Johan Hopland Lofstad

# Alternative error prevention scheme for non-volatile memories

Master's thesis in Cybernetics and Robotics
Supervisor: Geir Mathisen & Rainer Herold
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of Science and Technology

MICROCHIP

Johan Hopland Lofstad

# Alternative error prevention scheme for non-volatile memories

Master's thesis in Cybernetics and Robotics
Supervisor: Geir Mathisen & Rainer Herold
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

**NTNU**
**Norges teknisk-naturvitenskapelige universitet**

**Fakultet for informasjonsteknologi, matematikk og elektroteknikk**
**Institutt for teknisk kybernetikk**

# HOVEDOPPGAVE/MASTER THESIS

| | |
|---|---|
| Kandidatens navn: | **Johan Hopland Lofstad** |
| Fag: | **Teknisk kybernetikk** |

Oppgavens tittel (norsk):

Alternativ feilhåndteringsmetode for ikke flyktig dataminne.

-

Oppgavens tittel (engelsk):

*Alternative error prevention scheme for Non-volatile memories*

**Oppgavens tekst:**

*A weakness of Flash (NVM) Memory is the non-zero probability of a memory cell of flipping its logic state due to charge loss over time. Traditionally, to handle these problems, error correction is used, correcting the error after it has occurred. An error* ***prevention*** *scheme aims to detect and prevent these errors before they occur. The aim of this thesis is to explore and possibly develop such an error prevention scheme based on the flash memories cell charge.*

**Scope of thesis:**

- Based on a literature study, describe how Error Correction Codes (ECC) are used today in NVM technologies based on Floating Gate Architecture.
- Based on a literature study, describe the characteristics of flash failure modes.
- Design and implement a simulator of flash memories, specifically emulating different failure modes of single bits.
- Design and implement an error prevention system of flash cells based on their cell charge using the simulator described above. Explore different solutions to the problem.

| | |
|---|---|
| Oppgaven gitt: | 09. January, 2019 |
| Besvarelsen leveres: | 03. June, 2019 |

Utført ved Institutt for Teknisk kybernetikk

Hovedveileder: Rainer Herold

Trondheim, den 09.01.2019

Geir Mathisen
Faglærer

# Abstract

Flash memory is a common memory used in a variety of devices, using an electric charge to store data. This electric charge can leak and cause an error in the form of a bit-flip. Traditionally methods such as Error Correction Codes (ECC) has been used to mitigate these errors.

The following thesis presents an alternative to the conventional ECC method to find and reduce these errors. By measuring, storing, and evaluating the electrical charge over time, leakage can be detected and prevented before the error propagates into a failure.

To develop and test the proposed system, a simulator for flash has been developed. The simulator uses statistical properties collected from previous work to simulate a flash cell distribution with various error modes introduced.

The proposed error prevention system has been developed on the x86 platform as a proof of concept, with a port to AVR32 in mind. The system has been evaluated against the simulator, resulting in a functioning system. The proposed system manages to find, track, and reprogram cells with a leakage trending towards failure.

# Sammendrag

Flashminne er et vanlig dataminne brukt i en rekke enheter som bruker elektrisk ladning til å lagre data. Denne ladningen kan lekke og forårsake en bit-flip feil. Tradisjonelt har metoder som feilkorrigeringskoder (ECC) blitt brukt til å redusere disse feilene.

Følgende avhandling gir et alternativ til den konvensjonelle ECC-metoden for å finne og redusere forekommelsen av disse feilene. Ved å måle, lagre og evaluere den elektriske ladningen over tid kan denne lekkasjen detekteres og forhindres før det forplanter seg til en feil.

For å utvikle og teste det foreslåtte systemet har en simulator for flash blitt utviklet. Simulatoren bruker statiske egenskaper som er samlet fra tidligere arbeider til å simulere en fordeling av flash cellene. Forskjellige feilmoduser er introdusert i distribusjonen.

Det foreslåtte systemet har blitt utviklet mot x86 plattformen som et bevis på konseptet, hvor en port til AVR32 arkitekturen er tatt i betraktning. Systemet har blitt evaluert med simulatoren, med et fungerende system som resultat. Det foreslåtte systemet klarer å finne, følge og reprogrammere celler med en lekkasje som går mot en feil.

# Preface

This thesis is written for the department of cybernetics at NTNU (Norwegian University of Science and Technology) in collaboration with Microchip Technology Inc. The author of this thesis was approached by Microchip with a proposal to investigate erratic and moving bits in flash memories, and how these could be detected and prevented using internal test methods.

A prelude project for this thesis was conducted fall of 2018, exploring the problem of measuring the cell charge of flash memories internally. This thesis is largely based on the work from the prelude project written by the same author.

There are two supervisors for this thesis, Prof. Geir Mathisen from NTNU and Rainer Herold from Microchip. Rainer Herold provided supervision for the contents of the thesis. Rainer provided help with the direction of the thesis, which avenues to explore and general questions regarding non-volatile memory. Geir Mathisen has provided supervision for the structure of the thesis, such as the disposition and general layout.

Sections of the chapters Introduction, Theory and Literature Study has been taken from the prelude project. These sections has been marked. The remaining contents of the thesis is work done by the author during in the period January - June 2019.

Approved: GMathisen

Prof. Geir Mathisen
Supervisor of thesis

Approved: [illegible]

Johan Hopland Lofstad
Author of thesis

# Table of Contents

# List of Tables

# List of Figures

# Abbreviations

| | | |
|---|---|---|
| RNN | = | Recurrent Neural Network |
| ECC | = | Error Correction Codes |
| HW | = | Hardware |
| NVM | = | Non-Volatile Memory |
| MLC | = | Multi Level Cell |
| FGMOS | = | Floating Gate Mosfet |
| BER | = | Bit Error Rate |
| PD | = | Program Disturb |
| mXT | = | MaxTouch® |
| FN Tunneling | = | Fowler Nordheim Tunneling |
| DMA | = | Direct Memory Access |
| FDMA | = | Fast Direct Memory Access |
| IC | = | Integrated Circuit |
| PR | = | Presence Ratio |
| ToL | = | Time Of Life |
| JSON | = | Javascript Object Notation |

Chapter 1

# Introduction

## 1.1 Background

Flash memories are a commonly used type of Non-Volatile Memory[1]. These memories are used in a wide variety of applications, such as Solid State Drives (SSD) for personal computers and holding the program code for microcontrollers.

These memories use an electrical charge to store its logical state, zero or one. Flash memories suffer from a weakness where the logical state of a cell can flip due to anomalous charge loss. This charge loss happens over time due to imperfections in the silicon known as traps. It may take months to years before the charge loss accumulates high enough to cause an error.

Hardware Error Correction Codes (HW ECC) [2] is widely used in various forms to reduce these errors. ECC are numerical codes calculated from the logical state of several cells and can correct multiple errors. These codes are especially common when a flash cell holds more than one bit in a Multi-Level Cell (MLC).

## 1.2 Motivation

HW ECC comes at a significant silicon cost and introduces overhead when reading and writing. A software solution could be explored to reduce this cost. If a software solution were to obtain the same accuracy as the HW solution, error correction codes must be stored on the flash. Thus a partition of the flash must be reserved for storing these codes, as well as introducing CPU overhead larger than the HW ECC solution.

[1]Memory which does not lose data on power loss

[2]ECC implemented with hardware, i.e. with silicon and not software

Instead, an error prevention scheme is proposed. The charge level[3] can be measured and stored. By looking at the trend and value of the stored measurements, it might be possible to predict a cell failure before it happens. Such a solution would require less storage, and the measurement and calculations could take place as the CPU idles.

## 1.3 Limitations

This thesis limits itself to a proof of concept for such a system. It is not implemented on an AVR32-Microcontroller (which is the targeted architecture), but on x86[4]. The x86 platform gives access to more powerful debugging tools, simulators, and faster evaluation of different strategies.

A system which measures the cell current of flash cells has been previously developed in [5], and is therefore not a part of this thesis. Small summaries from [5] are found throughout the text where the relevant background knowledge is needed.

A challenge with writing this text is the abstraction level. As this project is a deeply technical one, the interesting results and problems occur on a particularly low abstraction level, which requires the reader to have certain background knowledge. At the same time, the text should be easily readable to understand what has been done and what the results represent.

Because of this challenge, parts of the text is not required to read if the reader does not desire a deeper understanding. Throughout the text, these ”non-mandatory” sections have been attempted marked, and the reader is free to skip these on his or her leisure.

## 1.4 Disposition

This section gives a quick overview of what the different chapters contain.

**Chapter 2: Theory** introduces the theory needed to understand non-volatile memory, flash memory, ECC codes, and machine learning. This is required reading to understand the problem at hand.

**Chapter 3: Literature Study** explores two failure modes in non-volatile memory, specifically erratic and moving bits. It looks at the current state of error correction codes in NVM and a new machine learning approach which shows promise but has no known wide commercial use at the time of writing.

**Chapter 4: Specification of project** outlines the project specification and acceptance criteria. It presents what is to come in the coming chapters.

[3]The actual electrical charge, not the logical state

[4]Intel x86 is the architecture most modern PCs use

**Chapter 5: Temperature effect on FDMA (Fast Direct Memory Access)** presents an experiment performed on the temperature effect on FDMA. It proposes a hypothesis, presents the experimental setup and methodology before concluding after a discussion.

**Chapter 6: Design of Simulator** introduces how a simulator of the flash is designed. To test the proposed error prevention system, a simulator is required for rapid development. It goes into how the simulation loop functions, which error modes are simulated and how its interface works.

**Chapter 7: Implementation of Simulator** introduces an implementation of the design in Chapter 6. It discusses the simulation loop in detail, how the different types of cells are calculated for the next iteration. Other topics include temperature effect, network interface, and an accompanying command list.

**Chapter 8: Design of Error Prevention System** introduces how the error prevention system is designed. It discusses the main loop of the system, measuring, storing, and reprogramming cells. The chapter ends with a summary from the prelude project on the measurement system and a discussion on storing data points and dealing with temperature.

**Chapter 9: Implementation of Error Prevention System** outlines the implementation of the design in Chapter 8. It includes the different modules and their interaction and which cells are measured, stored, and reprogrammed in detail. It finishes with the storage module and a section on the main loop, putting it all together.

**Chapter 10: Machine Learning Approach** outlines and discusses a machine learning approach. The proposed approach is an alternative to the error prevention system. The chapter presents a starting point for a design with an accompanying discussion.

**Chapter 11: Testing** explains the setup and methodology for the tests performed on the error prevention system and simulator. It presents unit tests and system tests for both systems.

**Chapter 12: Results** presents the results gathered from the chapter on testing. This includes the results of the unit tests, system tests, a performance analysis of the simulator, and accompanying plots.

**Chapter 13: Discussion** discusses the results obtained, and if the acceptance criteria have been met.

The remaining chapters, **Conclusion** and **Further Work** concludes the thesis and proposes further work on the topic.

# Chapter 2

# Theory

This chapter presents theory which might be required to understand the rest of the text. The theory presented is on non-volatile memory, with the floating gate transistor and how it used in a NAND flash configuration and the difference between SLC and MLC technology. An introduction on error correction codes with an emphasis on Hamming codes follows. The chapter finishes with an introduction to machine learning, neural network, and recurrent neural networks.

## 2.1 Non-Volatile Memory

*The following section on Non-Volatile memory is taken from the theory section in [5] with some modifications and additions. The section on MLC and parts on NAND Flash Memory is new.*

Digital memory is often divided into two categories, volatile and non-volatile. Non-volatile memory, in contrast to volatile memory, retains data on power loss. One type of non-volatile memory is called Flash memory[1]. Flash memory is widely used in several applications where high capacity, low cost non-volatile memory is needed, and uses the **floating gate MOSFET** to store data.

### 2.1.1 Floating Gate Transistor

Every bit in a flash memory is a Floating Gate MOSFET, abbreviated to **FGMOS**. An illustration can be seen in Figure 2.1 and consists of three gates, the *control*, *source* and *drain* gate. The structure is similar to a conventional MOSFET which have the same gates.

[1]The name comes from its fast erase, it erases in a ”Flash”

In a conventional MOSFET transistor, as a voltage is applied to the control gate, current is allowed to flow from the source to the drain. This happens as the ”barrier” between the N-Type and P-Type silicon are broken down, allowing electrons to freely move across from source to drain. In a MOSFET transistor the current from source to drain is a function of the voltage on the gate, $I_{sd} = g(V_c)$. In flash memory, every such FGMOS is referred to as a **cell**.

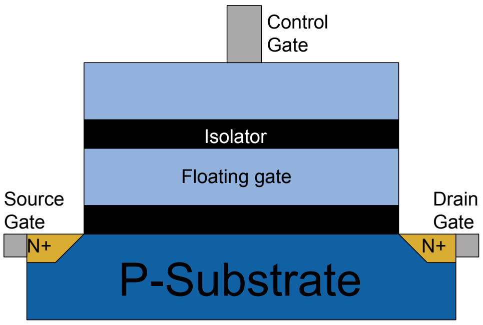


**Figure 2.1:** A Floating Gate MOSFET (FGMOS) Transistor.

In a MOSFET the amount of voltage applied on the control gate when it becomes conductive is called the **threshold voltage**. In other words, the threshold voltage is the minimum voltage needed to make the p-substrate become conductive.

The principal idea in a FGMOS is to store electric charge in the so called ”floating gate”, which sits in the middle of two isolators, i.e. completely electrically isolated. This allows for the storage of data in the form of charge in the floating gate which persists when power is lost. When there is no charge, the cell is labeled as **erased** and has a logical value of *one*. When there is a set amount of charge, the cell is labeled as **programmed** and has a logical value of *zero*.

In order to read a cell (i.e. measure the charge in the floating gate), a voltage is applied on the control gate (as in a MOSFET). However, in contrast to a MOSFET the FGMOS also contains charge in the floating gate ”resisting” the p-substrate to become conductive. In other words, the amount of voltage required to make the transistor conductive is a function of both applied voltage and stored charge in the floating gate $V_{threshold} = g(V_{gate}, Q_{floating\ gate})$ .

A related concept to the threshold voltage is the **cell current**. When applying a known fixed voltage between the source and drain gate, and a known fixed voltage on the control gate (called the **read voltage**), the current between the source and drain is the cell current. As the threshold voltage increases with an increasing $Q_{floating\ gate}$, the transistor becomes more conductive and thus more current flows. In flash memories, the cell current is used to determine if a cell is programmed or erased.

The cell current for a typically erased and programmed cell is shown in Figure 2.2. Both follow the *sigmoid* function, but they are shifted related to each other. The erased cell conducts ”all” the current at the read voltage of zero volts while the programmed cell outputs no current at all. This is the general idea of how a flash cells state is read.

In order for a flash cell to be useful, it must be able to be programmed and erased. The question is how to insert charge in the isolated floating gate. One way of adding and removing charge is **Fowler Nordheim Tunneling (FNT)**, which lets electrons ”tunnel” through isolators. To **erase** a cell and remove the charge, a high voltage is applied between the control gate and source. By FNT, the charge is tunneled through the isolator, removing the charge.

To **program** a cell, a high voltage between the source and drain is applied. A high sufficiently high voltage between the control gate and ground makes the transistor conductive. If these conditions are met, a high current flows from source to drain, making some electrons to jump into the floating gate. This technique is called **hot electron injection**.

As these high voltages and currents damage the transistor, a flash cell has a limited amount of program / erase cycles. After the transistor has been programmed / erased an amount of time, charge starts to leak from the floating gate until it is unable to hold charge for useable amount of time.

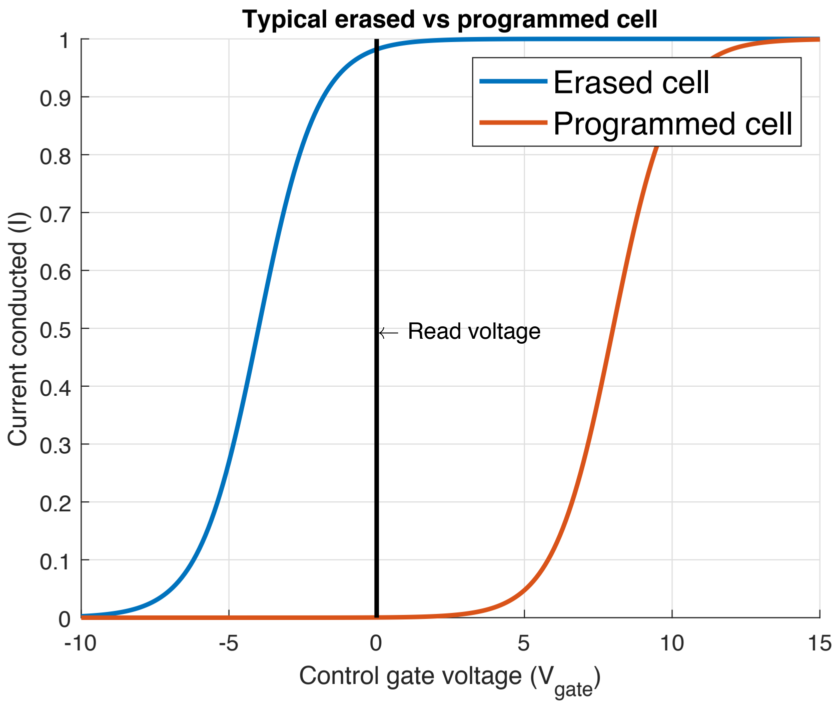


**Figure 2.2:** Illustration of current readout from a flash cell.

### 2.1.2 NAND Flash Memory

NAND (Not-AND) Flash memory is a type of **non-volatile** memory where the data is stored in a hierarchy of floating gate MOSFETs. The term **NAND** refers tosec:temperature-effect-on-fdmaected in series as can be seen in Figure 2.3.

The memory is divided into **blocks** with a number of **pages**. A page is defined as all the bits connected to the same word line. A *block* contains 32 or 64 pages and is the smallest *erasable* unit. A *page* is typically 2 or 4 Kbytes and is the smallest *programmable* unit [12]. When **reading** a NAND flash, the read voltage is applied to the *wordline* of the bit (a word

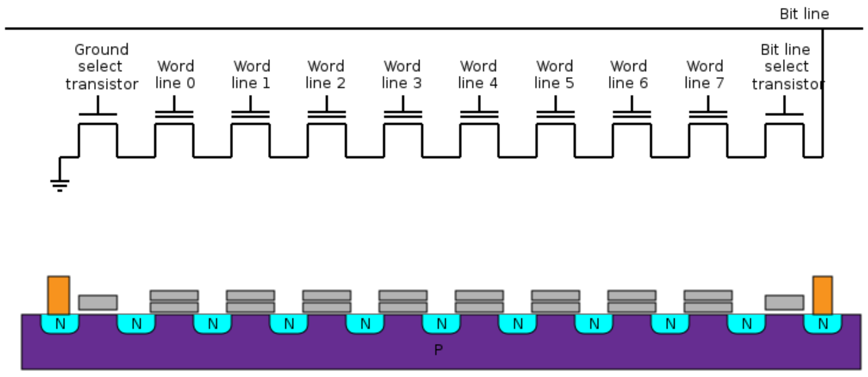


**Figure 2.3:** NAND Flash Configuration [18].

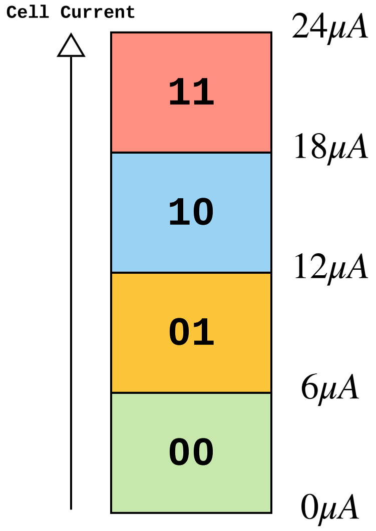
is a collection of bits, usually in the size of 16, 24 or 32 bits, the wordline selects which word is read). The *bit line select transistor* for the desired bit is set high. All the other wordlines in the bitline are set to a high voltage (Figure 2.3 is one bitline). This makes all the other transistors in the bitline conductive no matter the stored charge, as can be seen from the graph in Figure 2.2. The current value of the cell can now be read at the bitline.

Since it is not possible to erase individual bits, a full **erase-write** cycle must be performed when changing a bit. The data already existing on the block must be stored somewhere else, the block must be erased before the altered data can be written back to all the pages in the block. This degrades the oxide layer on the FGMOS, until a point where it can no longer hold the charge on the floating gate. This is why flash memory has a **limited number of writes/erases** before it is no longer usable.

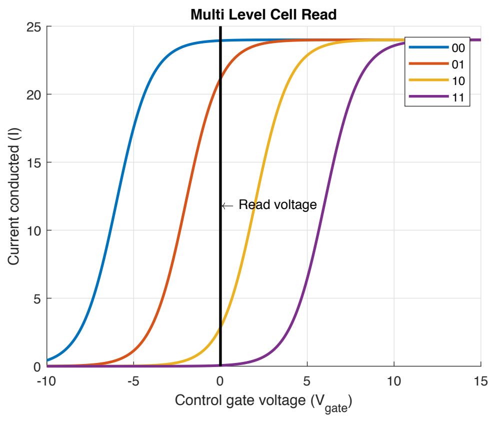
To increase the lifetime of a flash memory, several techniques are applied. By storing **Error Correction Code (ECC)** for each block, the flash controller can detect faulty blocks and mark them as unusable. The flash controller also tries to spread out the writes between all the block, such that no block is written to excessively [12].

### 2.1.3 Multi-Level Cells (MLC)

All the flash cells introduced so far only holds one bit per FGMOS transistor; this is called SLC (Single Level Cell). However, multiple bits are encoded into the same cell to increase the density of flash memories. Such a scheme is called MLC (Multi-Level Cell). Figure 2.4 shows two representations of this encoding. The main difference is that SLC only has two states (0 or 1), while MLC has four or more states (00, 01, 10, 11).



**(a)** Block representation of an MLC cell.



**(b)** Graph representation of an MLC cell.

**Figure 2.4:** Two representation of MLC.

The most significant advantage of MLC compared to SLC is the increased density. An MLC cell can hold quadruple the storage as SLC for the same amount of silicon. The biggest drawback is the endurance, as it is more susceptible to cell current drift. This drift is caused by the state limits being closer together, and as such erratic and moving bits are a much bigger problem. As errors occur more frequently in MLC, more advanced error correction mechanisms are applied compared to SLC[7, 8].

## 2.2 Error Correction Codes (ECC)

Error correction codes (ECC) are numerical codes used to correct errors in data. When the data is ”healthy” (no error has occurred), the respective ECC for that piece of data is calculated and stored. This extra data can then be used to detect and repair errors. These codes are beneficial to mitigate errors in NVM, and is the primary method to handle NVM error modes such as erratic and moving bits.

There are several ECC algorithms which fall into two main categories, **Block ECC**, and **Convolutional ECC**. Block ECC calculates codes for a *block* which is a fixed size of data. Figure 2.5 shows the concept of data blocks with their respective ECC data. Convolutional ECC uses a circular pattern to store information but is not covered in this thesis.

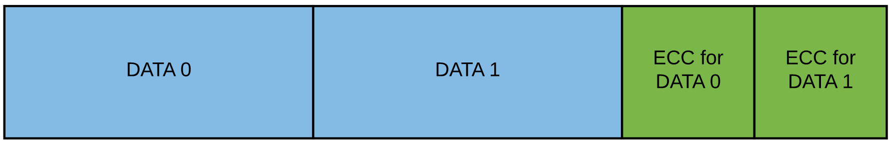


**Figure 2.5:** Illustration of block ECC codes.

The basic concept is illustrated in Figure 2.6. A block of data or *message* (**m**) is encoded into a *codeword* (**c**). A codeword is the data *plus the error correction code*. This codeword is stored in memory, as illustrated in Figure 2.5. As an error occurs, the stored codeword is damaged. When decoding the codeword, this error is corrected as long as the damage is not too great.

A **syndrome** is calculated to detect errors. The syndrome is a vector indicating which bit positions an error has occurred and if it can be salvaged.

### 2.2.1 Hamming codes and parity bits

Hamming codes are a type of **Linear Block Error Correction Codes** first introduced by Richard Hamming, hence the name. Due to their simplicity and effectiveness, Hamming codes are widely used.

#### Parity Bits

Hamming codes uses the concept of *parity bits* to encode the data. A parity bit is an extra bit put onto a block, indicating if there is an even or odd number of one’s. For instance, the 8-bit message $[1, 0, 0, 1, 1, 0, 1, 0]$ would generate a parity bit of zero, as there is an even number of one’s.

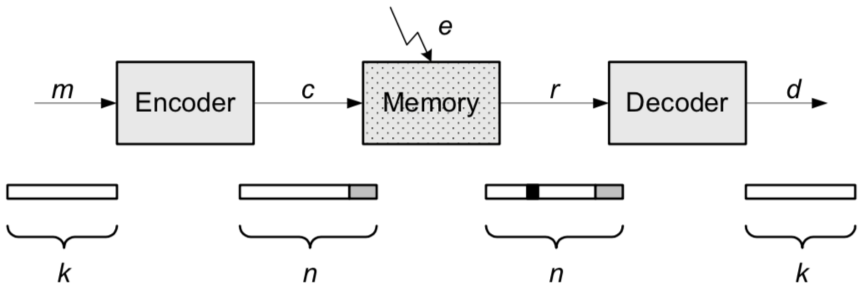


**Figure 2.6:** ECC Flow. A message $m$ of length $k$ is encoded into a codeword $c$ of length $n$. An error $e$ occurs on the codeword $c$ resulting in the damaged codeword $r$. The damaged codedword $r$ is decoded with the error corrected into $d$. Figure taken from [8].

