On outage recover (warm restart), the flash tag initialization detected that the tag write never completed, and marks the tag *insane.*

Tag ID

A 2-byte value identifying the tag. The range of legal tag ID’s, those that can be used by clients, is 1 to 65533 inclusive. The value 655534 is used internally to forcibly denote an insane tag.

Version

Each write of the tag increments the version number by one. Like the tag ID, the legitimate version numbers are 1 to 65533 inclusive, with 65534 as the insane version. The first time a tag is written, its version will be 1. The next time it’s written it will be 2, etc. Version numbering wraps at 65533, so when a tag of version 65533 is rewritten, the next version will be 1.

Length

A value from 0 to 65535. This is the length of the data portion of the tag—length does not include header length.

Data

The tag’s data

## Tag Writing Sequence

Rather than doing tag writes in a single driver flash write call, a tag is written in a series of steps. The reason for doing this is to insure that the progress of the tag write can be ascertained should an outage occur at any point.

1) The tag’s header, not including status bits is written

2) The Header Written status bit is then written to indicate that the header write is complete

3) Next, the tag’s data is written

4) The tag’s Data Written status is then written

## Tag Rewrite Steps

In the case where a tag already exists, writing a new one involves the following steps:

1) An algorithm is run to determine what sector the tag is to be written to. Naturally, the sector must have enough room (sector headroom and sector reserved space included) for the tag, and the sector cannot be in the process of being reclaimed or being erased.

2) The offset into the sector where the write will occur is determined. The tag is always written at the highest memory location, after the last tag written tag to that sector. The offset is always word (4-byte) aligned, so there will be 0 to 3 bytes of unused storage between the last tag and the current tag.

3) The tag header info is calculated. If an old version of the tag exists, the version number from the old tag is incremented (with a wrap), and this incremented version is applied to the new tag.

3) The tag is written according to the steps in the previous subsection.

4) The tag pointer is updated to point to the newly updated tag.

5) The old version of the tag has its Dirty status bit changed, marking the old tag as dirty.

6) The sector and space statistics for both the new tag’s sector and the old tag’s sector are updated.

## Garbage Collection Steps

An advantage of a flash tagging system is that flash erases are decoupled from flash writes—the writes and erases are independent of one another. *Garbage collection* is the process of creating more free space for writes to occur. Without garbage collection, progressive tag writes would eventually deplete a tag space of all free space, causing writes to fail.

Garbage collection is broken down into the following sequential steps:

A) A sector reclaim scoring algorithm.

B) Sector reclaim

Sector reclaims are further divided into the following sequential steps:

A) Sector abandoning

B) Sector erasure

C) Final cleanup after erase

Sector Scoring Algorithm

The garbage collection routine runs the space’s sector statistics through an algorithm to determine if the tag space needs more free space, or if there’s a single sector which simply has too many dirty tags. In the case where the tag space in general is low on free space, the scoring algorithm must select one sector to be reclaimed. Of course, these algorithms assume that amount of data being stored by client applications is within the capacity of the tag space to not only store this data, but to manage garbage collection on it.

Sector Reclaim

A *sector reclaim* is the means that dirty tags are eliminated, thus creating free space.

Sector Abandon

A *sector abandon* is part of a sector reclaim. Sector abandoning walks through all the tags in a sector. If a tag is the latest version, it must be moved to another sector—else the erase op will permanently destroy it. The tag move is accomplished the same way a tag rewrite with the same tag data would occur: the tag’s data is copied over to the new tag in the new sector, and the version number is incremented, the tag pointer is updated, and the old tag is marked dirty. After all the clean tags in the sector being abandoned have been moved, all that is left is dirty tags. The sector is then ready to be erased.

## Reclaim Scoring Algorithm

There are a few scoring algorithms for sector reclaim, but one is the primary algorithm. The primary algorithm works in the following way:

* The total storage capacity of the tag space is calculated. This total is equivalent to the amount of free space of a completely empty tag system.
* The tag space’s current free space, current clean space, and current unclean (dirty or insane) space are calculated.
* The values are normalized—assigned to a value between 0 and 1000.
* The sector with the greatest amount of unclean tag data is ascertained.
* A sliding trigger threshold is calculated, so that the trigger point decreases with a decrease in free space.
* If the total amount of unclean space exceeds this sliding trigger threshold, then the sector containing the most unclean bytes is reclaimed.
* Alternately, if any single sector has acquired too much unclean data, it is reclaimed.

# API’s

For a thorough description of API’s, see the source code. All API’s begin with the upper-case letters FT. The list of API’s is given here for reference sake:

client APIs

These are the only API’s needed by the client applications.

*NVMinit*

*NVMreadTag*

*NVMwriteTag*

Supporting Subsystem API’s

*NVMgarbageCollectNoErase*

*NVMeraseIfNeeded*

*NVMeraseSectorForeground*

*NVMeraseSectorBackground*

maintenance APIs

These APIs are used for debug purposes mostly. These are intended to be tied to table reads or writes.

