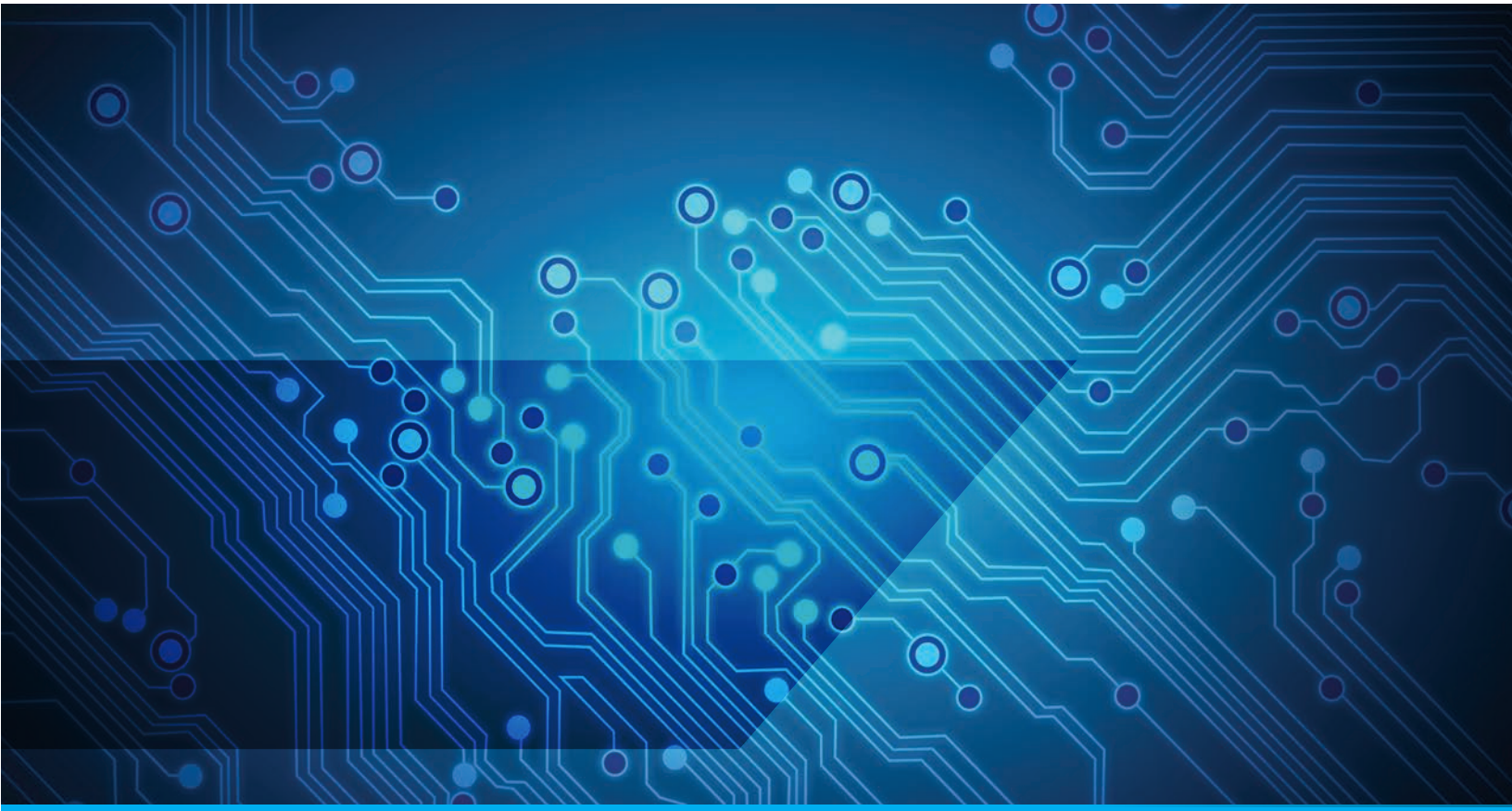
ADVANCED MRAM TECHNOLOGY

MRAM CAN BE >10,000X MORE WRITE-ENERGY EFFICIENT THAN FLASH



SPIN MEMORY™

MRAM CAN BE >10,000X MORE WRITE-ENERGY EFFICIENT THAN FLASH

This is a time of exciting change in the non-volatile memory (NVM) industry. Long dominated by floating-gate technology (flash or EEPROMs), scaling challenges have encouraged broad innovation into new ways of storing data.