The big drawback of such a parity bit scheme is that it can only detect errors, and not correct them. If a parity bit of one is recorded, and the message is read as $[1, 0, 0, 1, 1, 0, 1, 1]$ (the last bit is corrupted), the scheme would only know that an error has occurred, and not which bit caused the error. That is, it can detect *single bit errors, but correct none*

### Hamming Codes

Hamming codes solve the correction problem of parity bits, by allowing for correction and not just detection. The basic idea is to use more than one parity bit, and compare the results between these parity bits. Hamming codes are best explored through an example.

#### Example: Hamming[7,4] correction

A hamming code with four bits is encoded with three parity bits, creating a seven-bit message called the Hamming[7,4] code. Imagine wanting to encode the message $m$ as shown in Eq 2.1.

$$m = \begin{bmatrix} m_1 & m_2 & m_3 & m_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix} \tag{2.1}$$

There are three parity bits: $p_1$, $p_2$ and $p_3$. These indicate the parity of different sets of data bits. Specifically $p_1$ indicates the parity of $m_1$, $m_2$ and $m_3$. $p_2$ indicates the parity of $m_1$, $m_3$ and $m_4$. $p_3$ indicates the parity of $m_2$, $m_3$ and $m_4$. For instance, $p_1 = 1$ as there is an even number of one's in $m_1$, $m_2$ and $m_3$. This is illustrated as a venn diagram in Figure 2.7.

Imagine an error occurs for $m_2$, giving us $m_c = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}$. When calculating $p$ for $m_c$, one finds that $p_1$ and $p_3$ changes, while $p_2$ stays constant. From this one can conduct that the error is in the intersection between $p_1, p_3$, but not in the intersection $p_1, p_2$ or $p_1, p_3$, which leaves one bit namely $m_2$ as the faulty bit. This is illustrated in Figure 2.8

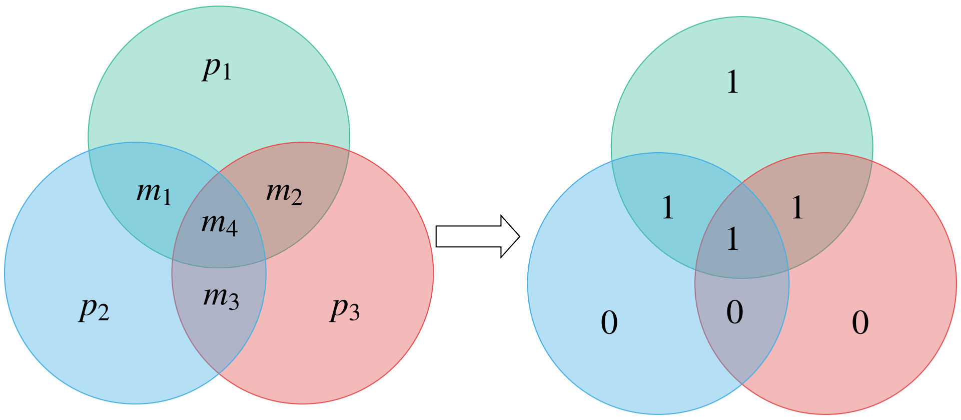


**Figure 2.7:** Hamming[7,4] illustrated as a venn diagram.

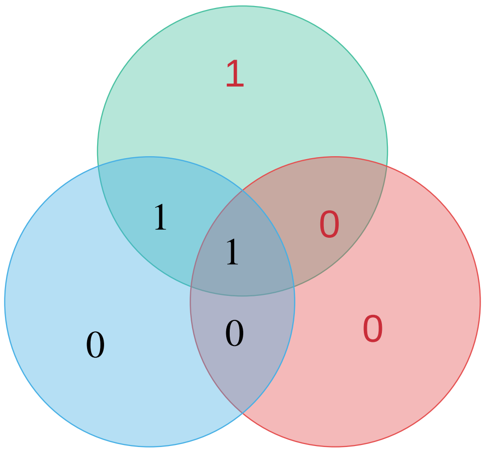


**Figure 2.8:** Hamming[7,4] illustrated as a venn diagram.

### Generating the codeword and syndrome with matrices

As Hamming codes are linear, the encoding process can be done with a simple matrix modulo two multiplication. The codeword $c$ of the message $m$ is calculated in Eq 2.2. The matrix $G$ is called a *Generator Matrix*

$$c = mG \tag{2.2}$$

As described above, the **syndrome** indicates whether or not an error has occurred. For hamming codes, the syndrome $s$ on the corrupted codeword $r$ is also calculated with a simple matrix calculation.

$$s = rH^T \tag{2.3}$$

The $H$ Matrix is called the *Parity Check Matrix*. How to produce the $G$ and $H$ matrices are not discussed here.

**Example: Hamming[7,4] with matrices**

Imagine the same problem as the earlier example, but this time solved using matrices. The message $m = \begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix}$ is corrupted to $m_c = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}$. Two possible forms of the G and H matrix for Hamming[7,4] are given in Eq 2.4.

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} H = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{2.4}$$

Notice the form of $G = [I_k|A]$. The columns in $A$ indicates our parity bits. The first column indicates $m_2$, $m_3$ and $m_4$, the second $m_1$, $m_3$ and $m_4$ and so forth.

**Step 1: Calculate the codeword $c$**

$$\begin{aligned} c &= mG \\ &= \begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 0 & 1 & 2 & 2 & 3 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Remember that these are modulo two multiplication, hence the last equality.

**Step 2: Introduce error and calculate syndrome**

The error ($e$) is introduced into our codeword ($c$).

$$r = c + e = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

As per Eq 2.3, the syndrome $s$ is

$$\begin{aligned} s &= rH^T \\ &= \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 2 & 1 & 2 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} = 2 \end{aligned}$$

The last equality is due to 010 in binary equals two as $H$ was constructed for this property, indicating that the **error is in the second bit position**.

## 2.3 Machine Learning - Neural Networks

Artificial Neural Networks is a type of machine learning that mimics biological neural networks to learn and predict mathematical functions. In essence, given a set of X's and Y's, a neural network tries to predict F as accurately as possible where Eq 2.5 holds [11, 13, 9].

$$Y = F(X) \tag{2.5}$$

### 2.3.1 Neurons

A neural networks is built on **neurons**, which are takes some input X and produces one output, an illustration of which is shown in Figure 2.9. The weights (w's in Figure 2.9) determine how much each input controls the output of the neuron. The function $f$ is called the **activation function** [2], and determines the output Y depending on the input X and weights W.

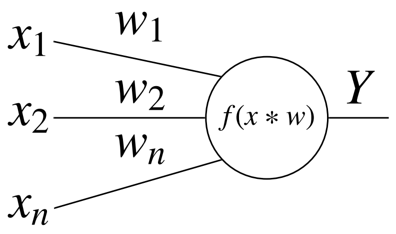


**Figure 2.9:** A neuron with input X, evaluation function f and output Y.

### 2.3.2 Neural Network

A **neural network** is obtained by combining these neurons in a network, connecting the output of some neurons to the input of other neurons. Most often neurons are arranged in *layers* as seen in Figure 2.10, which has three layers (input, hidden and output). One might stumble on the term **deep neural network**, which refers to a neural network with many layers [3], hence the naming *deep*.

The first layer is widely referred to as the **input layer**, which takes the input to the network. The last layer is the **output layer**, which outputs the final result. The network is **trained** by giving it training data in the form of (X, Y) pairs. The error on the output ($e = output\ of\ last\ layer - Y$) is propagated backward in the network to figure out how much each weight $w$ contributed to the error. This is called **backpropagation**. By using gradient descent, these weights are gradually updated with each training example to converge towards the true function $F$.

[2]There exists many activation functions, but a commonly used function is the *sigmoid function*

[3]How many layers for a network to become deep is not defined

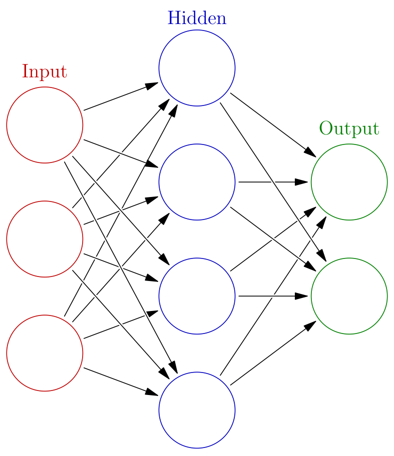


**Figure 2.10:** An illustration of a neural network [17].

### 2.3.3 Recurrent Neural Networks

The neural network described above is a **feed-forward network** (FFN) as the information is fed forward in the network, and nothing feeds backward (all computation goes from left to right). FFN assumes that the decision is independent of the previous decision, which is not always the case. The concept of **Recurrent Neural Networks (RNN)** was introduced to solve this problem.

Imagine the problem of predicting the next character in a sequence given the previous character. For instance, a neural network which generates sentences. If the first decision was a "T", then it is more likely that the next character is an "h" then a "p". By using a hidden state $h_t$, and updating this state after every computation, the network stores information on previous computations.

A high-level block diagram of an RNN is shown in Figure 2.11. As with conventional neural networks, an input vector $x$ feeds into the network producing an output $y$. However, in addition to producing the $y$, the internal state $h$ is updated as indicated by the arrow in Figure 2.11.

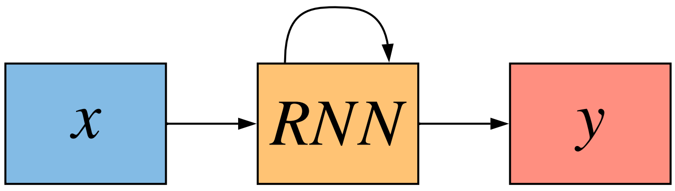


**Figure 2.11:** High level block diagram of a recurrent neural network.

The general equation for the hidden state in a recurrent neural network is Eq 2.6.

$$h_t = f_W(h_{t-1}, x_t) \tag{2.6}$$

Where

- $h_t$ is the hidden state at "time" $t$
- $f_W$ is some function with some weights $W$
- $h_{t-1}$ is the hidden state at "time" $t-1$
- $x_t$ is the input at "time" $t$

In order to understand RNNs, it might be helpful to look at its computational graph in Figure 2.12. At the start our hidden state is initialized as $h_0$. When the first input arrives, $x_1$, the $f_W$ function takes both $h_0$ and $x_1$ and computes $h_1$ as in Eq 2.6 and outputs the result $y$. When $x_2$ is entered, the network remembers by using $h_1$.

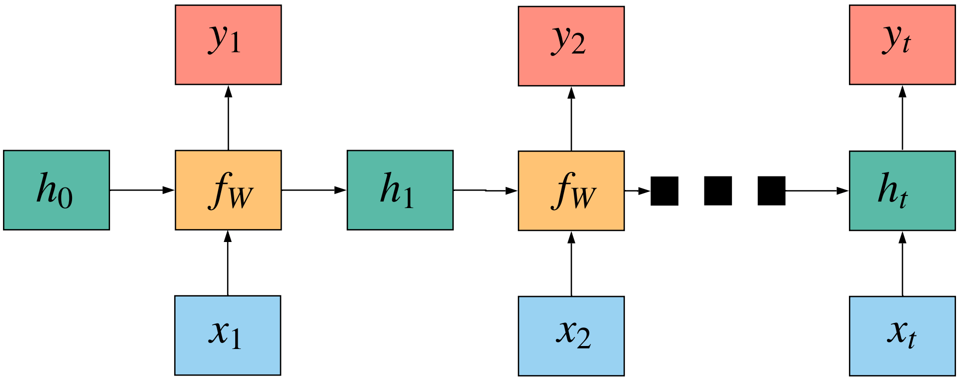


**Figure 2.12:** Computational graph of recurrent neural networks.

Chapter 3

# Literature Study

This chapter presents the literature study preceding the design and implementation. The first part presents failure modes in Non-Volatile Memory, classifying two common modes of failures. The second part looks at a machine learning approach taken to mitigate these errors.

If the reader does not seek a deeper understanding of these topics, parts of this chapter can be skipped. It is recommended that the basics are understood, such as definitions, before proceeding to the next chapter.

## 3.1 Failure modes in Non-Volatile Memory

Flash cells have multiple failure modes, with two of the most common being **erratic** and **moving bits**. A moving bit loses charge over time, causing an error when the leaked charge accumulates to a high enough level. An erratic bit shows sudden and unexpected jumps in charge levels while not showing a clear trend as a moving bit.

### 3.1.1 Erratic bits

An erratic bit is defined as a bit showing sudden and unexpected jumps between threshold voltage. These jumps are thought to be caused by positively charged clusters in the FG-MOS isolator. These clusters are caused by high voltage during erase and programming operations, such as Fowler-Nordheim tunneling [3].

The erratic behavior occurs through the dynamics of the creation or destruction of the clusters. These clusters are created by hot electron injection and destroyed by thermally emitted electrons detrapping the clusters [19]. An illustration of such a charged cluster can be seen in Figure 3.1.

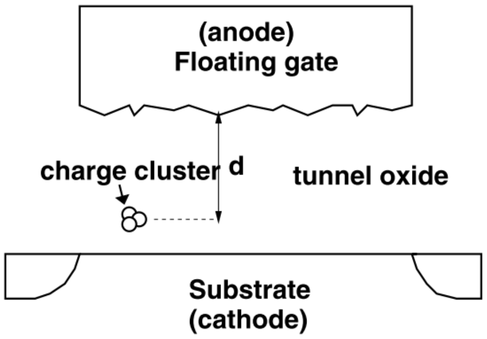
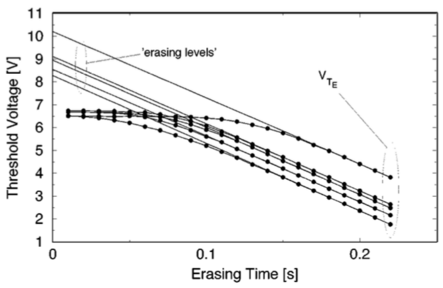
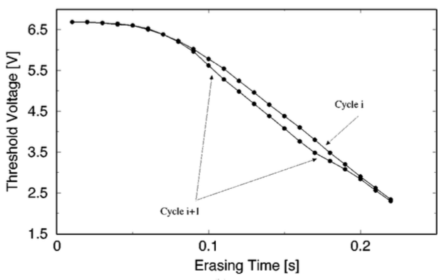


**Figure 3.1:** Positively charged cluster causing erratic behavior [19].

There have been attempts to analyze the behavior of these erratic bits such as *Analysis of erratic bits in flash memories* by Chimention [3]. A special erase pulse was applied on some flash cells, measuring their threshold voltage after each pulse. Healthy bits followed a linear trend as seen in Figure 3.2a, while erratic bits showed a deviation from the linearity as seen in Figure 3.2b.



**(a)** Healthy bit linear trend during special erasing pulse [3].



**(b)** Erratic bit linear trend deviation during special erasing pulse[3].

**Figure 3.2:** Healthy and erratic bit threshold voltage trend with special erase pulse. Figures taken from [3].

### 3.1.2 Moving Bits

*The following subsection on moving bits is taken from the literature study in [5].*

A bit is defined as **moving** when its threshold voltage moves slowly over time, on a timescale of months to years due to anomalous charge loss. One way moving bits differ from erratic bits is that moving bits are more predictable. By measuring the threshold voltage over time, a moving bit can be identified by looking at the slope. According to Schuler ”Anomalous charge loss in cells with tunnel oxide thickness of more than 6.4 nm results from chains of at least two or more less aligned defects” [14].

The rest of this section explores the reason for this leakage. *This gives insight into the nature of moving bits but is not necessary to understand or follow the rest of the text.*

When used to make electrical circuits, parts of the semicoductor is ”doped” to either be conductive or isolating. Sometimes atoms are not doped correctly, causing them to stay semiconducting when designed to be isolating.

The defects Schuler writes about are these ”wrong” atoms and are known as **traps**, which forms a conductive path through the silicon. Figure 3.3a illustrates how these traps construct this conductive path. If these traps align, a conductive path is constructed. This causes the electric charge stored in the cell to slowly leak over time.

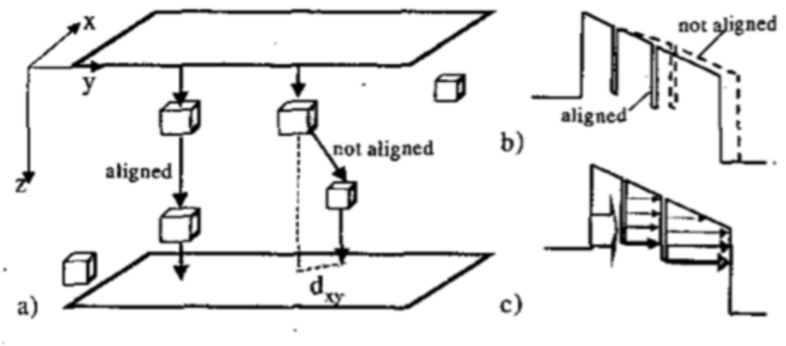


**Figure 3.3:** a) Schematics of a 3-dimensional distribution of traps building a leakage path.
b) Influence of misalignment on the band diagram of hthe effective tunneling path
c) Trap-assisted charge transport taking different energy lvels into account. Figure taken from [14].

According to F. Schuler, the leakage current can be modelled exponentially by a *1-step* *tunneling* model.

$$I_{leak} = \sigma A_{FN} F^2 exp\{-\frac{4}{3h}\sqrt{2qm^*(\phi_1 - \Delta\phi)^3}\frac{1-(1-\frac{Fx_t}{(\phi_1-\Delta\phi)})^{\frac{3}{2}}}{F}\} \tag{3.1}$$

$$\Delta\phi = \sqrt{\frac{qF}{4\pi\epsilon_{ox}\epsilon_0}} \tag{3.2}$$

Where

- $\Delta\phi$ is the barrier lowering
- $F$ is the electric field
- $A_{FN}$ is the Fowler-Nordheim parameter
- $\phi_i$ is the trap depth
- $\sigma$ is the effective cross-section
- $x_t$ is the tunneling distance

The full model explanation can be found in [14]. The model in Eq 3.2 is a simplified version of the *Multi-Phonon-Assisted tunneling model* presented in the same study, where the *trap to trap* distance was found as the most dominant parameter. The multi-phonon-assisted tunneling model found that there is a slow down of charge loss of some bits, which the simplified model could not predict as shown in Figure 3.4.

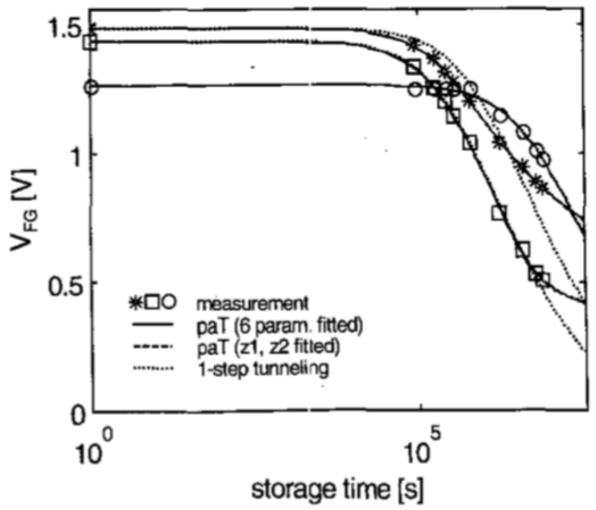


**Figure 3.4:** Measured transient charge loss of some typical Moving Bits as well as the simulated charge loss characteristic using the multi-phonon-assisted tunneling. Figure taken from [14].

## 3.2 Error correction in Non-Volatile Memory

Error correction codes are prevalent in flash memory due to the reliability issues with floating gate [8, 7]. In later years there has been an increase in the usage of MLC (Multi-Level Cell) technologies, which incorporates several bits on the same FGMOS transistor. As such, most modern research on increasing endurance in flash memory is done on MLC cells, as they are much more susceptible to error modes such as moving and erratic bits.

Traditionally simple Hamming code has been used on SLC, and are still used to some extent. It has however been an increase in the use of cyclic error correction codes such as Bose-Chaudhuri-Hocquenghem (BCH) codes, as simple linear codes such as Hamming codes are no longer sufficient on the larger page sizes of MLC [2, 8].

BCH codes offer multiple error correction capabilities, and efficient decoding algorithms exist. A major drawback of BCH codes is the need for highly parallel implementations too met the speed requirements for decoding, especially for MLC with large page sizes. A large area of silicone is needed to meet this speed requirement.

A paper by Chen et al. explored the possibility to use Reed-Solomone (RS) codes for error correction in NAND memories, and thus reduce the area needed [2]. RS codes are especially good at correcting burst errors, while errors in flash memories are random errors (they appear one at a time), making them on the surface not seem like an appealing choice. Chen argues that the reduction in complexity gained from RS codes compared to BCH codes still gives it a significant advantage.

The paper concludes with

“It can be concluded that the RS codes can achieve similar error correcting performance as the BCH codes in MLC flash memory applications with lower decoding complexity. Therefore, RS codes are more suitable for these applications. [2]”

### 3.2.1 Classifying erratic bits

The following section explores a method for classifying erratic bits. *This gives insight into the nature of erratic bits but is not necessary to understand or follow the rest of the text.*

In *Erratic Bits Classification for Efficient Repair Strategies in Automotive Embedded Flash Memories*, Zambelli et al. looked at classifying erratic bits into different categories. The classification method is proposed to be during processes such as Wafer Sort in the factory to monitor which type of erratic bit occurs in the specific flash implementation. By using these result, it is possible to speculate on the best strategy to combat erratic bits (ECC, Redundancy bits, or other solutions).

The paper proposes two statistical properties to classify erratic bits.

1. ”**Presence Ratio (PR)** is defined as the ratio of the erratic event occurrences during the test concerning the total erase cycles number” [19]

2. ”**Time of Life (ToL)** is defined as the maximum number of consecutive erase cycles displaying erratic behavior.” [19]

In simpler terms, *PR* tells how much erratic behavior is taking place while *ToL* tells how long an erratic behavior lasts. By calculating the average of these properties, they can be used to define three signatures for erratic bits.

- Signature 1: Low PR (lower than the average PR) and short ToL(shorter than the average ToL)
- Signature 2: High PR, Long ToL
- Signature 3: High PR, Short ToL

Figure 3.5 shows an example of a erratic bit under the signatures described above.

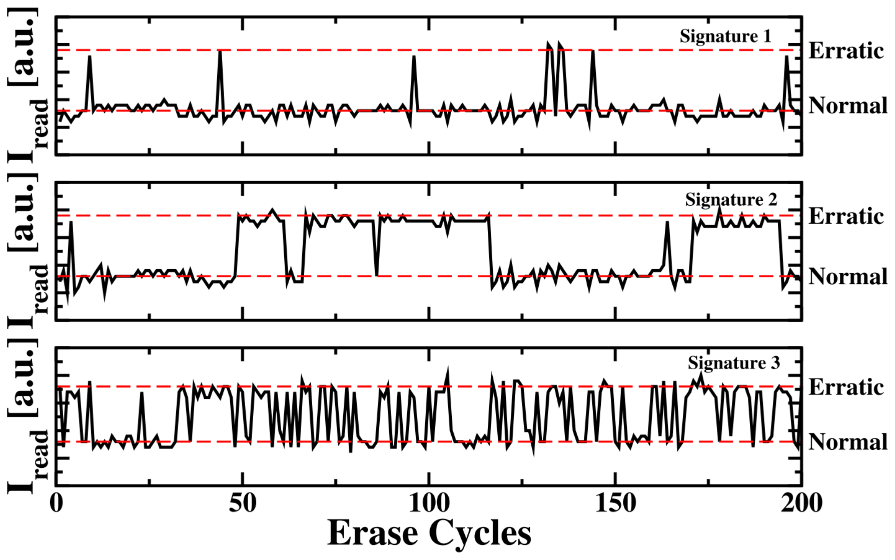


**Figure 3.5:** ”Examples of different EB (Erratic Bit) signatures. Normal and Erratic states for the read current ($I_{read}$) are highlighted for clarity” [19].

Zambelli suggests that the first signature can be treated only by ECC, as they occur quite seldom. The second signature can be treated with redundancy bits, as they have a higher occurrence probability and have predictable failing behavior. The third signature is very unpredictable and often occurs, making them the most dangerous for the flash. These can be handled with either redundancy or ECC.

### 3.2.2 Machine Learning approach

The following sections look at using machine learning to reduce flash failures in contrast with traditional ECC. It is recommended reading before Chapter 9.

One of the big drawbacks of ECC is the amount of redundant storage required. In order to recover about 1-2% of the errors in a page, 5-10% of the memory area is needed to store this information [4]. This 1-2% error is referred to as the **Bit Error Rate (BER)**.

In order to reduce the amount of redundant information stored, a machine learning approach has been explored to predict cell failure [4, 10]. By predicting cell failure, **redundant bits** for these predicted cells can be stored instead of ECC for the entire page. Redundant bits only require 1% of data to recover 1% of the memory area, vastly reducing the amount of storage needed in contrast to conventional ECC solutions.

In ”Machine Learning Prediction for 13x Endurance Enhancement in ReRAM SSD System”, such as solution was explored for ReRAM. ReRAM is a type of Non-Volatile memory which shows promise but suffers from a higher variability than traditional flash memories [4, 1]. The study explored several solutions to predicting cell failure, two machine learning approaches, and one non machine learning approach.

Two prediction types were evaluated, one to predict random behavior between PROGRAM/ERASE cycles, and one to predict likely-to-fail cells. Both of these types were tested on different machine learning methods, *Random Forest*, *Neural Network* and *Support Vector Machine*.