*NVMtotalReset*

*NVMlatestTagInfo*

*NVMsanityCheckSector*

*NVMfetchSectorStatus*

*NVMnVersions*

# File Structure

It is best to architect any NV firmware in layers, restricting the functionality of each layer. Here is the structure of the NV code for the disco board project. Listed are the layers from top to bottom:

* *Serialization Layer*. (Hasn’t been coded yet.) Has mutex guarding access to all APIs of the bottom layer, allowing tasks to access the NV subsystem one at a time. This could also be implemented as a task which takes requests.
* *Tag Management Layer*. Manages 1 to N tag spaces. Code is currently in the files *nvm-tag.c* and *nvm-tag.h*. Presents *NVMxxx()* APIs to the Serialization Layer. It is also possible to have tasks call the *NVMxxx* APIs directly, bypassing the Serialization Layer—so long as the contract is honored whereby the *NVMxxx* APIs are not reentrant: only one task at a time can call these.
* *NVM Platform Layer*. This layer defines the tag spaces. A tag space specifies a list of contiguous sectors to be used together with a set of tag values. The addresses and sizes of these sectors is contained in this layer. The NVM Platform Layer also presents the FLASH read, write, and erase functions to the tag manager. Files for this layer are *nvm-platform.h, nvm-desc.h, nvm-\*.c* (like *nvm-disco.c* for the disco project).
* *FLASH Driver*. This layer may or may not be necessary, depending on the difficulty of writing or erasing to the particular device, or depending on the need to have other services directly access FLASH programming. The NVM Platform Layer could skip using a FLASH driver and write to FLASH directly. In the disco project, this driver consists of the files *nvm-stm32f4xx.c, nvm-stm32f4xx.h.*

# Performance Considerations

## Tag Space Sizing and Wear Life

As a rule of thumb, I recommend that a tag space’s capacity be 20% or more greater than the maximum amount of tag storage the clients might use. When one analyzes the tag performance, one discovers that as the excess capacity of a tag space is cut back, the amount of “thrashing” occurs—*thrashing* meaning excessive and wasteful sector reclaims. Unlike the serial data flash, when a flash tag sector is reclaimed prior to an erasure, it must find a new home for all the valid data still residing in the sector to be reclaimed. These valid tags are copied over to another sector. Generally speaking, the more writing done, the more erasing that will need to be done as well. The moving of clean, valid tags is a waste. As a tag system fills up, the percentage of dirty tags in a given sector to reclaimed gets lower and lower, as the tag system frantically tries to ensure that there’s enough room for a worst-case burst write scenario—enough room, that is, in the quiescent config, without a reclaim, and hence erase, to create more free space. The near-full config, the one that forces reclaims that have a small percentage of dirty space, are not efficient and therefore cause excessive wear on the flash device.

Wear life can be extended by oversizing the tag spaces. The ability to oversize a tag space is an option that a tag system holds over a data flash/direct mapped solution. To extend wear life in a tagging system, one may simply throw more resources at the problem—in other words, oversizing a tag space trades off flash utilization efficiency for wear life.

Yet another factor to consider is that a tagging system will distribute erases fairly evenly over all sectors in the space. In a data flash/direct mapping solution, a “favorite” file, one which is rewritten at a high rate, will wear out the sector that it’s mapped to, causing a hardware failure. This favorite file might be as small as one sector (256 bytes of data). All such direct mapping systems needs is for one sector---even if it’s a small percentage of the overall size of the meter—to fail, and the whole meter fails.

## Tag Write Times

Since tag writes take a few cycles, a single tag write consists of the following operations:

* One write of 8 bytes (header)
* One write of 1 byte (header write status)
* One write of 1 to N bytes (data)
* One write of 1 byte (data written status)
* One write of 1 byte (old tag marked invalid)

The ST Micro STM32f4xx flash part allows writes in different sizes, according to voltage: write 1 byte at a time, 2 bytes, 4 bytes, 8 bytes. The write times are as follows:

*Mode Max Bytes Written Time*

Byte Write 1 16 microseconds (typical)

Half Word Write 2 16 microseconds

Word Write 4 16 microseconds

Double Word Write 8 16 microseconds

Applying these modes and times to a typical tag write, it will take the following number of write operations:

3 writes (3 status bytes)

4 word writes (header)

Any number of writes (data)

So the tag writing overhead (the time required to write everything *except* for the tag data itself) 112 microseconds (ST in voltage 4 config). Add to that the amount of time it takes to write the data portion of the tag. For a large tag, this is at best 2 microseconds/byte. For a small tag, this is 64 microseconds. So, the total time to write a tag is 176 microseconds and up. This is based on flash write times only—all other times are excluded.

## Tag Read Times

Pretty much goes directly to memory. Depending if tagging system is wrapped in a task or not. But the biggest factor is whether there is a write in progress. If a write is in progress, a read will have to wait before it can occur.

## Sector Erase Times

This is dependent on the manufacturer’s part. Also, times are proportional to the size of the sector. An STM32f4xx part features a typical erase time of a 16kbyte sector of 250millisecs.

**Footnotes**

1Tests from an ARM7 based system (20MHz, thumb mode, 1 wait-state 16-bit data bus) with approx. 250 tags, and 8 x 8kbyte sectors, with sectors almost filled: FLASH tag storage initialization took < 50 milliseconds