As these new technologies evolve, energy consumption has emerged as a major constraint for systems-on-chip (SoCs) or other application-specific ICs (ASICs) on which embedded NVM will play a key role. Many of these chips will serve as the brains of small battery-powered devices, such as those in the evolving Internet of Things (IoT). Any new technology that fails the energy frugality test will not be taken seriously for this important industry growth segment.

Embedded NOR flash is the incumbent technology for these energy-constrained devices. Unlike NAND flash (found in thumb drives and solid-state drives), NOR flash is typically used for storing code and small bits of data. Its read interface makes it suitable for execute-in-place operation, where a processor reads program instructions directly from the flash, without storing them first in local SRAM. This simplifies the system architecture while also allowing for nominal amounts of data storage.

A leading NVM contender for these applications is spin-transfer torque Magnetic RAM (ST-MRAM, or simply MRAM), which promises scalability to the most aggressive silicon nodes – while embedded flash struggles to get below the 40-nm node. In addition, it has a simple read/write interface with byte addressability. This eliminates an extraordinary amount of overhead and complexity.

OF THE ADVANTAGES THAT MRAM HAS OVER FLASH, NONE IS AS STRIKING AS THE SAVINGS IN THE AMOUNT OF ENERGY REQUIRED TO WRITE DATA INTO THE MEMORY. THIS ARISES MOST DRAMATICALLY FROM THE LONG TIMES REQUIRED TO WRITE AND ERASE A FLASH DEVICE. THE FOLLOWING ANALYSIS WILL SHOW THAT MRAM PROVIDES THE OPPORTUNITY TO REDUCE WRITE-ENERGY CONSUMPTION BY 3 OR 4 ORDERS OF MAGNITUDE.

WRITING TO MRAM VS. FLASH

Flash technology depends on floating gates to store a state. Writing involves moving charge across a dielectric barrier onto the floating gate; erasing means moving the charge back out. In the most abstract sense, “writing” and “erasing” should be about writing 1s and 0s, but the nature of flash technology makes it much more difficult than this.

With SRAM or DRAM, when you store a variable, you pick a memory location, and that location remains fixed. Changes to the data happen at that location, with new values written over old values. But this isn’t possible with flash. Flash requires that new data be written only over fresh, erased cells. This alone suggests that, rather than simply writing a new value, one would need to first erase and then write.

But there’s a further limitation: flash can be erased only in sectors (or even larger blocks). So it’s impossible to selectively erase only those cells that are to be rewritten. This places a heavy management burden on the system.

MRAM, by contrast, relies on a magnetic element to store values. Like an SRAM, MRAM allows Write in Place (WiP) and can be addressed byte-by-byte for both reading and writing. By simply writing 1s and 0s, existing cells can be overwritten without the need for prior erasure. Furthermore, the time required to program an MRAM cell is much shorter: less than 100 ns, as compared to a typical flash write time of 40 µs and erase time of 50 ms.

In order to illustrate the practical implications of these differences, we will examine three different use cases. Exactly how they're implemented are, of course, subject to system design considerations, so there will be multiple options. We will review a few strategies, but it's not expected that other alternatives will result in a material difference in the ultimate conclusions. The three use cases we'll examine will be:

1. **Code updates: how much energy is required to do a code update?**
2. **Variable storage: how much energy per day is used to manage variables?**
3. **Array storage: how much energy per day is used to manage array storage?**

We will outline the challenges, handling each in isolation. A particular system may need to use two or three of these use cases in a single memory; we won't examine those combinations, since they can interact in complex ways that further increase the latency of a NOR flash device.

Note that there is more to an energy comparison than total energy consumed, which impacts battery life. For example, peak power consumption can be a critical parameter for embedded flash devices. This is the maximum current the device will draw from its battery. That current may "spike" during erasure, exceeding the maximum output capability of the power supply – which could damage it or cause it to shut down. While MRAM also has benefits here, this paper will focus exclusively on total energy consumed.

Specific numbers will depend, of course, on which technology is used. For our calculations, we will use the following numbers as representative of realistic flash and MRAM offerings.

Results will be rounded to three significant digits. Each results table will show individual flash and MRAM results and then a ratio of the flash result to the MRAM result.

For those readers familiar with flash implementations, some of this may be a review. So we've tried to make it clear which portions explain background for those unfamiliar with the details and which portions provide the comparison results so that you can skip explanations that tread familiar ground.