The ”predict likely-to-fail cells” used 12 variables on $12 * 10^3$ points of data. As can be seen in Figure 3.6, the endurance can either be increased by 13 times, or the BER can be reduced by 2.85 times compared to having no prediction or replacement. The best result came from the Random Forest algorithm, predicting 65% of the failures correctly.

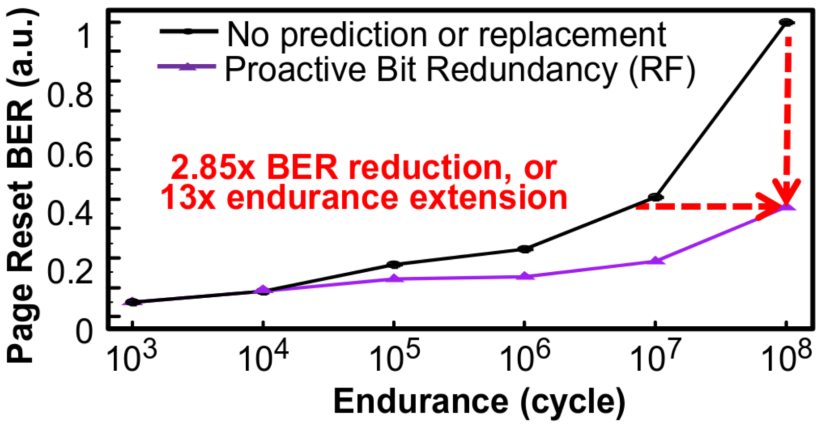


**Figure 3.6:** Reliability improvement with Proactive Bit Redundancy [4].

### Attempts made on flash memories

There has been some attempt to use machine learning for flash in the same manner as ReRAM. In ”Machine Learning-Based Proactive Data Retention Error Screening in 1Xnm TLC NAND Flash”, the usage of machine learning algorithms to detect ”Program Disturb” errors. A **program disturb (PD)** error is the phenomena where a neighbouring cell is incorrectly programmed. These *PD-Weak* cells have 2.4x worse data retention than their non-weak cells [10].

The study used three different machine learning algorithms, *Logistic Regression*, *Random Forest* and *Support Vector Machine*. Two metrics were used to describe the performance, *detection rate* and *cost* which are both described in Eq 3.3 and Eq 3.4

$$Detection\ rate = \frac{Correctly\ identified\ PD\ cells}{All\ PD\ Weak\ Cells} \tag{3.3}$$

$$Cost\ (False\ Detection) = \frac{Incorrectly\ detected\ as\ PD - Weak}{All\ PD\ Weak\ Cells} \tag{3.4}$$

All three algorithms performed quite similar, with a result of $40\%$ detection rate and a cost of $8\%$. The authors of the study argue that at first glance $40\%$ might seem low, but at such a low cost, it could be useful to error-prone cells through screening. It also shows the promise of machine learning methods to solve these problems in non-volatile memories.

Chapter 4

# Specification of project

This chapter outlines the functional specifications of the project as well as the acceptance criteria for completion

## 4.1 Functional specifications

1. The goal is a fully functioning ”Error Prevention Scheme,” which detects and repairs failing bits in flash memories by monitoring their cell charge over time. This data prevention scheme should monitor and read the cell charge of the flash at appropriate times, storing important data points, and using these for later analysis to find failing cells.
2. Which data points are deemed important and how these data points are analyzed is a core part of the project. The error prevention system should have two different modes, one distributing the storage fairly among the cells, the other prioritizing cells with a higher chance of failure. Previously a data collection scheme for cell charge has been developed, which is a module in the full error prevention scheme.
3. In addition to an error prevention scheme, a machine learning approach should be explored and discussed, but not implemented or tested.
4. As the error modes very rarely occur and previous accelerated aging techniques have been unsuccessful[5], a simulator for an NVM Flash memory should be developed. This simulator should be based on statistical properties of the flash, whereas each cell should not be simulated using mathematical models of floating gate MOSFET. However, some simulating of the individual cells should be made, but with a simple model.

5. The first iteration of the prevention system should be written in C and implemented on a PC. This first iteration should serve as a ”proof of concept” using the simulator. Different metrics should be logged, such as how cells are deleted and which data points are stored at any given iteration. The design should keep an AVR32 port in mind, optimizing for storage not performance.

## 4.2 Acceptance criteria

The project is deemed complete when the following acceptance criteria have been met. As this thesis is research-oriented, the acceptance criteria are vague by design. All criteria has been marked with the functional specification they implement (FS x means Functional Specification Number x).

1. A simulator has been developed. It simulates normal, moving, and erratic cells. **FS 4**
2. The simulated moving cells are mimicking the multi-phonon-assisted tunneling model. **FS 4**
3. The simulator has been evaluated in terms of performance and how it tracks its real counterpart. **FS 4**
4. A error prevention system has been developed for the x86 platform. It logs, tracks and reprograms failing cells. **FS 1 & 5**
5. The prevention system deletes redundant data points with little or no information gain. **FS 2**
6. The prevention system has two modes. One distributes the storage fairly among the cells. The other tries to prioritize cells with a higher chance of failure. **FS 2**
7. The prevention system stores the data points in a storage-optimized manner. **FS 5**
8. A machine learning approach has been explored separately to the proposed error prevention scheme. **FS 3**

Chapter 5

# Design of simulator

This chapter introduces the design of the simulator for simulating normal, erratic, and moving cells. It introduces the two main modules, the simulation loop and a network interface for controlling the simulator.

## 5.1 Introduction

As the error modes of NVM such as moving and erratic bits occur very seldom, it is hard to test a system against these errors. Thus the need for an NVM Simulator arises, which can simulate the flash and insert error modes. More specifically, it simulates the cells cell current.

When simulating hardware, one must always choose which abstraction layer to use. For instance, in this scenario, every individual cell could be simulated through a *SPICE* simulation; however, this would be very complicated and time-consuming while not adding particularly relevant functionality. The solution approached in this thesis uses a higher abstraction level, depending only on statistical properties observed from an earlier measurement.

The system consists of two main modules, the *simulator* and *Network Interface*. The simulator calculates and stores simulated cell current values. The network interface provides an interface for applications and user which uses the simulator, allowing remote access.

## 5.2 Simulation Loop

The simulator module stores the simulated values and calculates their next value. It runs in a loop, where each iteration progresses time. The simulator simulates three types of cells

1. **Normal Cells:** Cells which operates as expected. No errors, cell current value stays constant with some minor fluctuations.
2. **Moving Cells:** Cells which are moving. The cell current value declines slowly over time. See Chapter 3.1.2
3. **Erratic Cells:** Cells, which shows sudden and unexpected jumps in cell current value. See Chapter 3.1.1

The number of normal cells is much larger than the amount of moving and erratic bits. Figure 5.1 shows a simple flowchart of the simulation loop. All the cells are initialized to their default values. The simulator waits for a signal ahead of each iteration before it proceeds to iterate (calculate the next value) for all three types of cells.

As discovered in [5], normal flash cells tend to fluctuate to a minor degree. If there were no fluctuations in the simulated normal cells, the prevention system would only have to look for any $\Delta I$ to identify any moving or erratic cell. To make sure the developed system is robust, these kinds of shortcuts (which could be made by mistake) must be avoided as much as possible.

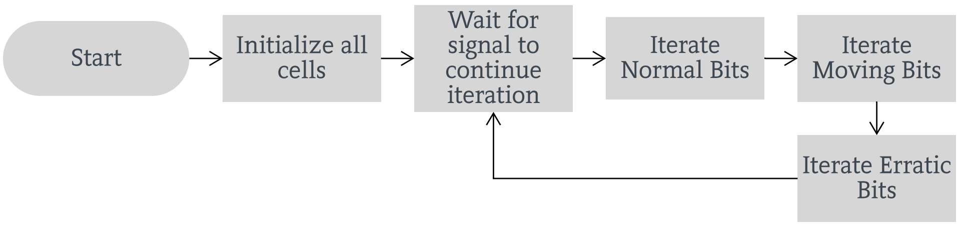


**Figure 5.1:** Flowchart of the simulation loop.

The simulator can run several simulations in parallel with differing states. There exist one simulation loop with its respective cell values for each simulation that is running. This allows several version of the error prevention system to be tested in parallel, allowing for greater flexibility during development.

## 5.3 Network Interface

The network interface module provides an interface for clients which uses the simulator, allowing remote access. As the simulation can be quite resources intensive, having a network interface allows the simulation to be run on a powerful remote machine while the client is on a weaker system such as a laptop or even a powerful microcontroller. Through the network interface, the client can issue commands such as

- Progress the time, i.e. iterate the simulator
- Read the cell value for a specific word or bit
- Make a bit erratic, moving or normal

The simulator never initiates any communication with the client, and only responds to commands and requests.

Chapter 6

# Implementation of simulator

This chapter outlines the implementation of the simulator. The simulation loop is discussed in detail, including how each type of cell is calculated for the next iteration. The effect temperature has on cell current measurements are discussed and introduced into the simulator. Finally, the network interface and a manager module are presented with a command list for interacting with the simulator.

## 6.1 Introduction

The simulator is implemented using Python 3.7 with Numpy 1.16, which gives the flexibility of python while preserving computational performance through numpy. As discussed in the design chapter, there are two main modules, a network interface, and the simulator.

The simulator module has two sub-modules, the ”simulation” module and ”manager” module. The simulation module runs simulations, and the manager controls which simulations are active, and redirects messages from the network interface to the appropriate simulation.

Every simulation has a unique number called its *simulation ID*. The manager stores every simulation with this number, allowing the user to access several concurrent simulations at will.

## 6.2 Simulation Loop

In [5] it was discovered that the cell current of a programmed Microchip flash is normally distributed around $20.68$ with a standard deviation of about $0.9149$. These numbers depend on various factors such as temperature. As a new simulation is started, cell values for all cells are randomly calculated normally using the above mean and standard deviation.

A random subset is chosen for each iteration. All cells in this random subset have their next value calculated depending on which ”type” of cell it is. For instance, a normal cell would most likely show no changes from the previous iteration, while a moving bit would most likely show a lower value than the previous iteration. This loop is illustrated in Figure 6.1.

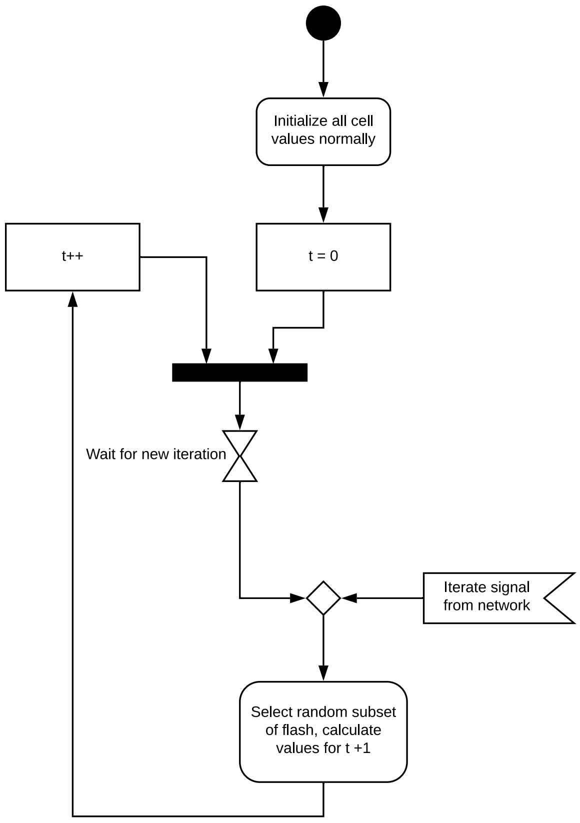

Initialize all cell values normally
t++
t = 0
Wait for new iteration
Iterate signal from network
Select random subset of flash, calculate values for t +1

**Figure 6.1:** UML Activity Diagram of the simulation loop.

### 6.2.1 Calculating the next iteration

There are three ”types” of cells in the simulator, *normal*, *moving* and *erratic*. These all have their next value calculated differently. These error modes are chosen as they are the most common, the aim of this thesis is to detect these, especially the moving bits.

All values are digitized into buckets with the size of $0.25$. For instance, $17.82$ becomes $17.75$ and $13.98$ becomes $14.0$. The data collection scheme developed in [5] uses digitized values, and is therefore also included in the simulator to mimic this as much as possible.

#### Normal Cells

The following model is based on data found in [5]. A normal cell is defined as being stable with no leakage. However when measuring the cell current, some variations are observed, such as small fluctuations when measured over a substantial amount of time.

The reason for these changes in measurement is not known, but there are some factors, such as temperature, which are known to influence the measured value significantly (See Chapter 10). In the end it is probably a matter of non-ideal components.

The simulator implementation of normal cells can change their value by a predefined $\Delta I$ on a given probability. The probabilities are $P_p$ for increasing and $P_n$ for decreasing. When a cell has changed its value in either direction, $P_p$ and $P_n$ changes to increase the chance of an opposite changed in the next iterations. This simple model allows for cells to change their value by a relative amount around their initial value. An example of such cell can be seen in Figure 6.2. The algorithm is also illustrated in Figure 6.3.

**Example:** A cell has $I_0 = 22\mu A$, $P_p = P_n = 0.1$ and $\Delta I = 0.25$. There is a $10\%$ chance for the cell to either decrease or increase. The positive probability ”wins”, giving is the $22.25\mu A$ value. This changes $P_p = P_p - \Delta P = 0.05$ and $P_n = P_n + \Delta P = 0.15$ for the next iteration.

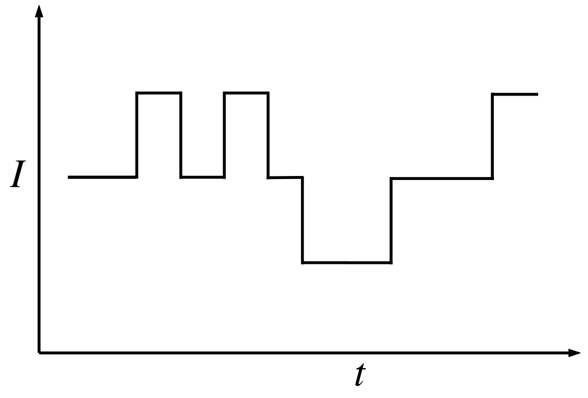


**Figure 6.2:** Simulation of a normal cell.

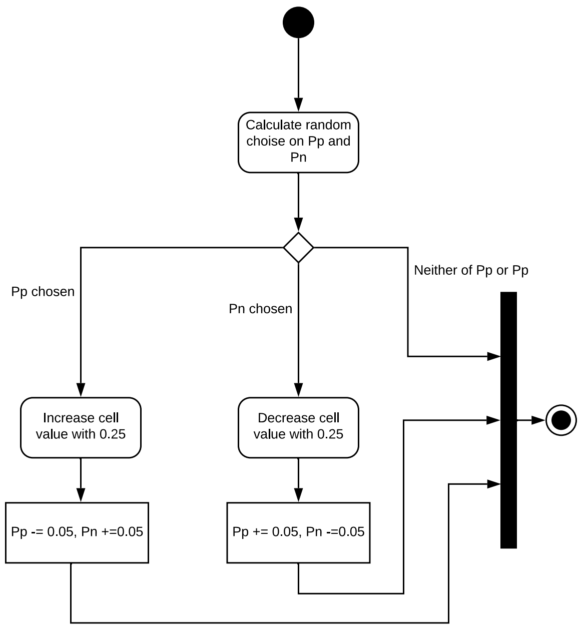

Calculate random choise on Pp and Pn
Pp chosen
Pn chosen
Neither of Pp or Pp
Increase cell value with 0.25
Decrease cell value with 0.25
Pp -= 0.05, Pn +=0.05
Pp += 0.05, Pn -=0.05

**Figure 6.3:** Algorithm for calculating then next iteration of a normal cell.

### Moving Cells

Moving cells try to mimic the observed moving cells in reality, which loses charge gradually over time. This is a very important error mode, as they are relatively common [1] and easiest to catch. The moving cells follow an exponential trend given by Eq 6.1.

$$I(t) = ae^{bt} + c + \omega \tag{6.1}$$

The reasoning for this model is the Multi-Phonon-Assited leakage model described by Schuler [14], which models moving bits as an exponential curve with a non-zero $c$. A challenge is to find good $b$ and $c$ values to be as close to reality as possible.

A moving bit was discovered in [5], and the $b$ and $c$ parameters are based on those results. The parameter $a$ is the moving cells initial value. Gaussian white noise[2] $\omega$ is added to mimic the observed read error. A graph of such a moving bit is shown in Figure 6.4.

[1] Strong emphasis on relatively, error mode occurs in a parts per billion scale
[2] Noise with zero-mean and a fixed standard deviation

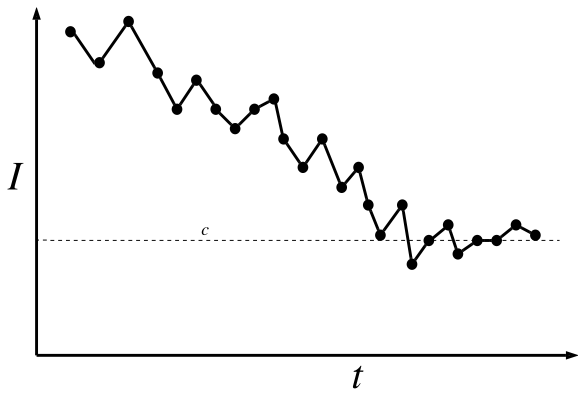


**Figure 6.4:** Simulation of a moving bit.

### Erratic Cells

Simulated erratic cells try to mimic erratic cells, which exhibits sudden and unexpected jumps of cell current. In the simulator, erratic cells are a special type of normal cells, in the sense that they following the same algorithm as described in Figure 6.3. In addition, they can also give a very big read error randomly on a given probability. This gives the shape as illustrated in Figure 6.5.

Please note that this implementation sets the erratic bits to their original value after a while, which is not how erratic bit behaves. Usually, erratic bits do not return to their original value, but stay at their new value until reprogramming.

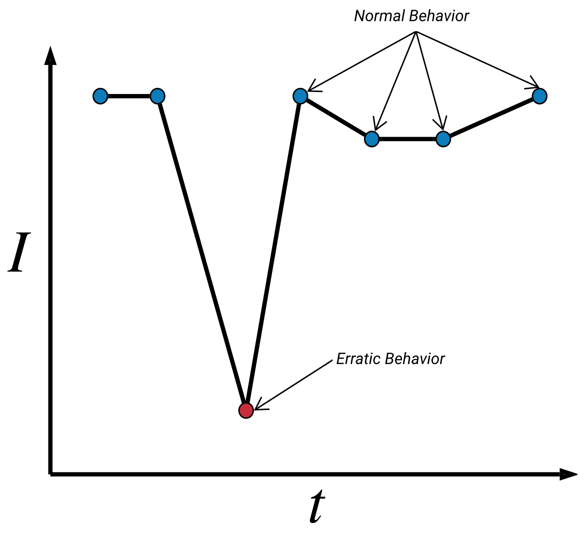


**Figure 6.5:** Simulation of an erratic cell. Blue cells are showing normal behavior. The red cell shows erratic behavior.

## 6.3 Temperature effect

Temperature has been shown to have a substantial effect on the cell current measurements [5]. The experiment presented in Chapter 10 shows a strong indication that the measured cell current changes proportionally with temperature. To include this effect in the simulator, every time a measurement is requested from the simulator, a temperature error $e_{temp}$ is added. This relationship is shown in Eq 6.2.

$$I_{returned} = I_{actual} + e_{temp}(T) \tag{6.2}$$

The function $e_{temp}(T)$ is a linear function which converts a temperature to the corresponding error in amperes depicting the error shown in Figure 10.1. This temperature error is normally[3] initialized with an appropriate value taken from taken from [5]. Initializing $e_{temp}$ in such a manner might not be very realistic if there are multiple measurements in a row.

For instance, two measurement such as ($t = 00:00$[4] ,$T = 20°C$) and ($t = 00:10$,$T = 120°C$) could be returned by the simulator. However, a temperature change of $100°C$ over 10 seconds seems very unreasonable. A solution to this problem could be to simulate the temperature for every time iteration, and not give a random value each time.

[3]Normally as in a Gaussian distribution
[4]minute:seconds

## 6.4 Network Interface & The manager Module

All interaction with the simulator is done through a network interface, implemented as a TCP server. Messages can be sent to the server in a predefined JSON format which responds with a JSON message.

The **manager** keeps track of which simulations are active and their simulated values. This allows many simultaneous simulations to take place. Every simulation has a unique ID, which identifies the simulation to any client.

If a command received over the network interface is simulation specific (such as to measure cell $y$ for id $x$), it is redirected to the manager. If it is not a simulation specific request, it is handled by the network interface module directly.

When a simulation specific request is redirected to the manager, the simulation number is deciphered. The message is forwarded to the correct simulation, which responds accordingly. The response is sent back to the network interface and the initial client. This is shown as a sequence diagram in Figure 6.6.

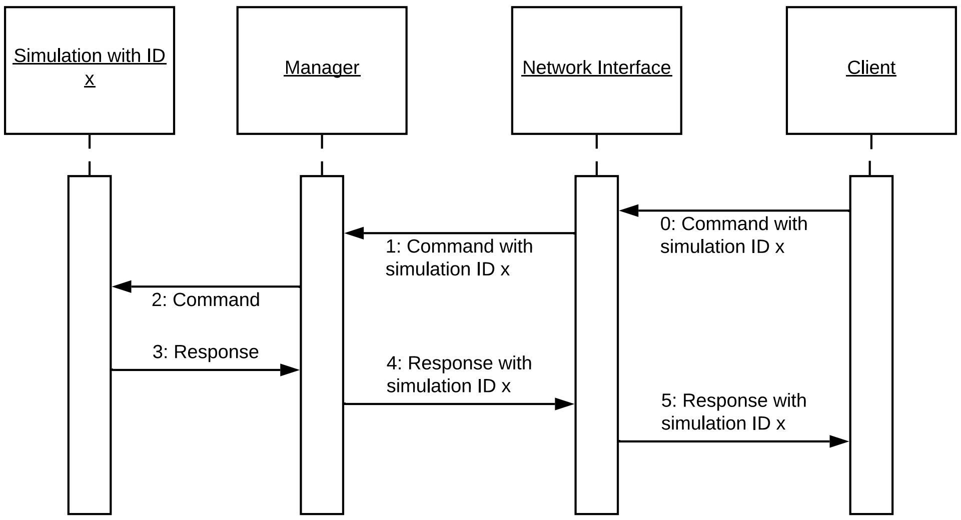

Simulation with ID x
Manager
Network Interface
Client
0: Command with simulation ID x
1: Command with simulation ID x
2: Command
3: Response
4: Response with simulation ID x
5: Response with simulation ID x

**Figure 6.6:** A UML Sequence diagram showing the communication for a simulation specific request.

### 6.4.1 Command list

#### Start Simulation

Starts a simulation, responds with the simulation ID of the newly spawned simulation.
**Message:** {"cmd": "start_sim"}
**Response:** {"status": "OK/FAIL", "sim_nmb": X}

#### Progress time

Progresses time for a simulation with id X for N iterations.
**Message:** {"cmd": "prgs_time", "sim_nmb": X, "n": N}
**Response:** {"status": "OK/FAIL"}

#### Request data for single bit

Returns the cell current value for a single bit number Y.
**Message:** {"cmd": "rqst_data_sb", "sim_nmb": X, "bit": Y}
**Response:** {"status": "OK/FAIL", "data": bitValue}

#### Request data for a word

Returns the cell current value for all bits in word W
**Message:** {"cmd": "rqst_data_word", "sim_nmb": X, "word": W}
**Response:** {"status": "OK/FAIL", "data": [bitValue0, bitValue1, ..., bitValue31]}

#### Request mean of data

Returns the mean of all the programmed cell.
**Message:** {"cmd": "rqst_true_mean", "sim_nmb": X}
**Response:** {"status": "OK/FAIL", "data": mean}

#### Request current time

Returns the current iteration number.
**Message:** {"cmd": "rqst_time", "sim_nmb": X}
**Response:** {"status": "OK/FAIL", "data": time}

#### Request a sweep

Performs a sweep mimicking the FDMA sweep. Please see Chapter 8.2.1.

Chapter 7

# Design of error prevention system

This chapter introduces the design of the error prevention system. It discusses the main loop of the system; measuring, storing, and reprogramming cells. A summary from the prelude project [5] on measuring cell charge follows. The chapter ends with a short discussion on storing data points and how the system deals with the effect temperature has on the cell current measurements.

## 7.1 Introduction

The error prevention system is the main goal of this thesis. At aims to detect error modes in the flash before the error occurs. By monitoring the cell charge values of the flash over time and storing relevant data points, the system should be able to decide on if a bit is going to fail. The primary focus is on moving bits, as these are the easiest to detect and correct.

Certain limitations have quite an impact on the design. As the prevention system is designed to run concurrently with other firmware, the amount of flash dedicated to both storing cell current values and the program code itself is severely limited. This has led the design to be more focused on optimizing for space and not for performance.

The first version of the prevention system is not built directly for the AVR32 MCU, but for Intel x86. This allows for simpler development, debugging, and interfacing with the simulator. However, all the code is written in C with a port to AVR32 in mind. Specifically for the mXT1067T from Microchip, which is an AVR32 touch controller.