Parameter (UNIT)	Flash	MRAM
VDD (operating) (V)	1.8	1.2
VDD (during write; VW) (V)	1.8	1.2
VDD (during erase; VE) (V)	1.8	1.2
Write current (IW; mA)	20	6
Write time (tW; µs)	40	0.1
Erase current (IE; mA)	20	NA
Erase time (tE; µs)	50,000	NA
Write command overhead (cycles)	3	1
Erase command overhead (cycles)	6	NA
Read time (µs)	0.07	*
Read current (mA)	15	*
Bytes per read or write	4	4
Operating frequency (MHz)	10	10
Sector size (Kb)	2	NA

* NOT NEEDED FOR EXAMPLES

BASIC WRITE ENERGY

While the selected scenarios reflect the complexity that a flash solution requires, the seeds of this massive difference in energy usage can be seen by examining a single write operation or a single erase operation. The scale of the energy difference comes from three simple parameters: write voltage, write current, and write time. Flash has three additional versions of these parameters for erasure. The product of the first two gives power, or the rate of energy consumption; power multiplied by time gives the energy consumed. This is summarized in the following equations, where a W subscript refers to a write operation, and an E subscript corresponds to an erase operation.

$$E_W = V_W I_W t_W$$
$$E_E = V_E I_E t_E$$

We can use the numbers in the table below to find the energy used in a single write or erase operation.

Operation	Flash energy (nJ)	MRAM energy (nJ)	Ratio (flash/MRAM)
Write	1440	0.72	2,010
Erase	1,800,000	NA	NA

Note that, for our purposes, an erase operation is not the same as writing a 0. As we'll see in the examples, a write operation may be accompanied by an erase operation, but not always. In the aggregate, however, erasure dominates the energy picture.

UPDATING CODE

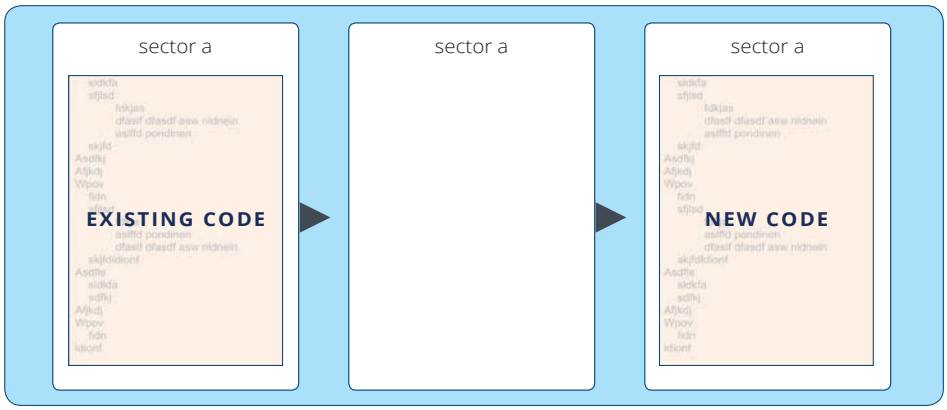
Code execution is, for the most part, an exercise in reading the memory. Any incremental change in the code will be extremely unlikely; much more likely will be a complete update of the entire code image (even if much of it remains the same as before).

MRAM DISCUSSION

With an MRAM, one can simply overwrite the existing code – or write new code at any new location if there's a pointer to relocatable code. No specific sector allocation or erasure is required.

FLASH DISCUSSION

With flash, the new code must be written into erased sectors. While it's possible to erase the existing code segments first, following that up by write operations, a more reliable approach, if enough sectors are available, is to write the new code into new sectors, erasing the old sectors only after a successful write has been confirmed. Then the flash will always have at least one set of good code in it.



Regardless of strategy, from an energy standpoint, this involves erasing some number of sectors (before or after the code write) and writing the new code. How many sectors are involved will depend on how big the c ode image is.

RESULTS

The following table illustrates the difference in code update energy for a variety of code image footprints.

Code size (kb)	Flash energy (nJ)	MRAM energy (nJ)	Ratio (flash/MRAM)
1	2,170,000	338	6,420
3	4,710,000	1,010	4,650
5	7,250,000	1,690	4,290

MANAGING VARIABLES

Another use case deals with managing variables in non-volatile storage. There are two obvious situations here:

- Configuration variables that are set up at installation; they are likely to be read at power-up or at other times during execution, but are not likely to be rewritten. We refer to these as static variables.
- Other variables that need to be persistent when the power goes off. These are likely to be rewritten; the write frequency may vary widely and will have a strong influence on the energy consumed in a given period of time. We refer to these as dynamic variables.

MRAM DISCUSSION

Like an SRAM, MRAM can write and overwrite variables with no special management.

FLASH DISCUSSION

The typical approach to variable management in NOR flash is to build a table. When a variable changes value, you can't rewrite the old location, so instead, you write a new location and invalidate the old one. This table of entries (some current, some invalid) will grow with each change until its sector is full.

The program keeps track of the variables by maintaining pointers to the current version of the variable, updating those pointers when the value – and location – changes. This means that more than just the variable must be stored: an entire record, including metadata, must be stored as a table entry. The way we've modeled it is to have four fields in each table entry:

- **A NAME FIELD**
  Used when powering up so that the program can scan the table and identify the current versions of each variable. We've assumed 16 bytes for this (as it's likely to be text)
- **A DATA LENGTH FIELD**
  If all the data has the same length, then this isn't necessary. A more general solution would require it, however; we've assumed one byte for this (that is, a data value using up to 256 bytes)
- **THE VARIABLE VALUE**
  While the data length field allows up to 256 bytes, we've assumed 4 bytes.
- **A CURRENT/INVALID FIELD**

That last field is tricky. The challenge is that you want to write a table entry and have it be current; some time in the future you need to be able to mark it as invalid. Since you can't overwrite existing data, then there's really only one way to do this: the "current" value must be the fully erased value, and the length of the field must be the minimum writable unit (4 bytes in our example). This allows you to come back and write a new value over it later, when marking it as invalid.

As you update variables, you continue writing new table entries until you run out of room in the sector, at which point you need to perform something akin to garbage collection. The process involves refreshing the current values in a new sector, eliminating all the old invalid entries. We've modeled three different strategies for handling this process to evaluate their impact on energy consumption. As it turns out, the energy is dominated by the amount of energy required to erase old sectors, so the three strategies end up not differing greatly. We include them, however, to show that our conclusions are not the result of a select favorable configuration.

SINGLE-SECTOR

This strategy applies if you have a limited number of sectors available for variables. The approach here is that, once you've outgrown the sector, you read all the current variables into working memory (presumably SRAM), erase the sector, and then rewrite new current entries. The steps are:

1. **Read all variables (static and dynamic)**
2. **Erase the sector**
3. **Rewrite all variables (static and dynamic)**

PING-PONG EAGER

This strategy (and the next one) can be used when you have a spare sector available. It allows you to write the refreshed entries before you erase the old sector, increasing reliability. We have modeled a couple ways of handling this, however. In the "eager" case, we do the garbage collection as soon as we spill over into the new sector. The steps are:

1. **Read all current entries except the one that we just wrote when starting the new sector**
2. **Write new entries for those variables into the new sector**
3. **Erase the old sector**

PING-PONG LAZY

This approach is like the prior one, except that you save some effort by waiting until all the dynamic variables have entries in the new sector as a result of normal operation. This means that you need to copy over only the static variables during clean up. It also assumes that all variables will be updated at least once before you overrun the new sector. We modeled all variables as having the same update frequency, so it works for our examples; other examples might not be able to use this strategy. The operations here are:

1. **Read the static variables**
2. **Write new entries for the static variables into the new sector**
3. **Erase the old sector**

RESULTS

The following table shows the impact of these scenarios. All of them assume 5 static variables. The ratios for the various scenarios are close enough that we just list one average value.

Dynamic Variables	Write Frequency (per day)	Flash Energy (nJ/day)			MRAM Energy (nJ/day)	Avg. Ratio (flash/MRAM)
		Single-Sector	Ping-pong Eager	Ping-pong Lazy		
20	100	93,600,000	93,300,000	86,400,000	2640	34,500
5	100	18,500,000	18,400,000	18,100,000	660	27,800
5	20	3,690,000	3,680,000	3,620,000	132	27,800
5	1000	185,000,000	184,000,000	181,000,000	6600	27,800

MANAGING VARIABLES IN NON-VOLATILE STORAGE

MANAGING ARRAYS

Arrays are multi-valued variables. How they are handled may depend on their size, however. They are another example of a data type that is trivial to manage in MRAM, but can be quite complex in NOR flash.

MRAM DISCUSSION

In MRAM, arrays require no management. Each cell can have any value written, regardless of array size or cell size or type.