## 7.2 Main Loop

The purpose of the main loop is to find cells that are indicating a failure, store data points for these cells over time and make a decision if the cell should be reprogrammed or not. This is illustrated as a flowchart in Figure 7.1. It starts by measuring the cell. If there is free storage, the cell is stored. If there is no space, the ”least valuable data point” must be deleted before adding the new data point. If the new data point, together with the previously stored data points, indicate a failing cell, it is reprogrammed.

The term ”least valuable” is vague by design, and is the data point currently stored which gives the least amount of information. For instance, a cell has four data points (5, 20.25), (10, 20.25), (18, 20.25), (26, 18.75) [1]. The least valuable data point would be (10, 20.25), as it adds very little information. The implementation of the ”find least valuable data point” algorithm is therefore important for the performance of the system.

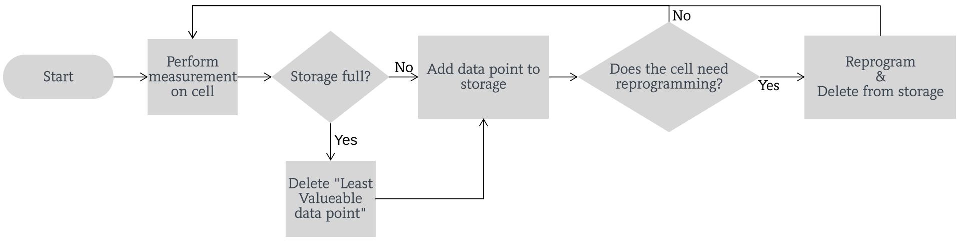

Start
Perform measurement on cell
Storage full?
No
Yes
Add data point to storage
Delete "Least Valueable data point"
Does the cell need reprogramming?
Yes
No
Reprogram & Delete from storage

**Figure 7.1:** The main loop of the system as a flowchart.

### Measuring cell charge

In order for such a system to work, it must be able to measure the cell charge for an arbitrary cell in the flash. Such a cell charge collection system was developed in the prelude project for this thesis [5]. What follows here is a summary.

When reading the binary value of a flash cell, a read voltage is applied to the gate, which causes current to flow from source to drain. The amount of current that is allowed to flow is determined by the amount of charge stored in the floating gate. This output current is called the **cell current**.

The cell current is compared against an internal current generator. If the cell current is lower than the internal current generator, it is **programmed** with a logical value of zero. If the cell current is greater than the internal current generator, the cell is **erased** with a logical value of one.

[1]([hours], [$\mu A$])

It is the value of the internal current generator, which defines how much cell current must be present for a cell to be considered programmed. An internal test-mode called **FDMA (Fast Direct Memory Access)** utilizes this fact by overriding the internal current generator. Figure 7.2 illustrates this.

When the current generator is overridden with the FDMA Current Generator, the current is called **FDMA Current**. While monitoring the digital value of a cell (zero or one, programmed or erased), the FDMA Current is increased from a low value to a higher value with a given increment. As soon as the digital value of the cell flips (goes from zero to one or vice verse), the FDMA current has just surpassed the cell current. In other words, the FDMA current at that moment is a close approximation of the cell current.

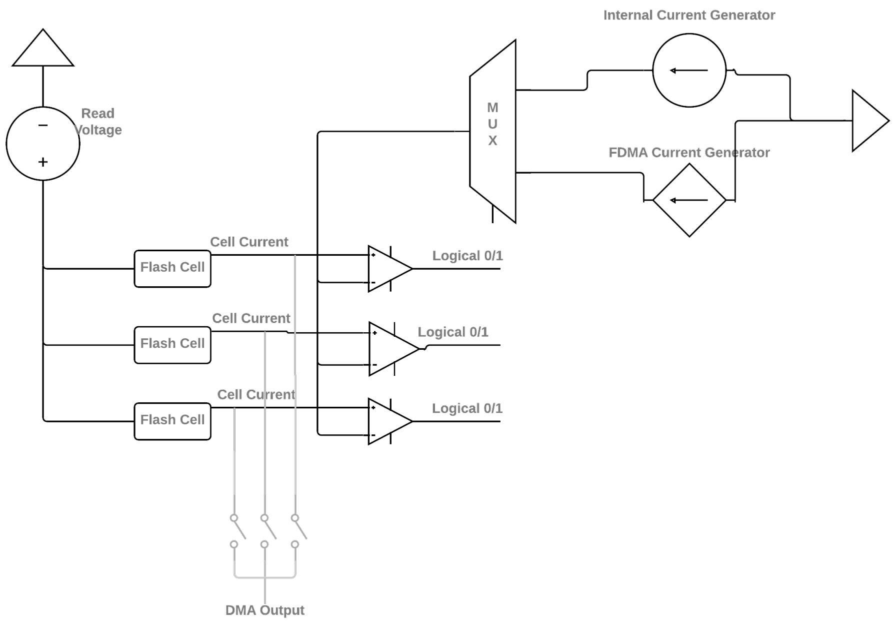


**Figure 7.2:** Illustrative circuit diagram of the FDMA Test mode [5].

## 7.3 Storing data points

In order to monitor over time, data points must be stored. Each data point contains info on which cell it belongs to, the time and the measurement value. As there is no sense of time for a microcontroller, the time must be read from an external system. This should not be a problem as a MaxTouch[2] chip is always part of a larger system that tracks time.

[2]Touch controller from Microchip which the target for the error prevention system

As there is a minimal amount of space available for storing data, care must be taken on how the data is stored and which data is stored. The data points are stored on the flash, and must thus not be written too excessively.

## 7.4 Temperature effect

In [5] it was discovered that temperature had a significant effect on the cell current measurements. As the proposed system should function under temperature ranging from $-20°C$ to $100°C$, this effect must be taken into account when measurements are logged. The error induced is assumed uniform across all cells, increasing linearly with time. This claim is backed up by the experiment conducted in Chapter 10.

If the error, $e_{temp}$, is applied uniformly and linearly across all cells, it is possible to find $e_{temp}$ by looking at the mean cell current of all the cells $\mu$. As a step in the calibration of the chip, the mean at $22°C$ is logged as a reference $\mu(T = 22°C) = \mu_{ref}$. This gives $e_{temp}$ by equation 7.1. For every measurement of a cell, the temperature error is calculated and subtracted before the measurement is stored in the system with Eq 7.2.

$$e_{temp} = |\mu_{ref} - \mu| \tag{7.1}$$

$$I_{cell} = I_{measured} - e_{temp} \tag{7.2}$$

Finding $\mu$ is not directly trivial as the amount of cells is very large[3]. The cell measurements are assumed to be normally distributed as indicated by the results in [5]. The *central limit theorem* says if the cell measurements are normally distributed, the mean $\mu$ is approximately equal to the mean of a small subset of the cells [16]. This gives the following procedure

1. Measure cell current $I_{measured}$
2. Calculate mean at current moment in time $\mu$ by selecting a random subset of size $N$.
3. Calculate $e_{temp}$ with Eq 7.1
4. Use $I_{cell}$ from Eq 7.2 as the cell measurement for all further calculations, decisions and storage.

[3] 128KB flash has $128 * 1024 * 8 = 1\ 048\ 576$ cells

Chapter 8

# Implementation of error prevention system

This chapter outlines the implementation of the error prevention system. The different modules and their interactions are discussed. How cells are measured, which cells are stored and which cells are programmed is discussed in detail. The storage modules internal data structure is presented with accompanying algorithms. A section on the main loop and putting all modules together wraps up the chapter.

## 8.1 Introduction

The system is written in C, as the only compiler for AVR32 is a C compiler. The C programming language also provides a low-level interface to memory, which suits the needs of this system very well. All unit-tests for the system has been written in C++ with GoogleTest, which is a versatile and powerful testing framework. The code has been partially documented with DoxyGen.

There exist five primary modules for the system. Figure 8.1 shows the dependencies of the modules.

- **Search:** Conducts measurements and decide whether or not to store data points. Also determines if a data point should be reprogrammed.
- **Storage:** Stores data points and provide metrics on the stored data points.
- **Simulator Interface / Flash Controller:** In the x86 version, this module links to the simulator. On the AVR32 version, this module links to the flash controller.

- **ControlPanel:** Used during development. Allows the user to send commands and retrieve metrics as the system is running.
- **Datalog:** Used during development. Logs metrics to disk.

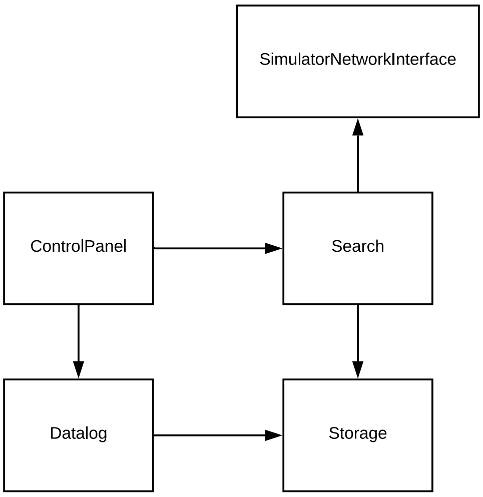


**Figure 8.1:** The overall design of the error prevention system. Arrows indicate dependency.

## 8.2 Searching

The **searching** module finds and determines which data points to store and if a given cell should be reprogrammed. The heuristics of the searching system is to log the data points for the cells which are most likely to fail. When enough data points have been gathered, the module decides if any cells in the system should be reprogrammed.

It is not possible to measure and store data for all cells, as there are too many. It is, therefore, necessary for the system to find out which cells to measure, and how often these should be measured.

### 8.2.1 Which cells to measure

The system measures a set of cells at a given iteration[1]. A cell can exist in the set if there is data stored for the cell, or if it is below a given fixed value $I_w$. The reason for using an arbitrary cutoff is that the absolute value of the cell is the only information available. As there is such a quantity of cells (128KB), it is not possible to obtain any trend information.

[1]The time between iteration, and the size of this are parameters which can be tuned.

To find all cells below $I_w$, every cell could be measured and compared. Such an approach would take a very long time (measuring 128KB flash takes approximately 20 minutes [5]), and would not be viable.

Instead, the system uses a technique called **sweeping**. In Chapter 7.2, the FDMA test mode was explained. The same technique can be used to find all cells below $I_w$ quickly.

1. Log all cells that are zero (programmed cells)
2. Set the internal current generator to $I_w$
3. All logged cells which flipped are below $I_w$

This procedure is illustrated in Figure 8.2. Every circle is a cell where the yellow circles are the targeted cells.

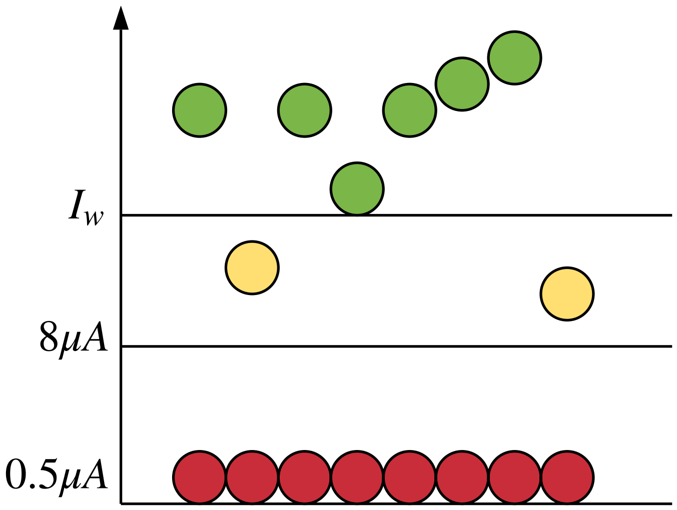


**Figure 8.2:** Illustration of the sweeping method. Every circle is a cell. Red cells are programmed (0), green cells are erased (1), and yellow cells are the programmed cells below $I_w$. Please note that this graph is 1D (no x line).

### 8.2.2 Which cells to store

After the sweeping procedure has finished, the system is left with a set of cells. There is a finite amount of data points which can be stored, and there must be some mechanism to decide which point is stored and which point is discarded. There are two modes implemented on the system

- **Mode one**: This mode tries to be fair on storage allocation. The storage is split evenly among all cells below $I_w$. If a new data point $p_{new}$ arrives for a cell $c$ when the storage is full, an old data point $p_{worst}$ for $c$ must be deleted.

- **Mode two:** Instead of distributing storage evenly as mode one, the more interesting cells have more storage than those deemed less interesting[2]. If a new data point $p_{new}$ arrives for a cell $c$ when the storage is full, the point $p_{worst}$ for the cell $c_{worst}$ is deleted.

In both of these modes, the term "worst" refers to the least valuable data point (see Chapter 7.2). A metavalue is assigned to every cell and point to determine which cells and points are the worst. A high metavalue indicates that the cell or point contain a high degree of information. Mode one only requires a metavalue for points, as the different cells are treated "fairly." Mode two requires a metavalue for points and cells both.

***Implementation detail one***: The code must be compiled with "-DMODE2" to use mode two. If this flag is not present, the system is compiled with mode one. If the default size of the storage is changed, some internal parameters such as *maxNmbPtsPerCell* must also change. Please read the doxygen comments in the code for more information.

***Implementation detail two***: Mode one allows for an unfair split of data points if there is room for it. When the storage is full, new data points are prioritized in a manner which makes a fair split.

*Example:*

There is room for ten data points, and two cells exist, 2 and 38. Cell 38 has two data points, and cell 2 has eight. If any new data points arrive (for either 2 or 38), a data point for cell two must be deleted until both cell two and cell 38 has 5 data points, obtaining an even split. When new data points arrive from this time, a data point from the same cell must be deleted.

### Assigning a metavalue to points

As previously stated, a data points metavalue indicate how much information it contributes. Some observation on the amount of information from a data point

1. The closer the point is to any neighboring points in time and value, the less information it contributes
2. A point closer to the start and end contributes more information than points in the middle[3]
3. The first and last point contribute the most amount of information

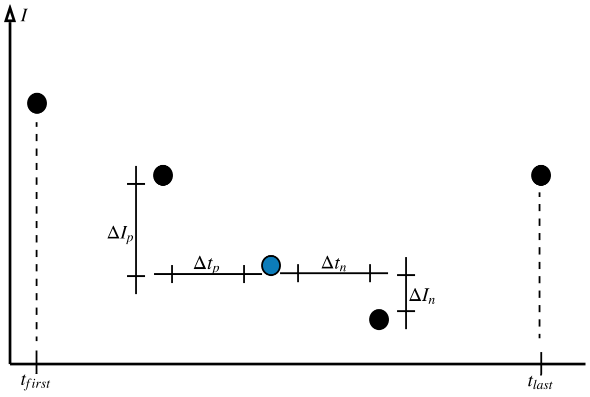
These observations are the basis for the metavalue $M$ given in Eq 8.1. The W's are weights which must be tuned for the specific chip. The rest of the parameters are shown in Figure 8.3a. The first and second term of Eq 8.1 represents the first observation. Small $\Delta I$ and $\Delta t$ contributes to a small metavalue.

[2]Interesting in the sense of likelihood to fail

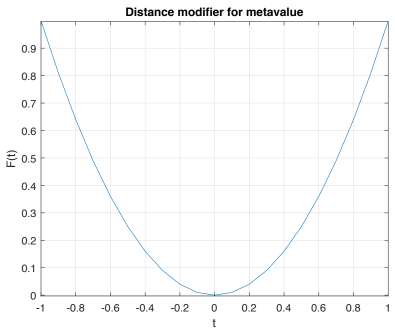
[3]Middle refers to the midpoint between the first and last data point in time

The last term $W_D\left(\frac{t}{t_{first}+t_{last}}-0.5\right)^2$ represents the second observation. It maps the data points time $t$ between zero and one before squaring it, making points with $t$ close to the middle have a lower value. How big of an impact this modifier should have is tuned with $W_D$. This modifier is plotted in Figure 8.3b. The third observation is implemented as an exception for the first and last point, always returning a value of $\infty$.

$$M = W_I\left(\Delta I_p + \Delta I_n\right) + W_t\left(\Delta t_p + \Delta t_n\right) + W_D\left(\frac{t}{t_{first}+t_{last}} - 0.5\right)^2 \tag{8.1}$$



**(a)** Definition of the different parameters used in Eq8.1. Blue dot is the data point in question.



**(b)** Graph of the distance modifier $\left(\frac{t}{t_{first}+t_{last}}-0.5\right)^2$. Data points reduce in value as they approach the middle.

**Figure 8.3:** Representation of metavalue $M$ from Eq 8.1.

### Assigning a metavalue to cells

A cells metavalue indicate how likely it is to fail. Some observation on metavalue for cells are

1. Cells with a higher $\Delta I$ from start to end are more likely to fail
2. Cells with a high negative derivative are more likely to fail
3. Cells with a "flat" trend at the end are more likely not to fail

These observations are the basis for the metavalue $M$ given in Eq 8.2. Figure 8.4 shows the definition of the different parameters. The W's are weights tuned for each specific chip. The first observation is represented as the total change in current from the first to the last cell. The second observation is represented as the total derivative (**TD**) from the first to the last cell. The third observation is implemented as a derivative of the three last data points (**D3**) [4].

$$\begin{aligned} M &= W_{\triangle}\Delta I - W_{td}\frac{dI}{dt} - W_{d3}\frac{dI_3}{dt_3} \\ &= W_{\triangle}\Delta I - W_{td}TD - W_{d3}D3 \end{aligned} \tag{8.2}$$

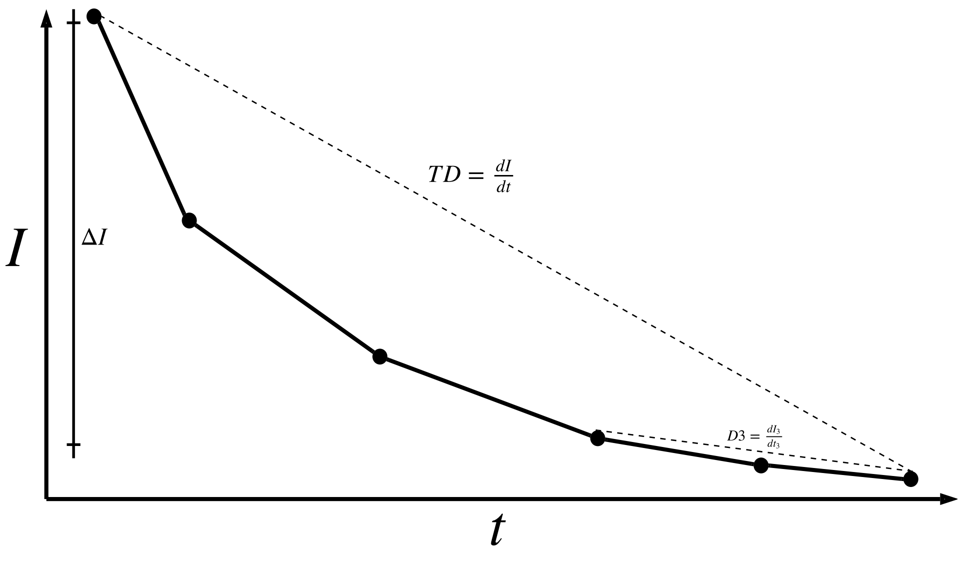


**Figure 8.4:** Definition of the different parameters used in Eq 8.2.

[4]Three points is a suggestion. The amount of points used for the "flattened out" derivative is a parameter which must be tuned.

### 8.2.3 Which cells to reprogram

A cell should be reprogrammed when the system is certain that the cell is going to fail based on the stored data. It is important to not program cells excessively or reprogram healthy cells since a flash cell has a limited number of program/erase cycles.

The reprogramming algorithm is based on the following observations

1. If a cell is below a critical value $T_{CRIT}$ it must be reprogrammed before an imminent failure.

2. If a cell is above a deemed ”safe” value $T_{ABS}$, it must not be reprogrammed.

3. If a cell is uncertain (below $T_{ABS}$ and above $T_{CRIT}$), it should be reprogrammed if it has a clear downwards trend (negative derivative) and the fluctuations are great enough to cause a failure.

4. A cell should not be reprogrammed if only a single data point gives a clear downward trend. That is if there is a measurement error, it should not be enough to trigger a reprogramming.

Based on these observations, the following algorithm is used to determine if a cell should be reprogrammed.

1. If number of data points is less than 3 $\Rightarrow$ STOP

2. If last data points value $\geq T_{ABS} \Rightarrow$ STOP

3. If last data points value $\leq T_{CRIT} \Rightarrow$ REPROGRAM & STOP

4. Calculate Total Derivative (TD) and Delta3 ($\Delta I_3$). $\Delta I_3$ is the highest possible delta between two of the last three points i.e. $\Delta I_3 = |max - min|$.

5. If TD $\leq T_{TD}$ & $\Delta I_3 \leq T_{DE3} \Rightarrow$ REPROGRAM

6. STOP

Figure 8.5 demonstrates the algorithm by showing three different cells. The green cell is over the $T_{ABS}$ and is never reprogrammed. The red cell is below $T_{CRIT}$ and is always reprogrammed. The black cell is reprogrammed based on $\Delta I_3$ and TD. This design leans heavily on $T_{TD}$ and $T_{DE3}$ being tuned to a good value. The $\Delta I_3$ metric with the corresponding condition is an implementation of the last observation.

The system keeps track of all cells that have been reprogrammed and how many times they have been reprogrammed. If a cell has been reprogrammed more than a set number of times, the system refuses to reprogram notifying the external system of the error. If the same cell repeatedly fails over and over again, the error cannot be corrected through reprogramming, raising the need for the chip to be replaced.

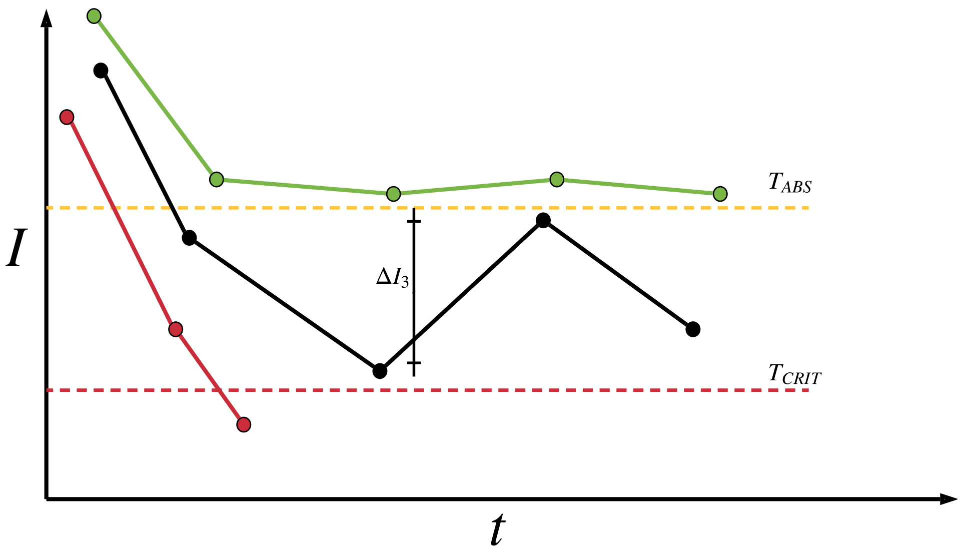


**Figure 8.5:** A graph illustrating reprogramming on three different cells. The green cell is over the $T_{ABS}$ and is never reprogrammed. The red cell is below $T_{CRIT}$ and is always reprogrammed. The black cell is reprogrammed based on $\Delta I_3$ and TD.

### 8.2.4 Removing data points

The amount of storage is finite, and some data points must be deleted to make place for newer data points. There are three types of deletes in the system.

- **Same Delete**: A new data point arrives, the system deletes a data point for the same cell to make room
- **Other Delete**: A new data point arrives, the system deletes a data point for another cell to make room
- **Prune Delete**: A pruning algorithm removes a data point.

As might be apparent from the previous sections, the more data points a cell has, the more accurate the systems representation of the flash is. It is, therefore, a need to remove data points which give redundant or no information at all. **Pruning** is the process of removing these redundant data points. The system has two pruning methods implemented, **prune cell simple** and **prune cell erratic**.

#### Prune Cell Simple

Prune Cell Simple searches for cells which have the same value consecutively over several data points and removes a number of them. Figure 8.6 illustrates this. When a cell has been found to have the same value for three points in a row, it deletes the middle point.

Please note that some information is lost with this pruning algorithm, but it relatively little information.

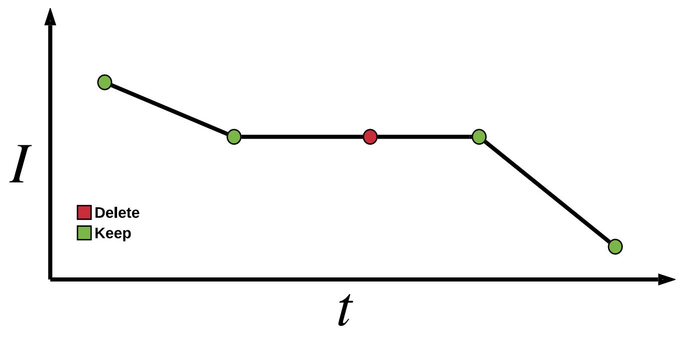


**Figure 8.6:** Illustration of the simple pruning algorithm. The red cell gives redundant information and is deleted.

### Prune Cell Erratic

Prune cell erratic evaluates if a cell is still worth monitoring. If a cell has a total derivative ($TD = \frac{dI}{dt}$) larger than a given threshold, and a desired time has passed ($\Delta T$), all data points for the cell is deleted. Figure 8.7 shows a cell which would be targeted by this algorithm.

Prune cell erratic is aimed to find two types of cells. It detects cells which had a trend towards failure, but recovered to a satisfactory degree. The second type of cells are the *erratic read* cells, which gives a massive measurement error for only one read. This type of cell is the one illustrated in Figure 8.7.