FLASH DISCUSSION

With NOR flash, if you have a small array, you can manage it just like you do variables, by using table entries. This works, in theory, as long as your array size (including table metadata) is less than half the size of the sector. In practice, efficiency considerations would probably suggest a smaller array than this.

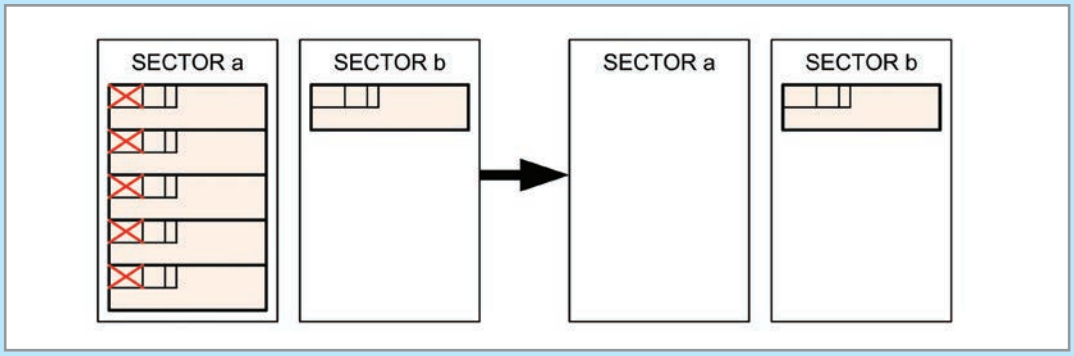
For larger arrays, you need to dedicate one or more sectors to the array. With each update, you would rewrite the array into fresh sectors. There are different ways to manage this, depending on how many sectors you have available:

- You could work with the minimum number of sectors needed to store the array. When updating, you'd need to erase each sector and write in the new values, sector-by-sector.
- You could allocate one spare sector. You would then update the first sector, writing into the extra sector, and then erasing the old version of that first sector. The just-erased sector would then be used to update the second sector's data, after which the old second sector would be erased. This allows write-before-erase for all sectors for better reliability, at the cost of one extra sector.
- If you have lots of sectors available, you could rewrite the entire array before erasing the old sectors.

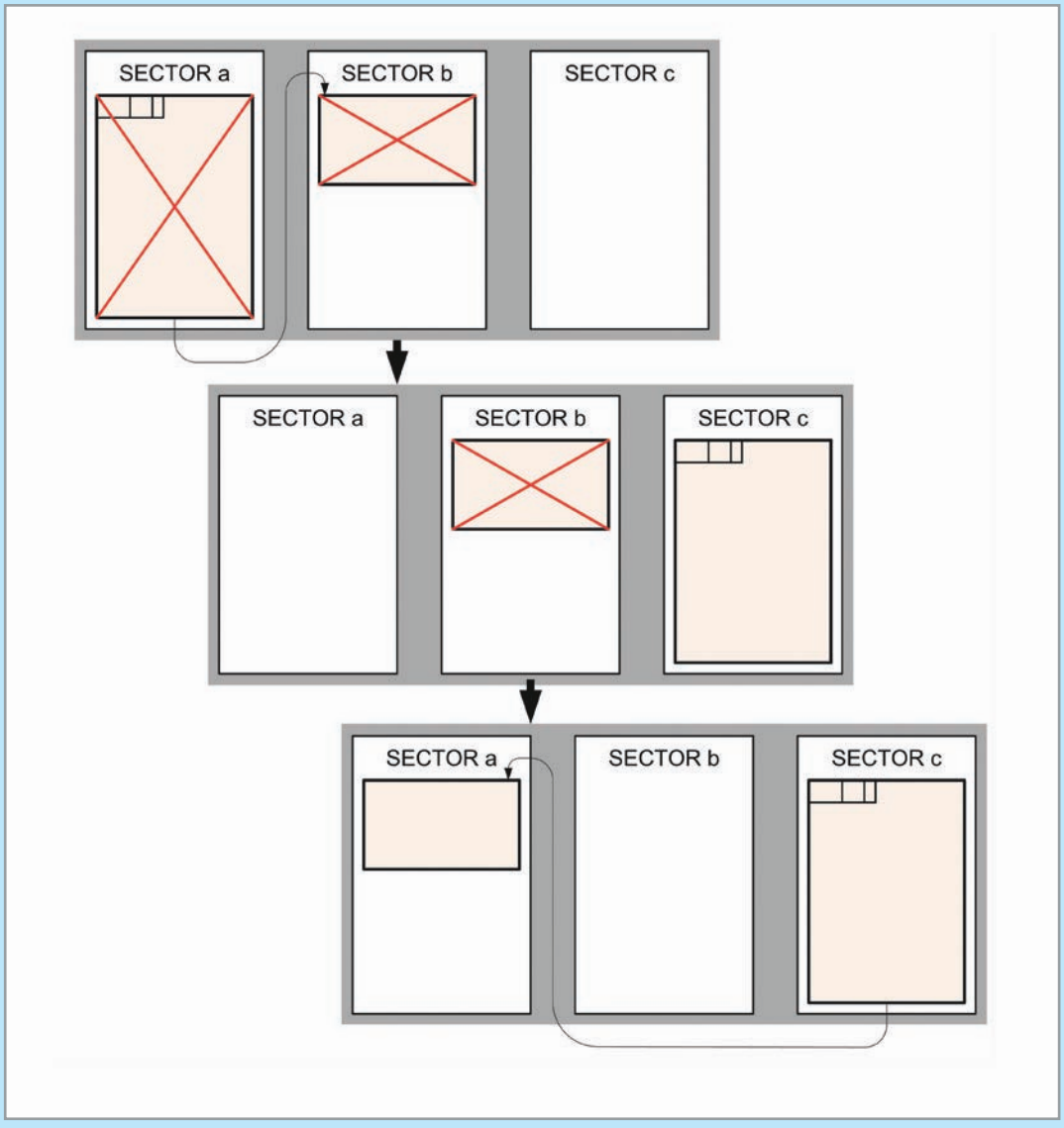
From an energy standpoint, these all involve the same number of writes and erases – just in a different order. So they're equivalent consumers of energy.

MANAGING ARRAYS IN NON-VOLATILE STORAGE

TABLE MODE



LARGE ARRAY MODE



RESULTS

The consumption for these strategies, with varying array sizes and update frequencies, is shown below. Note that array size refers to the number of bytes in the array (not including table metadata). A 1000-byte array could mean a 500-entry array with 2-byte cells or a 50-entry array with 20-byte cells; from an energy standpoint, only the total size matters.

Array Size (bytes)	Update Frequency (per day)	Flash Energy (nJ/day)		MRAM Energy (nJ/day)	Ratio (flash/MRAM)
		As Small Array (using table)	As Large Array		
5000	100	NA	721,000,000	165,000	4,370
100	100	16,000,000	184,000,000	3,300	4,860/55,600
1000	100	NA	216,000,000	33,000	6,550
1000	20	NA	43,200,000	6,600	6,550
1000	1000	NA	2,160,000,000	330,000	6,550

This shows that it's much more efficient to use table updates for small arrays, as you require fewer erase operations.

CONCLUSIONS

These examples illustrate just how much more energy flash uses for writing than MRAM does – by a factor of thousands or tens of thousands. In addition, MRAM can simply be treated as if it were SRAM, while flash requires complex management to deal with erasure and its associated sector requirements. It's no accident that most of this paper is dedicated to explaining flash management.

LOOKING BEYOND WRITE ENERGY, MRAM PROVIDES ADDITIONAL ADVANTAGES, INCLUDING:

- Low write latency and a mixed read/write command stream, allowing execute-in-place with minimal hesitation when writing data
- No theoretical scaling limit
- Better compatibility with CMOS for better cost
- Lack of charge pumps and other flash overhead, again for better cost
- Staggered-write feature for reduced peak current

IN SUMMARY, MRAM REQUIRES MUCH LESS ENERGY FOR A BASIC WRITE OPERATION, BY A FACTOR OF ABOUT 2000. ACCOUNTING FOR THE COMPLEXITY REQUIRED TO MANAGE WRITE OPERATIONS IN FLASH DEVICES INCREASES THAT MARGIN TO AS MUCH AS 50,000.

This makes embedded MRAM a compelling alternative to embedded flash in applications that suffer from the energy and complexity burdens that flash brings with it. It also opens up new applications that can't be addressed by flash for energy reasons – particularly those that involve frequent write operations, like battery-powered remote field data logging. As MRAM reaches commercial production, we expect to see it displacing flash in a broad range of

FOR MORE INFORMATION ABOUT HOW MRAM CAN SIMPLIFY YOUR SoC DESIGN AND SAVE POWER, CONTACT SPIN MEMORY

45500 Northport Loop West
Fremont, California 94538
(510) 933-8200 main
(510) 933-8201 fax
Email: info@spinmemory.com

www.spinmemory.com