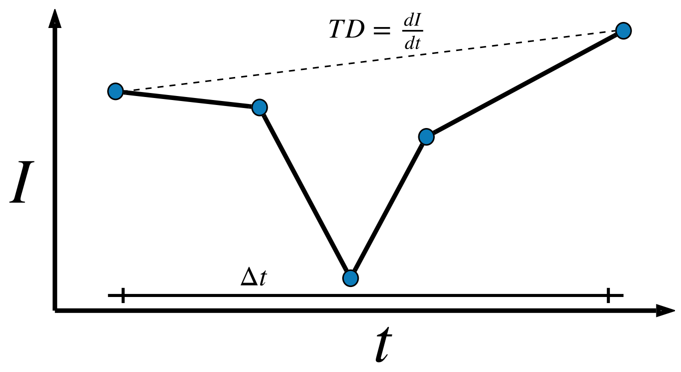


**Figure 8.7:** Illustration of the erratic pruning algorithm.

## 8.3 Storing data points

As discussed in the design chapter, care must be taken when storing data points as there is a limited amount of storage. As such, the design is optimized for storage, not performance.

All data points are stored in a linked list, where each data point points to the next data point for the same cell as can be seen in Figure 8.8. No link backward exists, only forward in order to save space.

Every data point the system stores is issued a ”cell index.” The cell index indicates the offset to the first data point in the linked list. For instance, in Figure 8.8, Cell 718’s cell index is 0 as the first data point is located at offset 0. Equivalently, the cell index for Cell 832 is 22.

If there were no cell index, every data point would have to store the cells number. There are $128 * 1024 * 8 = 1\ 048\ 576$ cells, which requires 20 bits to represent. By using such a cell index scheme, each data point only requires 32-bits.

- Time data point was measured [12 bits]
- Measurement value [12 bits]
- Offset to next data point [8 bits]

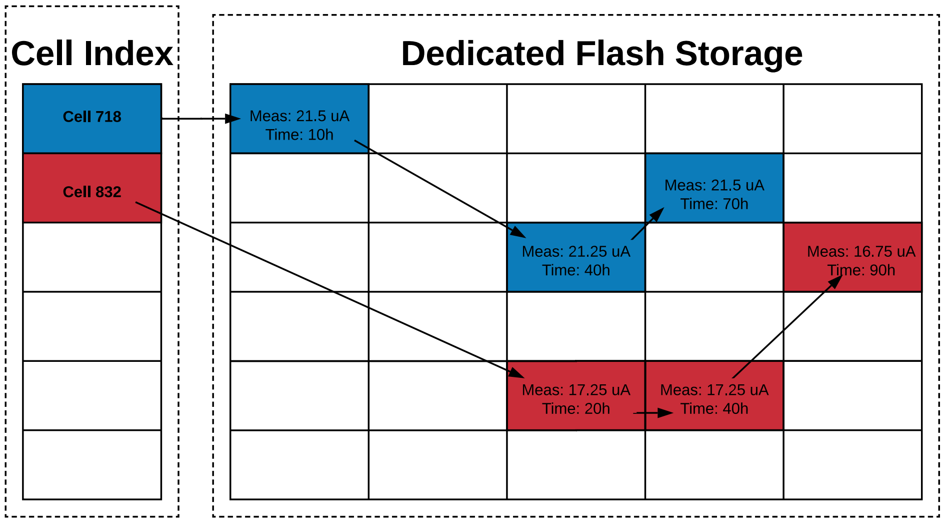


**Figure 8.8:** Linked list implementation of data point storage. Each ”block” can hold one data point.

### 8.3.1 Adding data points

In order to add a data point for a cell, two conditions must be satisfied.

1. There must be free space in the cell index
2. There must be a free block to store the data point

If both conditions are met, the data point is added to the first empty block. If the data point is the first for the cell, the cell is inserted into the cell index with the appropriate block offset. If there already exists a data point for the cell, the latest data points next pointer is updated. This procedure is shown in Figure 8.9.

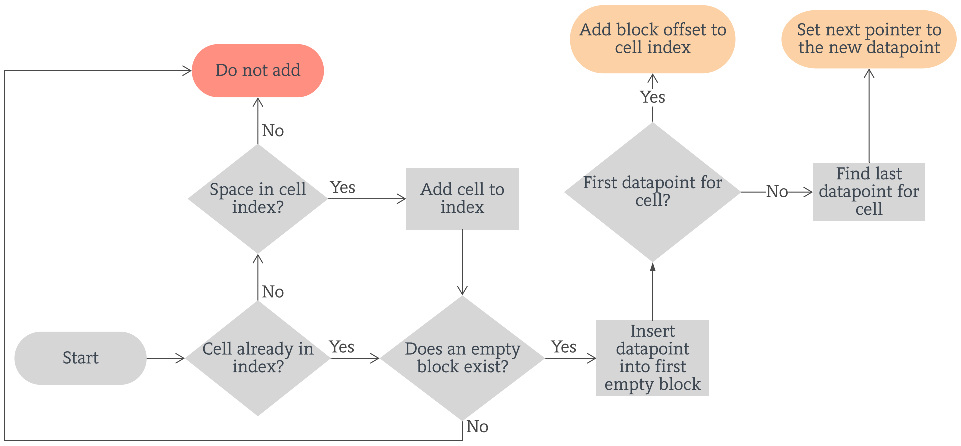

Add block offset to cell index
Set next pointer to the new datapoint
Do not add
Yes
No
Space in cell index?
Yes
Add cell to index
First datapoint for cell?
No
Find last datapoint for cell
No
Start
Cell already in index?
Yes
Does an empty block exist?
Yes
Insert datapoint into first empty block
No

**Figure 8.9:** Flowchart for adding a data point.

### 8.3.2 Deleting data points

A data point is deleted in a similar fashion to a conventional linked list. The following procedure deletes data point $x$ for cell $c$.

1. Iterate through the data points for $c$ until $x$ is found.
2. Set $x - 1$'s pointer to $x + 1$
3. Delete $x$
4. If the data point is the last for the cell, delete the cell from the cell index

## 8.4 Simulator interfacing - Network Interface

The current version of the prevention system uses the flash simulator described in Chapter 5 and Chapter 6. All communication with the simulator is done with JSON (Javascript Object Notation) through TCP (Transmission Control Protocol). The **simulator interface** handles all communications with the simulator, trying to mimic the same interface as found on a flash controller to a certain extent.

The simulator does not send any messages to the prevention system; the prevention system initiates all communications. The simulator therefor always responds to a command prepared by the prevention system. Figure 8.10 shows a flowchart for creating and sending these commands, before parsing and interpreting the results.

For instance, if the prevention system wants to know the cell measurement of cell 23, it sends a *measure cell 23* command. If the response has an ACK field with OK, it returns the measured value to the rest of the system.

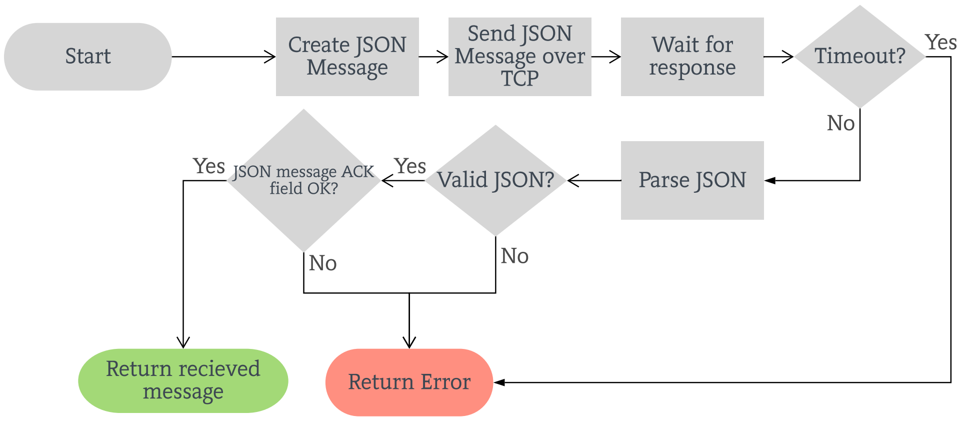


**Figure 8.10:** Flowchart for sending a command to the simulator and receiving the response.

## 8.5 Main Loop - Putting it all together

The prevention system is implemented with the touch screen controller mXT1067T in mind, which runs a dedicated firmware to register a touch on touch screens. The prevention system must, therefore, run concurrently with this firmware.

By this design, it is the firmware which chooses when the prevention system runs. As the time between each iteration of the system can vary a lot, it is important that the system does not rely on being run on set time intervals. For every iteration, the prevention system selects a handful of cells which are measured, stored, and evaluated as described in Chapter 8.2. The sequence diagram in Figure 8.11 shows one of these iterations.

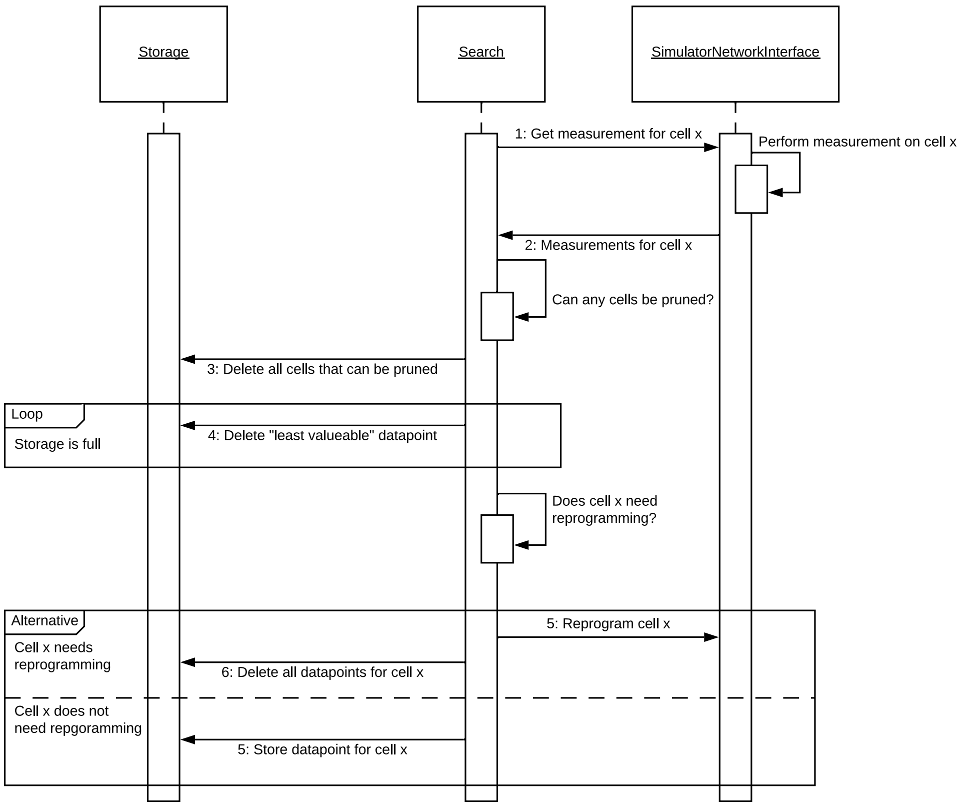

Storage
Search
SimulatorNetworkInterface
1: Get measurement for cell x
Perform measurement on cell x
2: Measurements for cell x
Can any cells be pruned?
3: Delete all cells that can be pruned
Loop
Storage is full
4: Delete "least valueable" datapoint
Does cell x need reprogramming?
Alternative
Cell x needs reprogramming
5: Reprogram cell x
6: Delete all datapoints for cell x
Cell x does not need repgoramming
5: Store datapoint for cell x

**Figure 8.11:** Sequence diagram for one iteration.

## 8.6 Control Panel & Datalog

The **control panel** module provides an interface for the user to control the prevention system as it is running. This module is primarily used for debugging and retrieving metrics. The module starts a thread running in parallel to the prevention system, listening for any user inputs such as ”start”, ”pause”, ”exit” and ”print state”.

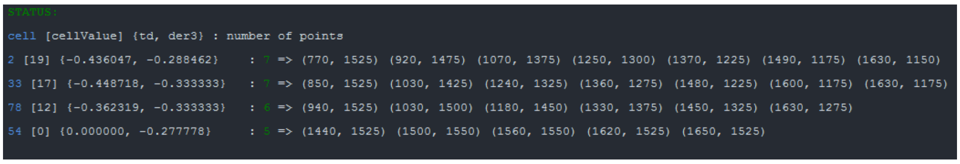
When a command has been entered, the command is passed to the prevention system thread, which subsequently executes it and prints any potential data. One command which require closer attention is the ”write state” command, which requests specific metrics to be stored to disk.

When a ”write state” command is initiated, the **datalog** module is used. It provides the functionality to log metrics and data during runtime. Some of these metrics are

- All logged data points for the current iteration
- How many data points and cells have been deleted so far

- How are the different cells and data points deleted

The data is stored in a CSV file, which can be used for later analysis. Figure 8.12 shows an example of the control panel. The command ”status” was issued, returning different metrics about the current state. In the example, the system reports four cells (2, 33, 78, 54) with the data points it has collected for them so far.



```
STATUS:
cell [cellValue] {td, der3} : number of points
2 [19] {-0.436047, -0.288462}    : 7 => (770, 1525) (920, 1475) (1070, 1375) (1250, 1300) (1370, 1225) (1490, 1175) (1630, 1150)
33 [17] {-0.448718, -0.333333}   : 7 => (850, 1525) (1030, 1425) (1240, 1325) (1360, 1275) (1480, 1225) (1600, 1175) (1630, 1175)
78 [12] {-0.362319, -0.333333}   : 6 => (940, 1525) (1030, 1500) (1180, 1450) (1330, 1375) (1450, 1325) (1630, 1275)
54 [0] {0.000000, -0.277778}     : 5 => (1440, 1525) (1500, 1550) (1560, 1550) (1620, 1525) (1650, 1525)
```

**Figure 8.12:** An example of the ”status” command issued to the control panel.

Chapter 9

# Machine Learning Approach

This chapter discusses a machine learning approach to the failing bit problems as an alternative to the error prevention system presented earlier. It does not present an implementation, only a starting point for a design with an accompanying discussion. The chapter proposes a network and how to obtain data for it. It does not go into any specific neural network details such as the number of layers, activation functions, hyperparameters, etc.

## 9.1 Introduction

In recent years, big data problems have been solved with excellent results using machine learning. There have been some attempts to introduce machine learning to increase the endurance of flash memories as presented in Chapter 3.2.2. However, the research done so far has been to identify cells that are worn out due to excessive amounts of program/erase cycles. This is of course very important when looking at flash memories in applications which programs and erases the flash frequently, as in a desktop computer SSD (Solid State Drive), but is not precisely what this thesis is studying.

The aim of this thesis is a system that can identify moving and erratic bits in applications with often only a single programming of firmware for the devices entire lifespan. In other words, the reason these bits fail is not due to the number of program/erase cycles. As the results from looking at write/erase cycles were so promising, it would be interesting to find out if such a machine learning algorithm would work on the moving and erratic bit problem.

## 9.2 Network

One of the big drawbacks of conventional neural networks is the time aspect. What happened before in a sequence has no impact on future predictions. The problem of predicting if a cell is going to fail based on its icell graph depends massively on time and is thus necessary information to provide to the network.

By introducing recursive neural network, as described in Chapter 2.3.3, the time aspect is learned by the internal state $h$. As with any neural network, the choice of input and output layer is important. Some parameters that might be used in the input layer are

- Cell current value
- Number of Write / Erase cycles
- Number of erratic reads observed
- Gate, source, drain and bulk [1] voltage
- Voltage on neighboring cells

Which parameters to choose for the input layer also depends on what information is available to the system at any given moment in time. Some parameters, such as the gate voltages are not available in the mXT1067T unless an external instrument is used, but from a research perspective might be interesting data to collect and use.

The network can be a single output such as the probability of failure, giving a many-to-one network, or several outputs giving a many-to-many network. Some output parameters that might be used are

- Probability of failure
- Cell current at time $t = t_{now} + \Delta t$

An example of such a network can be seen in Figure 9.1.

## 9.3 Collecting data

Machine learning requires a large amount of data to train the model, which is not always trivial to obtain. This is one of these cases. As discussed above, the approaches in Chapter 3.2.2 were designed to find a failing cell after a program or erase. This data is quite easy to obtain, as one can collect it through programming and erasing cells, doing measurements in between.

The data required for training on the cell current trend must be collected with time as a parameter, making the collection phase take much longer. All the parameters to the network must be measured at a given time interval and stored. As these errors (moving

[1] The BULK gate is a fourth gate located ”beneath” the transistor into the N-Substrate. It is not always included in illustrations.

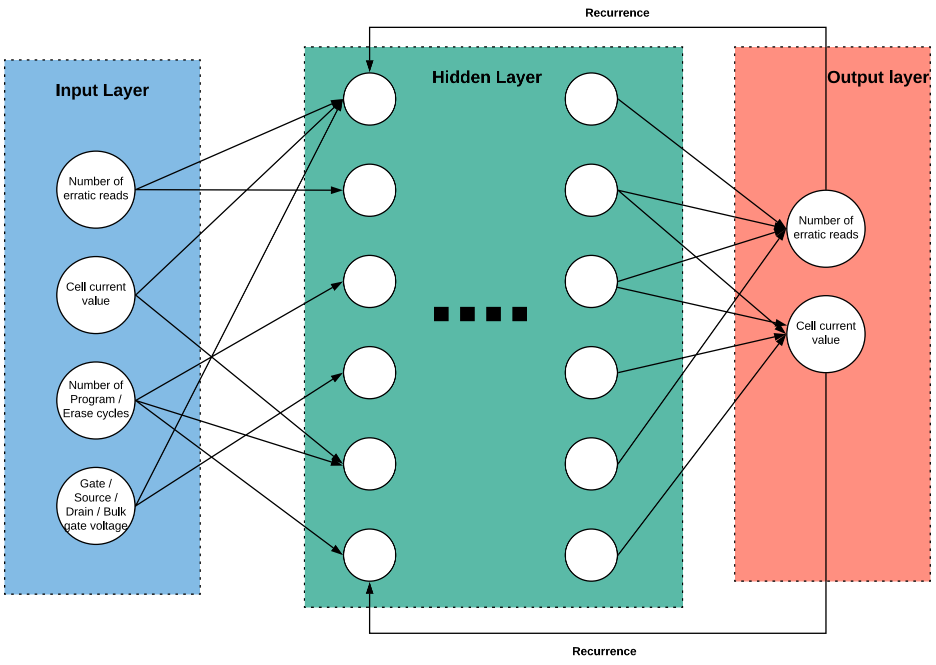


**Figure 9.1:** An example of an RNN network for the given problem. Number of nodes in the hidden layer is only for illustration.

and erratic bits) occur infrequently, the data collection stage would need to happen over a long period on of time on a large sample size of chips.

There might be ways to speed up the aging of the chip, reducing the time required to collect the data. This has been attempted on the mXT1067T with temperature aging, giving poor results [5]. This might be linked to the healing effects high temperature have on FGMOS [6]. There are other aging approaches such as high voltage that might give better results and would be a point for further work.

All of the discussion above assumes a collection in a lab, but there exist millions of Max-Touch chips in cars all over the world. If data could be collected from these through the built-in firmware, a large quantity of data could be collected at next to no cost. For instance, the internal measurement stored by the Prevention System described in this thesis could be valuable points of data. These could be read from the car every time the car is in the workshop.

More and more cars are connected to the internet at all times, and the increasing 5G mobile coverage gives massive bandwidth. This opens the window for collecting data and storing it in the cloud live, giving such a machine learning approach a neverending stream of training data and improve with time.

*The exploration of this machine learning approach complete acceptance criteria 8.*

Chapter 10

# Temperature effect on ”Fast Direct Memory Access”

This chapter presents an experiment conducted regarding the temperature effect on FDMA. A hypothesis, results, and conclusion are presented.

## 10.1 Introduction & Hypothesis

FDMA (Fast Direct Memory Access) is an internal testmode available on the mXT1067T chip from Microchip. It uses an internal test method to quickly measure the cell current of any cell in the flash. DMA (Direct Memory Access) is another internal test method which measures cell current, but it requires external instruments and is a direct readout (more accurate than FDMA). Temperature has been found to have a significant effect on the FDMA readout [5]. The following sections present an experiment conducted to test the following hypothesis

**Hypothesis:** The error introduced by changing the temperature on FDMA, $e_{temp}$, is linear and uniformly applied to the entire flash.

## 10.2 Experiment setup and methodology

The experiment was conducted in the same manner as [5], chapter 7.2. An mXT1067T chip was placed in a temperature chamber. With a $\Delta T = 15°C$, DMA and FDMA were measured on temperatures ranging from $-15°C$ to $105°C$. Two cells were measured on FDMA. One cell was measured on both FDMA and DMA. All parameters, except temperature, were kept constant throughout the experiment.

## 10.3 Results

The result from the cell with both DMA and FDMA readouts are presented in Table 10.1. The same data is plotted in Figure 10.1 in addition to the FDMA error ($e = |DMA - FDMA|$). Figure 10.2 shows the FDMA measurement for two cells over the various temperatures.

| **Temperature** | **DMA** | **FDMA** |
|---|---|---|
| -15 | 19.68 | 26.25 |
| 0 | 19.45 | 25.5 |
| 15 | 19.18 | 24.5 |
| 30 | 18.86 | 23.25 |
| 45 | 18.47 | 22.2 |
| 60 | 18.08 | 21 |
| 75 | 17.65 | 20.25 |
| 90 | 17.2 | 19.25 |
| 105 | 16.76 | 18.25 |

**Table 10.1:** DMA and FDMA measured on different temperatures.

## 10.4 Discussion and Conclusion

The data does not contradict the hypothesis. Both figures show a clear linear trend, and Figure 10.2 shows that the error is applied uniformly as they both have the same error. It is interesting that the error in Figure 10.1 is decreasing with increasing temperatures. This is caused by the derivative of the FDMA measurement to be lower than the DMA derivative.

The FDMA test mode uses an internal current generator to measure the cell current. This internal current generator might explain why the FDMA falls faster than the DMA, as the current generator is also affected by the temperature. This is backed up by the results in [5].

The sample size is quite small, with only two cells tested for uniformity, and only one chip was tested. This is not statistically significant, and further experiments are encouraged. It does, however, not disprove the hypothesis and strengthens it as such.

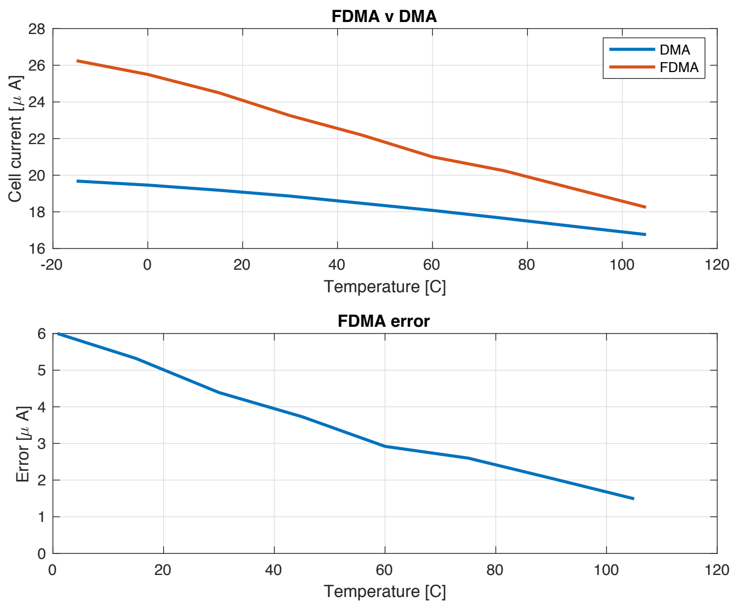


**Figure 10.1:** DMA and FDMA measured on different temperatures. The error ($e = |DMA - FDMA|$) is plotted for every temperature.

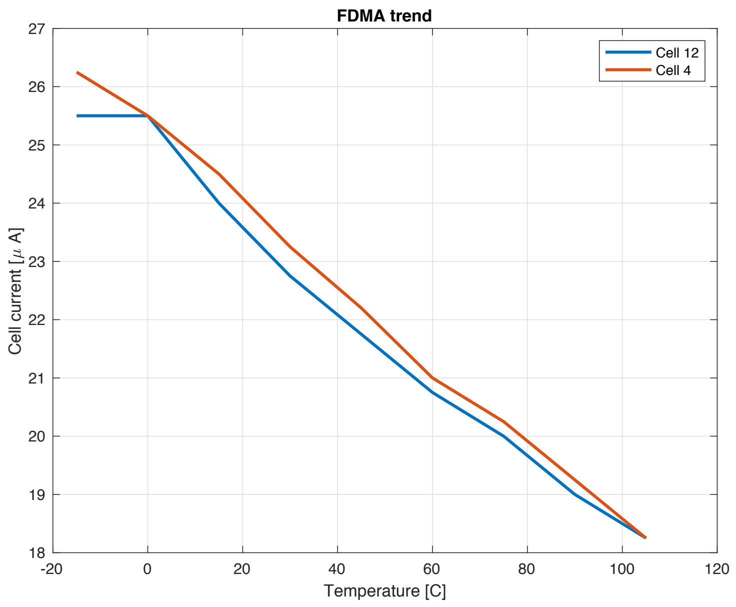


**Figure 10.2:** FDMA measured on different temperatures for two independent cells.

Chapter 11

# Testing

This chapter outlines the testing setup and methodology used to test the simulator and the error prevention system. Both systems are tested with unit tests and system tests. Both the simulator and the prevention system has been tested on a variety of tests. According to Sommerville [15], good practice is to perform *Unit Testing* and *System Testing*. Unit tests cover small units of the system, making bugs and errors easier to find and fix. System tests cover the entire system, testing specific use cases. The simulator has also been analyzed with a profiler called cProfiler.

## 11.1 Unit tests

Both the simulator and the prevention system has been tested through unit tests. The prevention system uses *Google Test*[1] and the simulator uses *PyTest*[2]. Continuous Integration (CI) through GitLab has also been used, making all the unit tests run for every git[3] commit. A list of all the unit tests follows. These unit tests are not necessary to follow the discussion in Chapter 13, but are included for completeness.

A common metric for how much of the code is covered by tests is code coverage.

$$\text{Code Coverage} = \frac{\text{Number of lines ran during tests}}{\text{Total number of lines}}$$

This metric has been measured for the simulator. Due to unforeseen errors with the code coverage analyzer for C (gcov), this metric has not been measured for the prevention system.

[1]https://github.com/google/googletest
[2]https://docs.pytest.org/en/latest/
[3]GIT is a version control system

### 11.1.1 Storage module

1. **addDatapoint**: Adds datapoints to the storage modules. Passes if all data points are added correctly
2. **deleteDatapoint**: Deletes two data points. Passes if the next pointers for the remaining data points are correct
3. **deleteCell**: Deletes two cells. Passes if all the data points for the given cells are deleted and if the cells are removed from the cell index
4. **ComplexOperationsSingleCell**: Adds and removes a number of data points in differing order. Passes if all the data points stored are correct.
5. **MultipleCellOperations**: Adds datapoints for two cells. Passes if all data points for both cells are added, and are separate.
6. **GetPrevPtr**: Adds datapoints for a single cell. Passes if the previous pointer algorithm returns the correct data point.
7. **GetNmbOfEmptyBlocks**: Adds a set number of data points. Passes if the system reports the correct number of empty blocks of storage.
8. **GetNmbOfCells**: Adds a set number of data points for two cells. Passes if the correct number of cells in the system is reported after one cell is deleted.
9. **LowestValuedDatapoint**: Adds a set number of data points for one cell. Passes if the system reports the data point with the lowest metavalue.
10. **LowestValuedCell**: Adds a set number of data points for two cells. Passes if the system reports the cell with the lowest metavalue.

### 11.1.2 Search module

1. **PruneCell**: Adds a set number of data points for one cell. Runs simple pruning twice. Passes if the correct data points were pruned
2. **Metrics**: Adds a set number of data points for one cell. Calculates total derivative, DER3, and absolute change. Passes if all returned metrics are correct
3. **CellValue**: Adds a set number of data points for one cell. Calculates the cells metavalue. Passes if the returned metavalue is correct.
4. **considerPointOverMax**: Adds a set number of data points for several cells in order to fill up the storage. When the storage is filled, it tries to add a new data point which should delete the ”worst” datapoint and add the new. Passes if the correct data point is deleted and the new data point is added.
5. **ReprogramEval**: Adds a set number of data points for one cell. Tries to reprogram. Passes if the cell is reprogrammed.

### 11.1.3 Simulator

1. **fiftyPercentIsLowCurrent**: Passes if half the cells initialised in the simulator are $0.5\mu A$.
2. **highCurrentGaussianDistributed**: Passes if all the non $0.5\mu A$ cells follow a Gaussian distribution.
3. **multipleSimulationsCanBeStarted**: Test several simulations at the same time. Passes if all simulations start and can iterate.

## 11.2 System Tests

System tests are used to evaluate the performance of the entire system. The simulator has been tested in isolation, using the software ”packet sender”[4] to interact with the simulators network interface. The simulator test consists of having it simulate all the different cell types (normal, moving, and erratic) over a significant number of iterations. As there is much more variety in the way a moving bit can behave in the simulator (compared to erratic and normal bits), it is simulated four times. The normal and erratic cells are both simulated once.

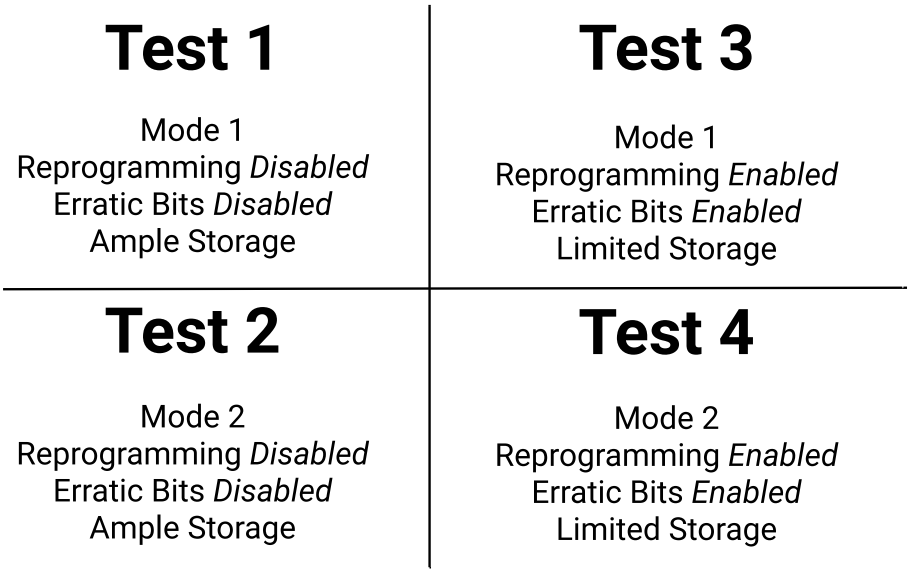
The prevention system requires the simulator to be tested. There are four tests conducted on the prevention system. Figure 11.1 shows all four tests in a table. Modes refer to which searching mode is enabled, see Chapter 8.2.2.

**Test one** is the simples of the four tests, having most features disabled, the simplest mode and ample storage. This test aims to test the core functionality of the system. For instance, how well do the stored data points track the actual cell current in ideal circumstances?

**Test two** is the same as test one, only changing the mode to the more advanced version. The primary goal of this test is to compare mode one vs. mode two in ideal conditions. Please note that neither test one or test two has reprogramming enabled, allowing moving bits to fall forever.

**Test three & four** has all the features of the prevention system, and the simulator enabled. There is also very limited storage. The only difference being that test three uses mode one and test four uses mode two. These tests are the ”extreme conditions” tests, as it not very likely conditions to occur in production. The system is also not intended to work with very limited storage. However, it is by testing the boundary of a system one often finds critical flaws.

[4]https://packetsender.com



| | |
|---|---|
| **Test 1**<br>Mode 1<br>Reprogramming *Disabled*<br>Erratic Bits *Disabled*<br>Ample Storage | **Test 3**<br>Mode 1<br>Reprogramming *Enabled*<br>Erratic Bits *Enabled*<br>Limited Storage |
| **Test 2**<br>Mode 2<br>Reprogramming *Disabled*<br>Erratic Bits *Disabled*<br>Ample Storage | **Test 4**<br>Mode 2<br>Reprogramming *Enabled*<br>Erratic Bits *Enabled*<br>Limited Storage |

**Figure 11.1:** The four tests conducted on the prevention system. Modes are constant on the rows, reprogramming and erratic bits are constant on the columns.

**Implementation detail:**

*For all tests* the following simulation parameters were used.

1. $\frac{1}{16}$ of the flash was simulated for every iteration.
2. The size of the flash was $S = 128KB$
3. Standard deviation of the flash was $\sigma = 0.9149$
4. The mean of the flash was $\mu = 20.68$
5. Normal bit simulation: $\Delta I = 0.25$, $P_p = P_n = 0.1$. $P_{initial} = 0.1$.
6. Four moving bits
7. One erratic bit if applicable to the test
8. *For moving bits:* $b$ was chosen randomly from a normal distribution with $\sigma = -0.000476$ and $\mu = 0.0001$.
9. *For moving bits:* $c$ was chosen randomly from a normal distribution with $\sigma = 6$ and $\mu = 4$

The following prevention system parameters were used

1. $I_w = 15.5\mu A$
2. $W_I = 1.0$, $W_t = 1$, $W_D = 300$
3. $W_\Delta = 0.05$, $W_{td} = 0$, $W_{d3} = 1$
4. $T_{ABS} = 13.5\mu A, T_{CRIT} = 11.5\mu A, T_{TD} = -0.2, T_{DE3} = 100$
5. The system has $25 * 32 = 800$ bytes on ample storage.
6. The system has $15 * 32 = 480$ bytes on limited storage.

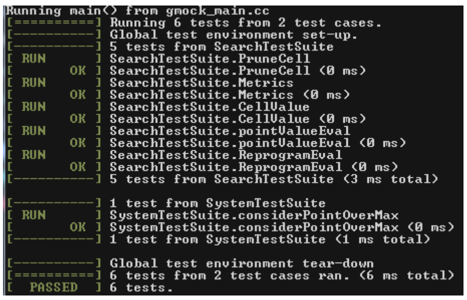
# Chapter 12

# Results

This chapter presents the results from the tests outlined in Chapter 11. This includes the results of the unit tests and system tests, a performance analysis of the simulator and accompanying plots.

## 12.1 Unit Tests

All the unit tests described Chapter 11 has been run. The results from prevention system unit tests are shown in Figure 12.1. All the unit tests passed for both the simulator and the prevention system. The simulator had a code coverage of about 69%.

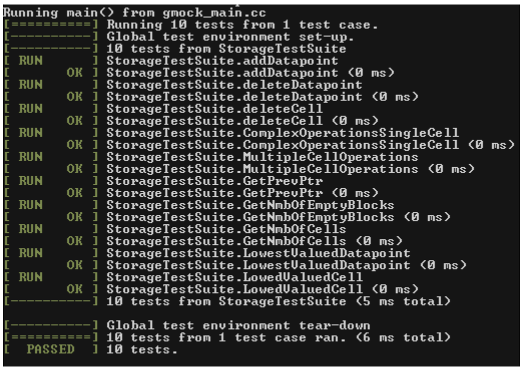


```
Running main() from gmock_main.cc
[==========] Running 6 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 5 tests from SearchTestSuite
[ RUN      ] SearchTestSuite.PruneCell
[       OK ] SearchTestSuite.PruneCell (0 ms)
[ RUN      ] SearchTestSuite.Metrics
[       OK ] SearchTestSuite.Metrics (0 ms)
[ RUN      ] SearchTestSuite.CellValue
[       OK ] SearchTestSuite.CellValue (0 ms)
[ RUN      ] SearchTestSuite.pointValueEval
[       OK ] SearchTestSuite.pointValueEval (0 ms)
[ RUN      ] SearchTestSuite.ReprogramEval
[       OK ] SearchTestSuite.ReprogramEval (0 ms)
[----------] 5 tests from SearchTestSuite (3 ms total)

[----------] 1 test from SystemTestSuite
[ RUN      ] SystemTestSuite.considerPointOverMax
[       OK ] SystemTestSuite.considerPointOverMax (0 ms)
[----------] 1 test from SystemTestSuite (1 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 2 test cases ran. (6 ms total)
[  PASSED  ] 6 tests.
```

**(a)** Results from running the search module unittests.



```
Running main() from gmock_main.cc
[==========] Running 10 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 10 tests from StorageTestSuite
[ RUN      ] StorageTestSuite.addDatapoint
[       OK ] StorageTestSuite.addDatapoint (0 ms)
[ RUN      ] StorageTestSuite.deleteDatapoint
[       OK ] StorageTestSuite.deleteDatapoint (0 ms)
[ RUN      ] StorageTestSuite.deleteCell
[       OK ] StorageTestSuite.deleteCell (0 ms)
[ RUN      ] StorageTestSuite.ComplexOperationsSingleCell
[       OK ] StorageTestSuite.ComplexOperationsSingleCell (0 ms)
[ RUN      ] StorageTestSuite.MultipleCellOperations
[       OK ] StorageTestSuite.MultipleCellOperations (0 ms)
[ RUN      ] StorageTestSuite.GetPrevPtr
[       OK ] StorageTestSuite.GetPrevPtr (0 ms)
[ RUN      ] StorageTestSuite.GetNmbOfEmptyBlocks
[       OK ] StorageTestSuite.GetNmbOfEmptyBlocks (0 ms)
[ RUN      ] StorageTestSuite.GetNmbOfCells
[       OK ] StorageTestSuite.GetNmbOfCells (0 ms)
[ RUN      ] StorageTestSuite.LowestValuedDatapoint
[       OK ] StorageTestSuite.LowestValuedDatapoint (0 ms)
[ RUN      ] StorageTestSuite.LowedValuedCell
[       OK ] StorageTestSuite.LowedValuedCell (0 ms)
[----------] 10 tests from StorageTestSuite (5 ms total)

[----------] Global test environment tear-down
[==========] 10 tests from 1 test case ran. (6 ms total)
[  PASSED  ] 10 tests.
```

**(b)** Results from running the storage module unittests.

**Figure 12.1:** Results from unit tests on the prevention system.

## 12.2 Simulator System Test

The simulator was tested with the system tests described in Chapter 11. Figure 12.2 shows four simulated moving bits. They all follow a decaying exponential curve with different derivatives and constants. Figure 12.3 shows a simulated erratic bit. Figure 12.4 shows the simulation of a normal bit, fluctuating slowly over time.

The erratic bit shows sudden drastic jumps occurring randomly over time. If those jumps are removed, one observes the same pattern as the normal bit. The simulation was performed with a total of four moving bits, which the prevention system also picked up. However, these have not been included in the figures to avoid clutter as they show the same result.

The simulator uses around 1.5 seconds per iteration on a high-end Dell Laptop from 2013. A profile of the simulator using cProfiler is shown in Figure 12.5. The methods of interest do the following

- **progressTimeOverFewCells** performs one iteration on a random subset of the cells.
- **digitize** is a numpy function that digitizes the values into buckets. For instance 22.83 becomes 22.75.
- **calculateParts** calculates the next iterations for *normal cells*
- **digitizeValue** calls digitize for a given value, and returns the appropriate buckets

Most of the time the simulator calculates the next iteration, with a huge chunk of the time going to digitizing the values. This was tested by removing the digitize method from the simulator, gaining 0.77 seconds per iteration, almost doubling the performance.

## 12.3 Prevention System Test

### 12.3.1 Test one & two

Test one and two is the same test, only differing in which searching mode is enabled. The tests ran for 2000 iterations each with the parameters given in Chapter 11.2. The main goal of these test is to evaluate how the prevention system data points tracked the true value from the simulator, and how the different modes compare in these ideal conditions.

Figure 12.6 shows which points the prevention system has stored at iteration 1200 & 1900 for **test one**, and the true value from the simulator. For both iterations, the curve is followed to an almost perfect extent. There are more data points for both cells at iteration 1200, whereas at iteration 1900 the data points are more spread out and further between.

Figure 12.7 shows the same figure for **test two**. The result is almost identical, following the true value very accurately. The data points are evenly spread out for all iterations for all cells. That is, there is no cell that has a significantly higher amount of cells than the other cells.

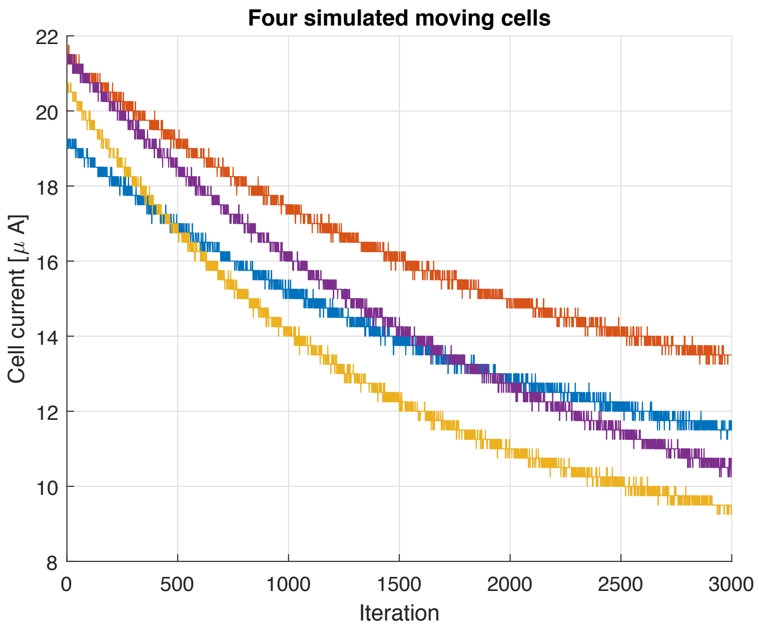


**Figure 12.2:** Four simulated moving cells. All four bits follow a decaying exponential curve.

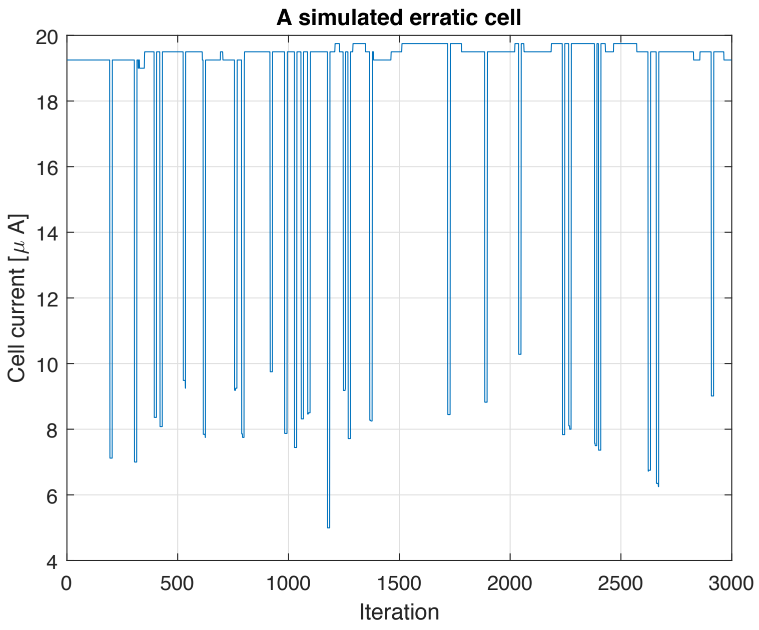


**Figure 12.3:** A simulated erratic cell. Shows sudden jumps in the cell current levels, otherwise behaving like a normal cell.

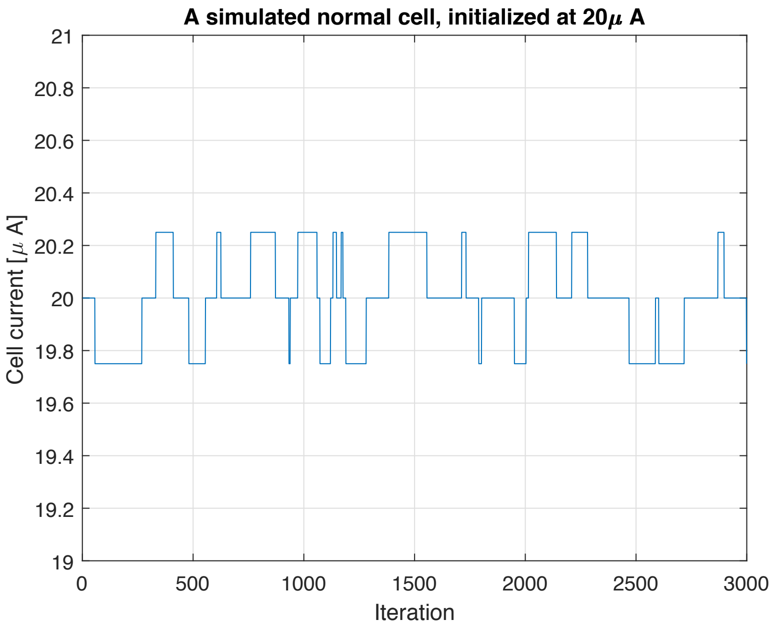
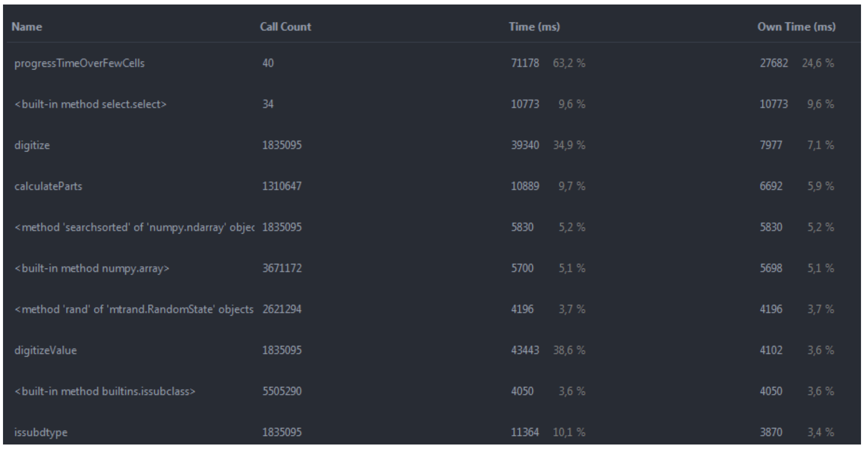
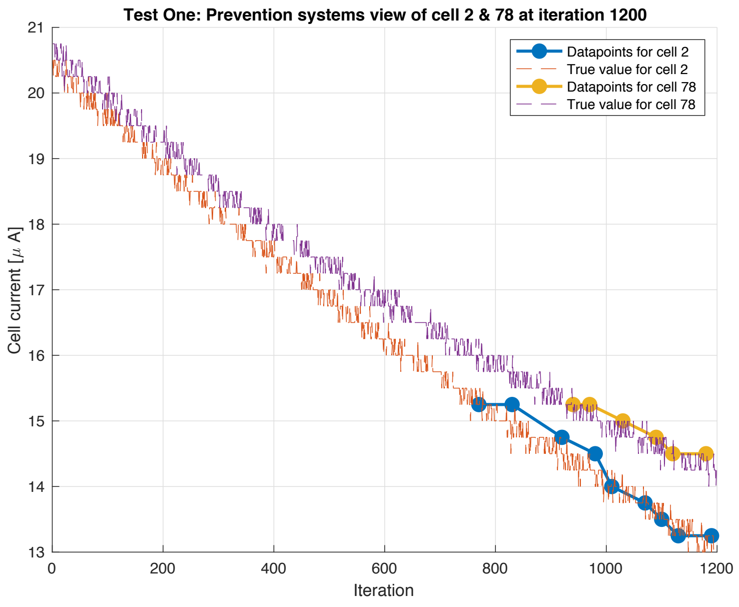


**Figure 12.4:** A simulated normal bit. Cell current is more or less constant, shows minor fluctuations.

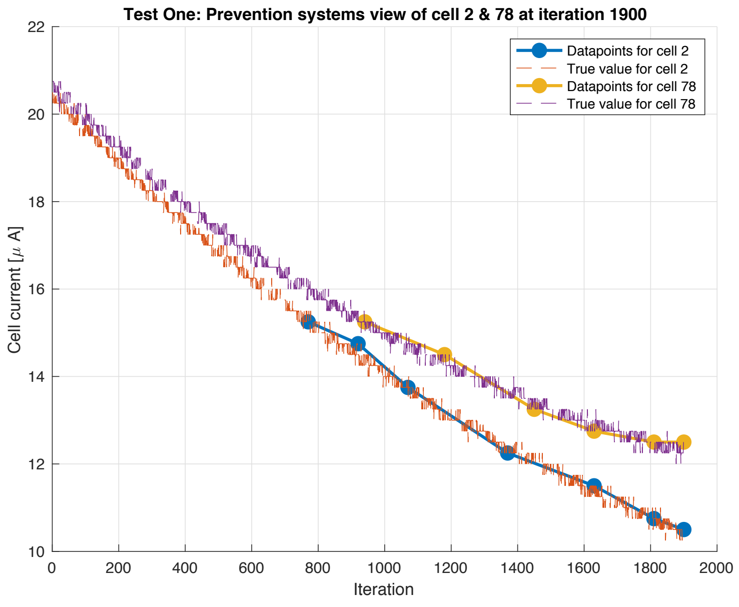


| Name | Call Count | Time (ms) | | Own Time (ms) | |
|---|---|---|---|---|---|
| progressTimeOverFewCells | 40 | 71178 | 63,2 % | 27682 | 24,6 % |
| <built-in method select.select> | 34 | 10773 | 9,6 % | 10773 | 9,6 % |
| digitize | 1835095 | 39340 | 34,9 % | 7977 | 7,1 % |
| calculateParts | 1310647 | 10889 | 9,7 % | 6692 | 5,9 % |
| <method 'searchsorted' of 'numpy.ndarray' objec | 1835095 | 5830 | 5,2 % | 5830 | 5,2 % |
| <built-in method numpy.array> | 3671172 | 5700 | 5,1 % | 5698 | 5,1 % |
| <method 'rand' of 'mtrand.RandomState' objects | 2621294 | 4196 | 3,7 % | 4196 | 3,7 % |
| digitizeValue | 1835095 | 43443 | 38,6 % | 4102 | 3,6 % |
| <built-in method builtins.issubclass> | 5505290 | 4050 | 3,6 % | 4050 | 3,6 % |
| issubdtype | 1835095 | 11364 | 10,1 % | 3870 | 3,4 % |

**Figure 12.5:** Profiling results of the simulator. Time is the real-time the method used. Own time is the time used by the method subtracted the time used by called methods.

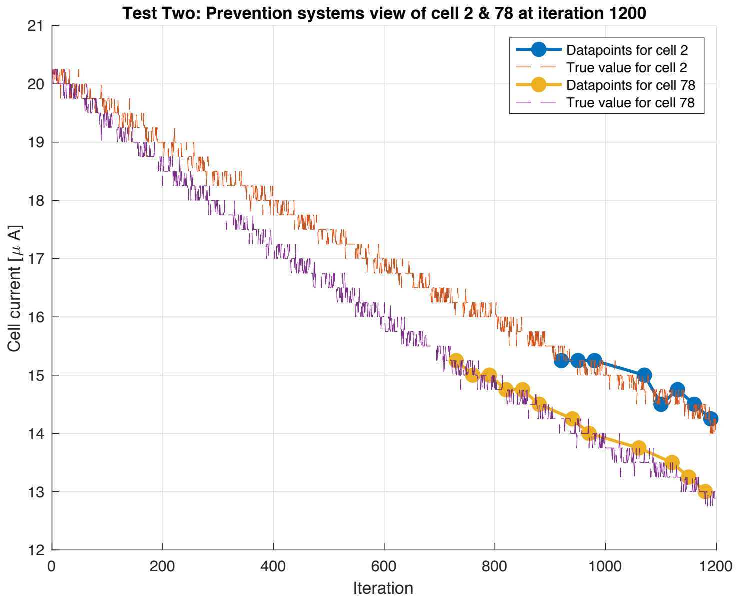


**(a)** Prevention system view of cell 2 & 78 at iteration 1200.

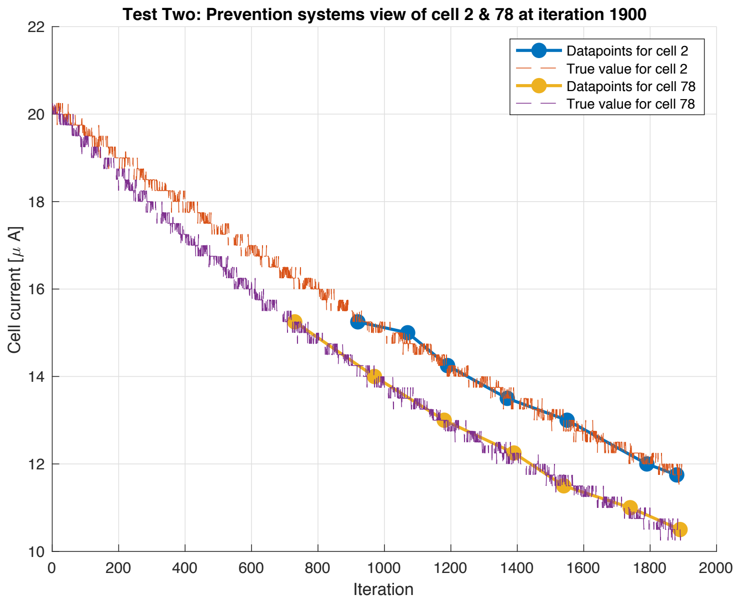


**(b)** Prevention system view of cell 2 & 78 at iteration 1900.

**Figure 12.6: Test one**: Prevention system view of cell 2 & 78 at iteration 1200 and 1900. The dots are the data points the prevention system has stored, the dotted lines is the ”true” value, which the prevention system tries to follow.



**(a)** Prevention system view of cell 2 & 78 at iteration 1200.



**(b)** Prevention system view of cell 2 & 78 at iteration 1900.

**Figure 12.7: Test two**: Prevention system view of cell 2 & 78 at iteration 1200 and 1900. The dots are the datapoints the prevention system has stored, the dotted lines are the "true" values, which the prevention system tries to follow.

### 12.3.2 Test three & four

Test three and four introduced reprogramming and erratic bits. In test three, all moving bits were detected and reprogrammed when they reached their critical levels, as shown in Figure 12.8. No cell was reprogrammed incorrectly. The erratic bits were also logged, as seen in Figure 12.9. The erratic pruning algorithm removed these logged erratic bits after about 200 iterations where no erratic behavior was observed.

Test four showed much of the same behavior as test three, except for how the data points were distributed among the cells. In test four (mode two), some cells are unable to obtain more than one data point, even though the real trend warrants that the cell should be monitored. This problem is labeled **trashing**, and is shown in Figure 12.10. Cell two is unable to get more than one data point before getting removed, while cell 54 and cell 78 have points to spare.

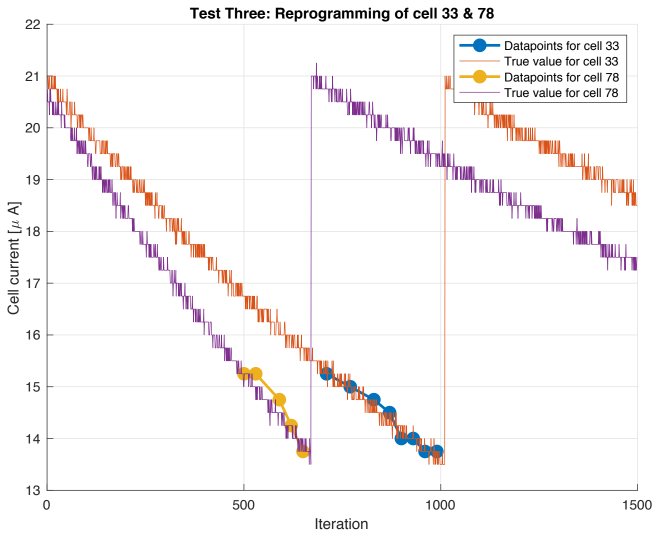


**Figure 12.8: Test Three:** Prevention system view of two cells being reprogrammed. Yellow and blue dots are the datapoints from the prevention system, dotted lines are from the simulator.

### 12.3.3 Deletion of cells & datapoints

The system has three types of deletes as described in Chapter 8.2.4, *Pruning, Same cell and other cell*. Another test with the same parameters as Test Four was conducted, logging every delete. The results is shown in Figure 12.11, showing the number of deletes with the corresponding trend. Most deletes were of the sort ”Same Delete”, with pruning picking up after around 2500 iterations.

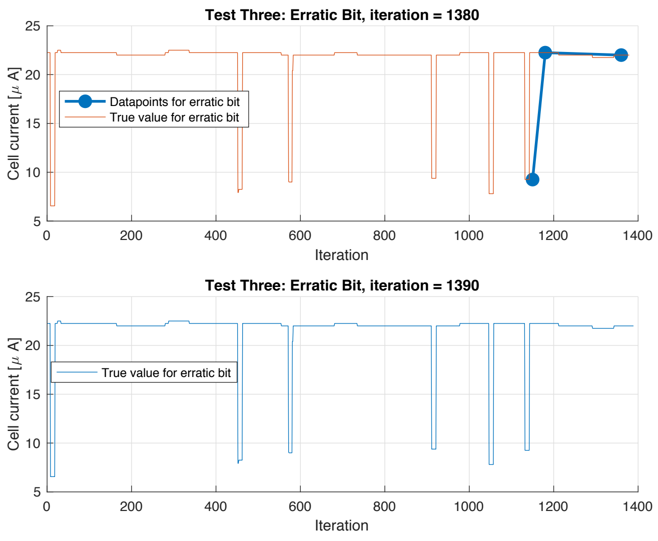


**Figure 12.9: Test Three:** Prevention system view of an erratic cell for iteration 1380 and 1390. Erratic prune removed all data points from 1380 to 1390.

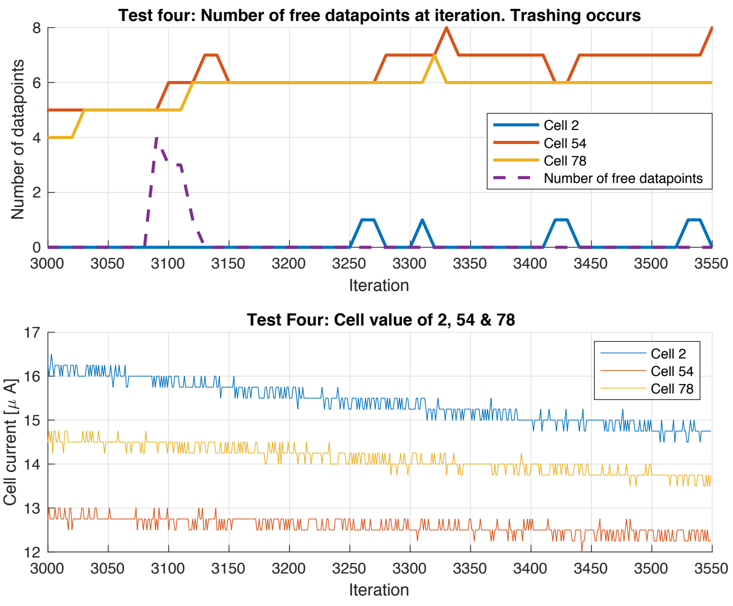


**Figure 12.10: Test Four:** Evidence of trashing. Cell 2 does not get to keep any of its data points, as they are discarded immediately.

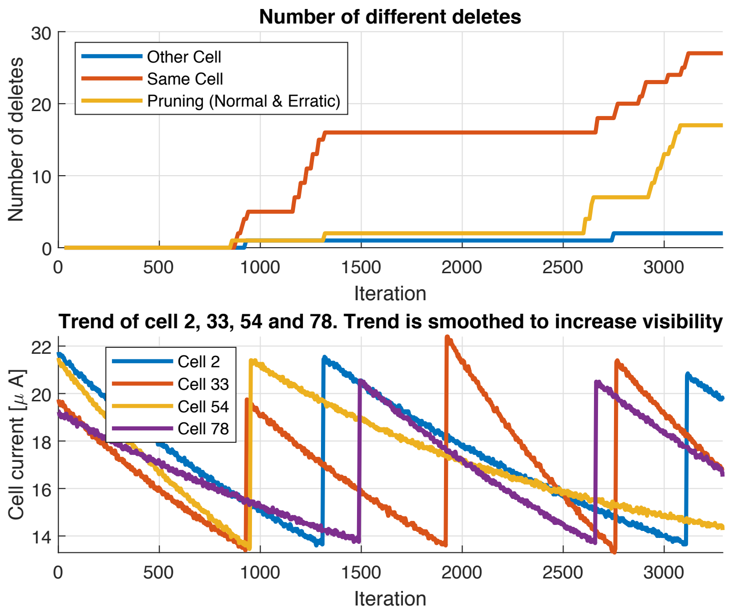


**Figure 12.11:** The amount of different deletes performed plotted against the real trend of all the moving cells in the system.

Chapter 13

# Discussion

This chapter covers a discussion of the results. The simulator is discussed in terms of behavior and performance. The main bulk of the chapter covers the error prevention system. The discussion looks at how well it dealt with the faulty bits from the simulator, looking at the underlying reason for the behavior. The viability and performance of the system is evaluated. Mode one and two are compared against another, and the problem trashing is discussed with proposed solutions.

## 13.1 Simulator

The simulator works as anticipated when compared to the specifications for all the three types of cells. The normal cells behaved as expected, showing small fluctuations. The main intention of including the normal cells was to introduce movement and unpredictability. If those were not included, the prevention system could look for any movement, and mark those as faulty bits.

*The simulator simulates normal, erratic, and moving cells. Thus completing acceptance criteria 1*

### 13.1.1 Moving Bits

The moving bits are decaying exponentially as the Multi-Phonon-Assisted tunneling model suggests. As the $b$ and $c$ parameters[1] are taken from real measurements[5], they mimic reality quite well.

*This completes acceptance criteria 2.*

[1] $I(t) = ae^{bt}$

All the moving bits start to decay at the very start. That is, there is no window of time they act as normal bits before they behave like moving bits. This also applies when they are reprogrammed, as seen in Figure 12.8. This is not necessarily accurate, but as they have a different derivative, it should not affect the results from the error prevention system to any significant extent. That is, the different moving bits hit the sweeping threshold ($I_w$) at different times.

### 13.1.2 Erratic Bits

The erratic bits are trying to mimic the erratic bits found in flash memories, and has been achieved to a certain extent, as the simulated erratic bit from Figure 12.3 matches the desired behavior presented in Figure 6.5.

As with moving bits, certain inaccuracies must be addressed. The simulated erratic bit from Figure 12.3 regains its previous value after a while. A real erratic bit would most likely not get a higher measurement value before reprogramming.

Real erratic bits are almost impossible to detect, and would often immediately fail when surfacing. The prevention system would, therefore, not be a viable solution to find and correct these bits. The simulated erratic bits are used to test if whether or not the system can deal with measurement errors, and not to detect and recover erratic bits.

One might argue that the simulated erratic bits are too aggressive, dropping to a very low value quite frequently as seen in Figure 12.3. This is intended, as it tests the prevention systems ability to rid itself of false positives.

### 13.1.3 Improving Performance

The performance of the simulator could be improved quite a bit. As Figure 12.5 shows, most of the time is spent on digitizing the values. When removing the digitization, the performance increased twofold. This digitization is strictly speaking only required on erratic and moving cells, as the normal cells are already calculated in a digitized manner (they increase and decrease in 0.25 intervals). As the vast majority of the calculations are normal cells, this could effectively remove the performance impact of digitization.

When calculating a new iteration of normal cells, a subset of the cells is selected for simulation with a probability of $p_{selected}$. All the selected cells have a probability of $p_{change}$ to change their value. In other words, any given cell has a probability of $p_T = p_{selected}p_{change}$ to change its value in the next iteration. Performance could be increased by using $P_T$ directly to select cells, and changing the value of the selected cells, reducing the number of calls to numpy drastically. [2]

*This discussion and evaluation of the performance complete acceptance criteria 3*

[2]This approach only works if the probability of increasing is equal to the probability of decreasing. If these are different, one would need to select two subsets.

## 13.2 Error Prevention System

The error prevention system manages to track, follow, and reprogram all cells which moved. The two cells in Figure 12.8 are both represented accurately with the data points. No data points are logged before the real value goes below $15.5\mu A$, as the sweeping watch parameter is $I_w = 15.5\mu A$. No cell can be reprogrammed before having a lower value than $T_{ABS} = 13.5\mu A$, which is why both cells are immediately reprogrammed when they reach $13.5\mu A$. The two other criteria for reprogramming ($T_{TD}$& $T_{DE3}$) are met before the cells reach $T_{ABS}$.

*This completes acceptance criteria 4*

The spacing in time between the data points increased as time passed, spreading the data points evenly out except a small bias towards the newest and oldest points as seen in Figure 12.6 and 12.7. This shows that the pruning algorithm and data point metavalue works as intended. The data point metavalues last term, $W_D(\frac{t}{t_{first}+t_{last}} - 0.5)^2$, is the main reason for the bias towards a higher data point density at the start and end. The other terms provide the spacing between the values.

As seen in Figure 12.11, the majority of deletions are of the same cell. The pruning algorithm seems not to remove many data points until later iterations. As cells are reprogrammed, the amount of pruning seems to increase. This is odd, as the pruning algorithms run on each cell with no overlap. Further investigation is required to find the root cause.

*This discussion on types of deletes completes acceptance criteria 5.*

### 13.2.1 Mode one & two

The two modes had almost identical performance when there was ample storage as seen in Figure 12.6 and 12.7. As there is ample storage, both cells can obtain many data points. As such, the system does not need to prioritize any cell, effectively removing the difference between the two modes.

When there is limited storage, the difference between the modes becomes clear. Mode one prioritizes the cells on the number of data points (fewer data points, higher priority), while mode two uses the cell metavalue from Equation 8.2. There is no clear evidence of higher performance for mode two, as the value of new data points has a diminishing return. Adding the sixth data point generally gives less information than adding the second.

Mode two has one clear disadvantage compared to mode one, the **trashing problem**, shown in Figure 12.10. The trashing problem only occurs when the storage is full. When a new cell falls below $I_w$, it obtains one data point before being deleted shortly after. This happens as cells with only one data point has a metavalue of zero. A cell with only one data point has no $\Delta I$ and a derivative of zero, making Equation 8.2 compute to zero as all the terms are trivial. A metavalue of zero means it is very likely the lowest valued cell, thus being deleted when a new data point arrives.

This problem could be attempted mitigated by several possible solutions.

1. Add a new term to Equation 8.2 representing the amount of data points a cell already has, $W4 \frac{1}{\text{Number of datapoints}}$, reducing its metavalue if there are many data points already. This would effectively prioritize cells with few data points.
2. Let all new cells obtain at least $n$ data points before the cell can be deleted. This allows all terms in Equation 8.2 to be non-trivial.
3. If there are very few data points, the metavalue could return a non-zero default value making it a higher priority.
4. Make the cell metavalue depend on other factors than $\Delta I$, $TD$ and $D3$.

*The first solution* seems the most appealing at first, but it does not remove the problem. The weights of the different terms must be very finely tuned, making it vulnerable to edge cases not taken into consideration. As there exist billions of these chips, there are sure to be edge cases. In other words, it would not be stable enough for production. If the weight $W_4 \rightarrow \inf$, the data points would be spread evenly among all the cells, which is what mode one already does.

*The second solution* might be the most stable. It would ensure the system has enough information before making a decision. Such a solution would require extra storage to store these data points, as there is no storage left (the system does not delete cells when there is free space). A decision must then be made on what to do if this additional storage gets full.

*The third solution* is the simplest, but would not remove the problem entirely. In essence, it creates a new zero point, allowing for ”negative” values. This default value needs to be finely tuned, making it have the same problems as the first solution and would not be stable enough for edge cases.

*The fourth solution* is simply a full redesign of the cell metavalue, and is not a topic for this chapter.

*This evaluation and discussion of the two modes complete acceptance criteria 6.*

### 13.2.2 System Performance & Viability

The general performance of the system has been adequate. All the unit tests passed, indicating healthy modules. After running simulations for several hours, the prevention system did not fail once, and the storage used is minimal. There exists no metrics for how fast the system performs, as the vast majority of the time is waiting for the simulator.

The biggest concern is the size of the binaries. After stripping away every component that could not be ported to AVR32 (network interface, control panel, datalog, etc.), it ended up at a size of $14KB$. The size of the data points is $1KB$[3], making the total size of the error prevention system $S_{PS} = 15KB$. This number is bound to change in an actual AVR32 port, most likely decreasing with more optimization introduced and removal of unused code.

[3]The tests used 480 bytes on limited and 800 bytes on ample storage.

For this system to be viable, the binaries and the stored data must be less than the equivalent ECC implementation with corresponding codes. ECC codes are stored for blocks of a given size. Larger blocks require less storage for ECC but take longer to calculate. Smaller block sizes require more storage but are faster to calculate. Chips similar to the $mXT1067T$ therefor use about $11\%$ of the total flash for ECC codes[8].

The mXT1067T has $128KB$ of dedicated flash memory, which gives $14KB$ of ECC codes. The size of the ECC implementation binaries are not known before an actual implementation is made and measured, but an estimation of $5KB$ is used for the sake of discussion. The total size for an ECC solution ($S_{ECC}$) is shown in Equation 13.1.

$$S_{ECC} = S_{codes} + S_{Implementation} = 0.11 * S_{flash} + S_{Implementation} \tag{13.1}$$

By putting Equation 13.1 equal to the size of the prevention system binaries ($S_{PS}$), the minimum size of the flash for this solution to be viable is obtained.

$$\begin{aligned} S_{ECC} &= 0.11 S_{flash} + S_{Implementation} = S_{PS} \\ S_{flash} &= \frac{S_{PS} - S_{Implementation}}{0.11} \end{aligned} \tag{13.2}$$

Given $S_{implementation} = 5KB$ and $S_{PS} = 15KB$ the flash size must be a minimum of $S_{flash} = \frac{15-5}{0.11} = 90.9KB \approx 91KB$. In reality, the size should be quite a bit larger than $91KB$, as ECC codes provide much better coverage than the prevention system[4]. The prevention system does all the operations when the system is idling, thus not impacting the apparent performance of the system, while ECC must be calculated for every read and write operation from the flash. This thesis has no metrics for how much of a performance impact ECC has on the flash, and must be investigated further.

The size of the error prevention system binaries is a lot larger than the size of the stored data points. A big chunk of the code is the storage module, implementing the linked list optimized for storage. If the size of the binaries implementing this code is larger than the performance gain from the decreased amount of storage required for the data points, it counteracts its purpose. Further testing on this might yield lower memory requirements.

The error prevention system is tested against a simulator and not an actual microcontroller. Even though the simulator is based on results gained from earlier experiments, some doubt must be shed on the results as they are obtained through a simulator with a simple model. The work and results presented are the first step towards a working system, but emphasis must be put on the importance of implementing and testing on a real system before reaching any major conclusions.

*This evaluation of the viability completes acceptance criteria 7*

[4]ECC covers both moving and erratic bits, the prevention system only covers moving bits

Chapter 14

# Conclusion

In this thesis, a proof of concept error prevention system has been developed to find, track, and reprogram failing flash cells using their cell current. To assist the development and evaluation of the system, a simulator for flash memory has been developed. The simulator uses statistical properties from data collected in an earlier project to simulate moving, erratic, and normal bits.

The normal cells followed a simple model, increasing and decreasing with a set probability as one would expect actual normal cells to behave. The simulated moving cells followed the multi-phonon assisted tunneling model, decaying before stabilizing. The erratic bits are resembling real erratic bits to a certain extent, with key limitations. However, their main purpose is to provide measurement errors to the prevention system.

The performance of the simulator was evaluated to $1.5$ seconds per iteration on a specific system. After a simple analysis, it was discovered that the digitization of values was a bottleneck. Removing the digitization increased performance twofold.

The prevention system is built as a proof of concept on the x86 platform with a port to AVR32 in mind. It managed to find, track, and reprogram all the cells produced by the simulator. All data points were spread out evenly over both time and cells, utilizing the available storage to a satisfactory extent.

In order for the prevention system to be a viable alternative to ECC, the flash size must be a minimum of $S_{flash} = 91KB$ with the numbers obtained. At $S_{flash} = 91KB$, the prevention system uses the same amount of storage as ECC. The prevention system is not able to prevent erratic cells from failing while ECC can prevent these failures. The prevention system only runs when the system idles, having less of a performance impact on the rest of the system compared to ECC.

Two modes are present in the prevention system. The first mode distributes the data point evenly among all cells. The other mode tries to give more data points to more likely to fail cells. Both modes showed comparable performance when there was ample storage. Mode two suffers from a *trashing problem* when there is limited storage. New cells data points are deleted immediately, as the system evaluates the cell as not likely to fail. Several solutions to this problem have been proposed, none implemented or tested.

In conclusion, this thesis has shown that it is likely possible to use cell current for flash cells to detect and prevent different error modes propagating into failures. The given system can be ported over to AVR32 without any significant effort in order to start long term testing. Further testing on AVR32 is required to reach any major conclusion.

Chapter 15

# Further Work

This chapter summarizes the further work proposed in the discussion chapter.

- Improve performance on the simulator by optimizing digitization and implement a smarter selection of cells.
- Improvement of the cell metavalue equation to make Mode Two function better
- Make the moving bits more realistic. For instance, starting at different points in time.
- Port the error prevention system to an AVR32 microcontroller. Perform tests over a prolonged period of time while collecting data.
- Implement (or find) a software ECC solution in order to compare the error prevention system against it.
- Make a feasibility study on the machine learning approach. Is it possible to collect the data? Would the suggested network work?

# Bibliography

[1] Akinaga, H., Shima, H., dec 2010. Resistive Random Access Memory (ReRAM) Based on Metal Oxides. Proceedings of the IEEE 98 (12), 2237–2251.
URL `http://ieeexplore.ieee.org/document/5607274/`

[2] Chen, B., Zhang, X., Wang, Z., Oct. 2008. Error correction for multi-level NAND flash memory using reed-solomon codes. In: IEEE Workshop on Signal Processing Systems, SiPS: Design and Implementation. IEEE, pp. 94–99.
URL `http://ieeexplore.ieee.org/document/4671744/`

[3] Chimenton, A., Pellati, P., Olivo, P., 2001. Analysis of erratic bits in flash memories. IEEE Transactions on device and materials reliability 1 (4).

[4] Iwasaki, T. O., Ning, S., Yamazawa, H., Sun, C., Tanakamaru, S., Takeuchi, K., may 2015. Machine Learning Prediction for 13X Endurance Enhancement in ReRAM SSD System. In: 2015 IEEE International Memory Workshop (IMW). IEEE, pp. 1–4.
URL `http://ieeexplore.ieee.org/document/7150294/`

[5] Lofstad, J., 2018. Detection of potential nvm (non-volatile memory) single bit fails by internal cell current measurements in embedded systems. Appendix B.

[6] Lue, H. T., Du, P. Y., Chen, C. P., Chen, W. C., Hsieh, C. C., Shih, Y. H., Lu, C. Y., 2012. Radically extending the cycling endurance of Flash memory (to $>$ 100M Cycles) by using built-in thermal annealing to self-heal the stress-induced damage. 2012 International Electron Devices Meeting.
URL `http://ieeexplore.ieee.org/document/6479008/`

[7] Micheloni, R., Crippa, L., Marelli, A., 2010. Inside NAND Flash Memories. Springer.

[8] Micheloni, R., Marelli, A., Ravasio, R., 2008. Error Correction Codes for Non-Volatile Memories. Springer.

[9] Mitchell, T. M. T. M., 1997. Machine Learning. MIT Press and The McGraw-Hill Companies, Inc.

[10] Nakamura, Y., Iwasaki, T., Takeuchi, K., apr 2016. Machine learning-based proactive data retention error screening in 1Xnm TLC NAND flash. In: 2016 IEEE International Reliability Physics Symposium (IRPS). IEEE, pp. PR–3–1–PR–3–4.
URL `http://ieeexplore.ieee.org/document/7574632/`

[11] Nielsen, M. A., 2015. Neural Networks and Deep Learning. Visited at 2019-10-02.
URL `http://neuralnetworksanddeeplearning.com/`

[12] Novotny, R., Kadlec, J., Kutcha, R., 2015. Nand flash memory organization and operations. Journal of Information Technology & Software Engineering 5 (1).

[13] Russell, S. J. S. J., Norvig, P., Davis, E., 2016. Artificial intelligence : a modern approach. Pearson Education Limited.

[14] Schuler, F., Degraeve, R., Hendrickx, P., Wellekens, D., 2002. Physical descriptopn of anomalous charge loss in floating gate based nvm's and identification of its dominant parameter. IEEE 02CH37320, 40th Annual International Reliabiity Physics Sumposium, Dallas, Texas.

[15] Sommerville, I., 2015. Software Engineering. Pearson.

[16] Walpole, R. E., Myers, R. H., Myers, S. L., Ye, K. E., 2016. Probability & Statistics for Engineers & Scientists Ninth Edition. Pearson.

[17] Wikipedia.org, 2018. Artifical neural network. `https://en.wikipedia.org/wiki/Artificial_neural_network#/media/File:Colored_neural_network.svg`, Visted at 2019-04-05.

[18] Wikipedia.org, 2018. Nand flash structure. `https://commons.wikimedia.org/wiki/File:Nand_flash_structure.svg`, Visted at 2018-09-05.

[19] Zambelli, C., Olivo, P., Koebernik, G., Ullmann, R., Bauer, M., Tempel, G., apr 2013. Erratic bits classification for efficient repair strategies in automotive embedded flash memories. In: IEEE International Reliability Physics Symposium Proceedings. IEEE, pp. 2E.3.1–2E.3.7.
URL `http://ieeexplore.ieee.org/document/6531964/`

# Appendix A - Source Code

The source code for simulator and prevention system has been attached for completeness. The source code is not fully documented where the doxygen comments are either lacking or inaccurate in large portion of the code. It is not in any shape ready for any production environment.

## 15.1 Tree view of files

### Error Prevention System

```
PreventionSystem
├── CMakeLists.txt
├── controlPanel
│   ├── controlPanel.c
│   └── controlPanel.h
├── datalog
│   ├── datalog.c
│   └── datalog.h
├── googletest
│   └── *
├── jsmn
│   └── *
├── main.c
├── search
│   ├── search.c
│   ├── search.h
│   ├── SearchTest.cpp
│   └── search_pi.h
├── simulatorNetworkInterface
│   ├── simulatorNetworkInterface.h
│   └── simulatorNetworkInterface.c
├── statistics
│   ├── statistics.h
│   └── statistics.c
└── storage
    ├── storage.h
    ├── storage.c
    └── StorageTest.cpp
    └── storage_pi.h
```

## Simulator

```
Simulator
├── testResources
│   ├── 0.csv
│   └── 1.csv
├── config.py
├── dataToCsv.py
├── movingBitsPlotter.py
├── simulation.py
├── tcpConnection.py
├── test_simulation.py
└── requirements.txt
```

# Appendix B - Prelude Project

**Term project Fall 2018**

Detection of potential NVM (Non-volatile memory) Single Bit Fails by internal Cell Current Measurements in embedded systems

**Johan Hopland Lofstad**

Department of Engineering Cybernetics
Norwegian University of Science and Technology
*In collaboration with*
Microchip Technology Norway AS

**NTNU**
**Norges teknisk-naturvitenskapelige universitet**

**Fakultet for informasjonsteknologi, og elektroteknikk**
**Institutt for teknisk kybernetikk**

# PROSJEKTOPPGAVE

| | |
|---|---|
| Kandidatens navn: | Johan Hopland Lofstad |
| Fag: | Teknisk Kybernetikk |

Oppgavens tittel (norsk):

Oppgavens tittel (engelsk):

*Detection of potential NVM (Non-volatile memory) Single Bit Fails by internal Cell Current Measurements in embedded systems*

Oppgavens tekst:

*A weakness of Flash (NVM) Memory is the non-zero probability of a memory cell of flipping its logic state due to charge loss over time. To mitigate the risk of erratic flash cells, factory-programmed systems that require extreme levels of data integrity, are often equipped with HW ECC at the expense of significant silicon cost. The charge leakage causing an erratic bit may evolve over a period of years, which makes it challenging to detect in traditional IC production test.*

*Scope of project:*

- *Based on a literature study, describe embedded Non-volatile memory effects causing Single Bit issues for NVM Technologies based on Floating Gate Architecture.*
- *Based on a literature study, describe how the cell measurements changes over time with accelerated aging. Attempt at least one accelerate aging technique on a specific Microchip microcontroller.*
- *Develop embedded code for a specific Microchip microcontroller, which has the needed HW-support to perform internal cell current measurements. The microcontroller system comprises CPU, memories, special test modes and bus structures, analog modules. Perform NVM cell Current measurements by internal method and compare with external Cell Current measurement*

*Upon successful completion of the project, the data may be used for a full-scale implementation of real-time error detection and correction on a microcontroller. This is however not in scope for this project*

| | |
|---|---|
| Oppgaven gitt: | 20. August 2018 |
| Besvarelsen leveres innen: | 18. desember 2018 |

Utført ved Institutt for teknisk kybernetikk

Hovedveileder: Rainer Herold

Trondheim, 20. August 2018

Geir Mathisen
Faglærer

# Abstract

Flash memory is a common memory used in a variety of devices, using an electric charge to store data. This electric charge can leak and cause an error in the form of a bit-flip. To prevent these errors, the electric charge on these flash cells must be monitored. This project presents a data collection system for flash cell charge using an internal test method on the mXT1067T chip.

The data collection system has been tested over a prolonged period on room and high temperatures. These tests measured the charge of all flash cells in the chip every hour. Results from these tests are presented with a short analysis. The internal test method used to collect the data has been evaluated. The system has also been benchmarked for performance, in terms of algorithm run-time and total run-time.

# Preface

This project is written for the department of cybernetics at NTNU in collaboration with Microchip Technology Inc. Microchip approached the author of this project with a proposal to investigate erratic and moving flash memory bits, and how they could be detected and prevented using internal test methods.

In order to make such an error prevention system, a data collection system for cell currents had to be developed, hence this project. It is meant as a preliminary to a Masters Thesis which explores the possibility of implementing such an error prevention system.

As this project is a collaboration, there are two supervisors. Geir Mathisen from NTNU agreed to supervise the layout of the report and make sure the work progressed at a reasonable pace. Rainer Herold from Microchip agreed to supervise on the technical and theoretical level, suggesting paths to take the project as it progressed.

# Table of Contents

# List of Tables

# List of Figures

# Abbreviations

| | | |
|---|---|---|
| FGMOS | = | Floating Gate Transistor |
| FN Tunneling | = | Fowler Nordheim Tunnelig |
| ECC | = | Error Correction Code |
| SAFT | = | Scale-Accelerated Failure-Time Models |
| IC | = | Integrated Circuit |
| PC | = | Program Counter or Personal Computer |
| mXT | = | maxTouch |
| HW | = | Hardware |
| ECC | = | Error Correction Code |

Chapter 1

# Introduction

## 1.1 Background

Flash memory is a prevalent type of non-volatile [1] memory, used in a wide variety of applications. These flash memories often hold the application code for microcontrollers, as it is crucial this data is not lost on power off.

A weakness of flash memory is the non-zero probability of a flash cell flipping its logical state. Flash memory uses an electrical charge to determine whether a bit is a zero or a one. A leakage of electrical charge can occur due to imperfections in the silicon known as traps. This charge leakage may evolve throughout years, making it challenging to detect.

To mitigate the risk of a flash cell flipping and causing data corruption, flash controllers have traditionally been equipped with HW ECC (Hardware Error Correction Code) which has a significant silicon cost. Error correction codes are mathematical codes calculated on some bits, which can detect and restore small errors.

## 1.2 Motivation

As HW ECC brings with it significant silicon cost, one can argue for a software solution. A software solution with the same effect as the HW solution would have to store error correction codes all cells, which again would be stored on the flash. CPU time must be dedicated to calculate and evaluate the ECC.

Instead, an error prevention scheme is proposed. By monitoring the charge level of all individual flash cells, it might be possible to detect charge loss before the logical state flips. Such a scheme would require far less storage and the calculations could transpire as the CPU idles.

[1] Volatile memory loses data on power loss

## 1.3 Limitations

This project has limited itself to a data collection system, which measures the cell charge by internal test modes. The measured data is transmitted to a PC which stores the data for analysis. Some effort has been made for an initial analysis of data, attempting to find erratic and moving bits. As it is desirable to limit the CPU time such a system uses, the effort has been put into optimizing the measurement algorithms.

No code has been provided with this text, as the psuedocode should be enough to reproduce the results. Appending the actual C code would only add confusion and no insight into the implementation.

A challenge with writing this text is the abstraction level. As this project is a deeply technical one, the interesting results and problems occur on a particularly low abstraction level which requires the reader to have certain background knowledge. At the same time, the text should be easily readable to understand what has been done, and what the results represent.

Because of this challenge, parts of the text is not required to read if the reader does not desire a deeper understanding. Throughout the text, these ”non-mandatory” sections have been attempted marked, and the reader is free to skip these on his or her leisure.

## 1.4 Disposition

This section gives a quick overview of what the different chapters contain.

**Chapter 2: Theory** introduces the theory needed to understand non-volatile memory and flash memory. This is required reading to understand the problem at hand.

**Chapter 3: Literature Study** explores two failure modes in non-volatile memory and two accelerated aging methods for silicon. It is recommended that the reader understands the basics of all these topics, but the deeper understanding of these topics, which is also presented, is not required.

**Chapter 4: Specification of project** presents the project specification and acceptance criteria. This presents what is to come later in the coming chapters.

**Chapter 5: Design of data collection system** introduces how the data collection system is designed on a top level. It explains the different test modes which are used and the basic layout of the modules.

**Chapter 6: Implementation of data collection system** introduces an implementation of the design in Chapter 5. Different measurement algorithms are discussed, the transfer of data between the modules and a simple analysis scheme. This chapter is not required to understand the rest of the text but is recommended as it gives insight into the reason for some the results.

**Chapter 7: Testing: Accelerated aging and FDMA v DMA** explains the setup and methodology for two tests, one at $22°C$ and one at $105°C$. The setup and methodology for testing the accuracy of FDMA are also explained.

**Chapter 8: Results** presents the data collected through from the tests alluded to in Chapter 7. It also shows metrics for system performance.

**Chapter 9: Discussion** discusses the results obtained, and if the acceptance criteria have been met.

The remaining chapters, **Conclusion** and **Further work** wraps up the text and suggest further work on the topic.

Chapter 2

# Theory

This chapter presents theory which might be required to understand the rest of the text. The theory presented is on non-volatile memory, with the floating gate transistor and how it used in a NAND flash configuration.

## 2.1 Non-Volatile Memory

Digital memory is often divided into two categories, volatile and non-volatile. Non-volatile memory, in contrast to volatile memory, retains data on power loss.

One type of non-volatile memory is called Flash memory[1]. Flash memory is widely used in several applications where high capacity, low cost non-volatile memory is needed, and uses the **floating gate MOSFET** to store data.

### 2.1.1 Floating Gate Transistor

Every bit in a flash memory is a Floating Gate MOSFET, abbreviated to **FGMOS**. An illustration can be seen in Figure 2.1 and consists of three gates, the *control*, *source* and *drain* gate. The structure is similar to a conventional MOSFET which have the same gates. In a conventional MOSFET transistor, as a voltage is applied to the control gate, current is allowed to flow from the source to the drain. This happens as the ”barrier” between the N-Type and P-Type silicon are broken down, allowing electrons to freely move across from source to drain. In a MOSFET transistor the current from source to drain is a function of the voltage on the gate, $I_{sd} = g(V_c)$. In flash memory, every such FGMOS is referred to as a **cell**.

In a MOSFET the amount of voltage applied on the control gate when it becomes conductive is called the **threshold voltage**. In other words, the threshold voltage is the minimum

[1]The name comes from its fast erase, it erases in a ”Flash”

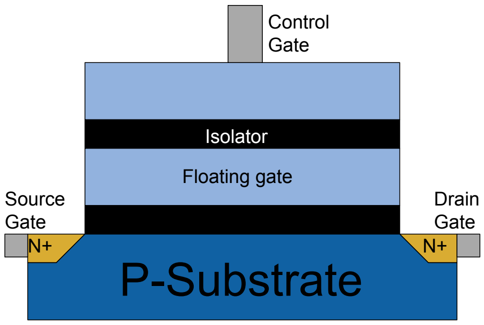


**Figure 2.1:** A Floating Gate MOSFET (FGMOS) Transistor

voltage needed to make the p-substrate become conductive.

The principal idea in a FGMOS is to store electric charge in the so called ”floating gate”, which sits in the middle of two isolators, i.e. completely electrically isolated. This allows for the storage of data in the form of charge in the floating gate which persists when power is lost. When there is no charge, the cell is labeled as **erased** and has a logical value of *one*. When there is a set amount of charge, the cell is labeled as **programmed** and has a logical value of *zero*.

In order to read a cell (i.e. measure the charge in the floating gate), a voltage is applied on the control gate (as in a MOSFET). However, in contrast to a MOSFET the FGMOS also contains charge in the floating gate ”resisting” the p-substrate to become conductive. In other words, the amount of voltage required to make the transistor conductive is a function of both applied voltage and stored charge in the floating gate $V_{threshold} = g(V_{gate}, Q_{floating\ gate})$ .

A related concept to the threshold voltage is the **cell current**. When applying a known fixed voltage between the source and drain gate, and a known fixed voltage on the control gate (called the **read voltage**), the current between the source and drain is the cell current. As the threshold voltage increases with an increasing $Q_{floating\ gate}$, the transistor becomes more conductive and thus more current flows. In flash memories, the cell current is used to determine if a cell is programmed or erased.

The cell current for a typically erased and programmed cell is shown in Figure 2.2. Both follow the well known *sigmoid* function, but they are shifted related to each other. The erased cell conducts ”all” the current at the read voltage of zero volts while the programmed cell outputs no current at all. This is the general idea of how a flash cells state is read.

In order for a flash cell to be useful, it must be able to be programmed and erased. The question is how to insert charge in the isolated floating gate. One way of adding and removing charge is **Fowler Nordheim Tunneling (FNT)**, which lets electrons ”tunnel” through isolators. To **erase** a cell and remove the charge, a high voltage is applied between the control gate and source. By FNT, the charge is tunneled through the isolator, removing the charge.

To **program** a cell, a high voltage between the source and drain is applied. A high sufficiently high voltage between the control gate and ground makes the transistor conductive. If these conditions are met, a high current flows from source to drain, making some electrons to jump into the floating gate. This technique is called **hot electron injection**.

As these high voltages and currents damage the transistor, a flash cell has a limited amount of program / erase cycles. After the transistor has been programmed / erased an amount of time, charge starts to leak from the floating gate until it is unable to hold charge for useable amount of time.

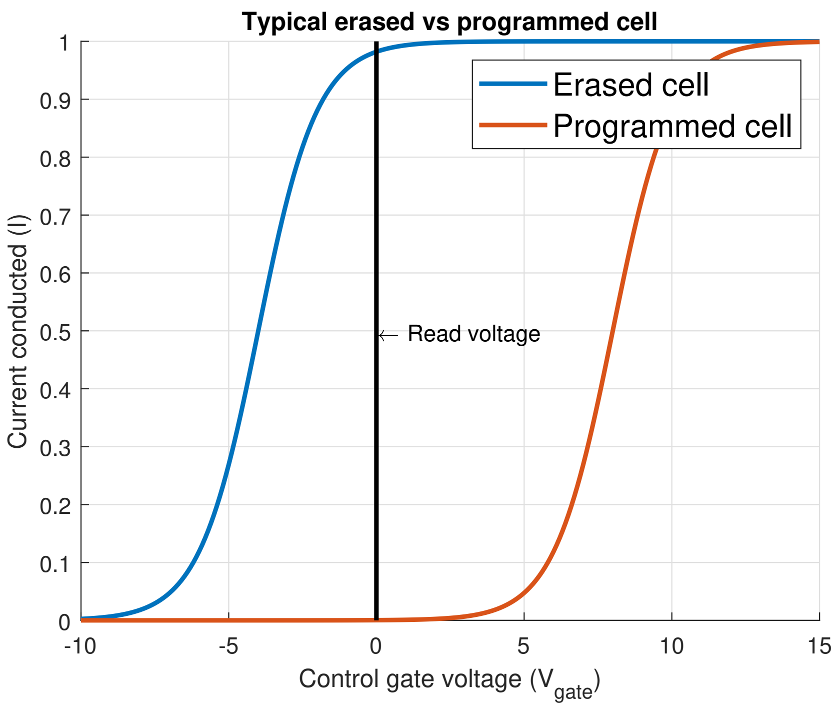


**Figure 2.2:** Illustration of current readout from a flash cell

### 2.1.2 NAND Flash Memory

NAND (Not-AND) Flash memory is a type of **non-volatile** memory where the data is stored in a hierarchy of floating gate MOSFETs. The term **NAND** refers to how the FG-MOS transistors are connected in series as can be seen in Figure 2.3.

The memory is divided into **blocks** with a number of **pages**. A page is defined as all the bits connected to the same word line. A *block* contains 32 or 64 pages and is the smallest *erasable* unit. A *page* is typically 2 or 4 Kbytes and is the smallest *programmable* unit [7].

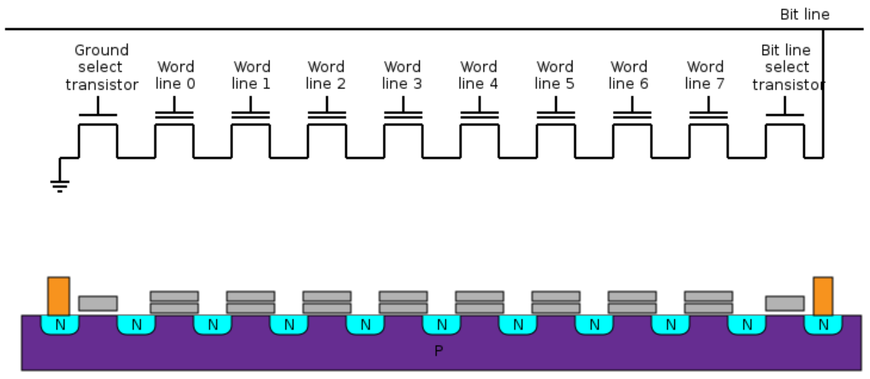


**Figure 2.3:** NAND Flash Configuration [11]

When **reading** a NAND flash, the read voltage is applied to the *wordline* of the bit (a word is a collection of bits, usually in the size of 16, 24 or 32 bits, the wordline selects which word is read). The *bit line select transistor* for the desired bit is set high. All the other wordlines in the bitline are set to a high voltage (Figure 2.3 is one bitline). This makes all the other transistors in the bitline conductive no matter the stored charge, as can be seen from the graph in Figure 2.2. The current value of the cell can now be read at the bitline.

Since it is not possible to erase individual bits, a full **erase-write** cycle must be performed when changing a bit. The data already existing on the block must be stored somewhere else, the block must be erased before the altered data can be written back to all the pages in the block. This degrades the oxide layer on the FGMOS, until a point where it can no longer hold the charge on the floating gate. This is why flash memory has a **limited number of writes/erases** before it is no longer usable.

To increase the lifetime of a flash memory, several techniques are applied. By storing **Er-**

**ror Correction Code (ECC)** for each block, the flash controller can detect faulty blocks and mark them as unusable. The flash controller also tries to spread out the writes between all the block, such that no block is written to excessively [7].

Chapter 3

# Literature Study

This chapter presents the literature study preceding the design and implementation. The first part presents failure modes in Non-Volatile Memory, classifying two common modes of failures. The second part discussed accelerated aging of silicon, and how effective they are on the given application.

If the reader does not seek the deeper understanding of these topics, parts of this chapter can be skipped. It is recommended that the basics are understood, such as definitions, before proceeding to Chapter 4.

## 3.1 Failure modes in Non-Volatile Memory

A flash cell can fail due to several failure modes. Two of the common failure modes are **erratic bits** and **moving bits**. When a flash cell is a ”moving bit,” it loses charge over time causing an error when the charge loss has accumulated to a high enough level. An ”erratic bit” does not show this loss of charge over time, but the measured charge level can start to jump between levels.

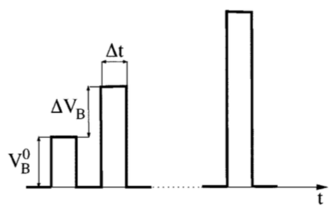
### 3.1.1 Erratic bits

A bit is defined as **erratic** when it exhibits sudden and unexpected jumps of the threshold voltage and is thought to be caused by holes in the FGMOS isolator. These holes are caused by the high voltage applied during erase operations [1]. Erratic bits are notoriously hard to detect due to their sporadic nature.

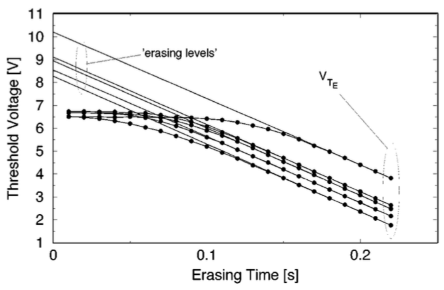
This section explores a study analyzing these erratic bits, attempting to find defining properties, and how these properties deviate from healthy cells. *This gives insight into the nature of erratic bits but is not necessary to understand or follow the rest of the text.*

In a *Analysis of erratic bits in flash memories* by Chimenton [1], an attempt was made to analyze the behavior of erratic bits. The experiment included an increasing erase pulse on some flash cells as seen in Figure 3.1a. After every pulse, the threshold voltage was measured. As the amplitude increase for every pulse, more charge is stored in the floating gate, which should cause a lower threshold voltage for every pulse.

The shape of the applied pulse is shown in Figure 3.1a and the measured threshold voltage in Figure 3.1b. After every increasing erase pulse, it is expected that the threshold voltage drops with $\Delta V_B$, where $V_B$ is the amplitude of the erase pulse. Figure 3.1b shows the time evolution for a "healthy" cell, where the threshold voltage asymptotically reaches the linear "erasing level."



**(a)** Shape of the erasing pulse applied [1]



**(b)** Time evolution of the threshold voltage of different cells during an erase operation. Each point represents the threshold voltage measured after each pulse, while the last one corresponds to $V_{TE}$. Here, $\Delta V_B = 0.3V$, $\Delta t = 10ms$, $V_B^0 = 3V$. The erasing levels on which the curves asymptotically move are also shown. [1]

**Figure 3.1:** Erase pulse and time evolution of the threshold voltage. Figure taken from [1]

When applying this erase pulse on known erratic bis, the erasing curve asymptotically reach the erasing curve as shown in Figure 3.2. Chimenton summarizes the following properties for erratic bits

1. Erratic behavior is characterized, during a single erase operation, by one or more level transitions
2. The transition can occur towards levels characterized by higher or lower $V_{T_E}$ (Threshold voltage after erase)
3. There is not a specific threshold or time during an entire erase operation at which these transitions are triggered.
4. A level change is not sudden but requires transitory of a few pulses

5. If the cell is found on an erase level at the end of the erase procedure, the following program operations do not modify this state and the next erase procedure is characterized, at least initially, by the same level

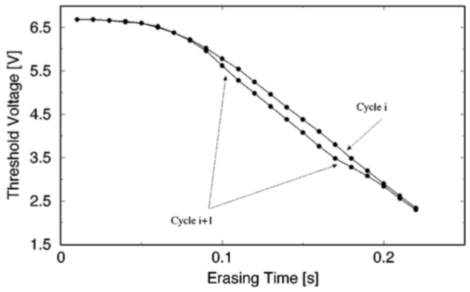


**Figure 3.2:** Erase curve on erratic bits. Figure taken from [1]

### 3.1.2 Moving bits

A bit is defined as **moving** when its threshold voltage moves slowly over time, on a timescale of months to years due to anomalous charge loss. One way moving bits differ from erratic bits is that moving bits are more predictable. By measuring the threshold voltage over time, a moving bit can be identified by looking at the slope. According to Schuler “Anomalous charge loss in cells with tunnel oxide thicknesses of more than about 6.4 nm results from chains of at least two more or less aligned defects.[8]”

The rest of this section explores the reason for this leakage. *This gives insight into the nature of moving bits but is not necessary to understand or follow the rest of the text.*

When semiconductors are used to make electrical circuits, parts of the semiconductor is ”doped” to either be conductive or isolating. Sometimes atoms are not doped correctly, causing them to stay semiconducting when designed to be isolating.

The defects Schuler writes about are these ”wrong” atoms and are known as **traps**, which forms a conductive path through the silicon. Figure 3.3 illustrates how these traps construct this conductive path. If these traps align, a conductive path is constructed. This causes the electric charge stored in the cell to slowly leak over time.

According to F. Schuler, the leakage current can be modelled exponentially by a *1-step tunneling* model.

$$I_{leak} = \sigma A_{FN} F^2 exp\{-\frac{4}{3h}\sqrt{2qm^*(\phi_1 - \Delta\phi)^3}\frac{1-(1-\frac{Fx_t}{(\phi_1-\Delta\phi)})^{\frac{3}{2}}}{F}\} \tag{3.1}$$

$$\Delta\phi = \sqrt{\frac{qF}{4\pi\epsilon_{ox}\epsilon_0}} \tag{3.2}$$

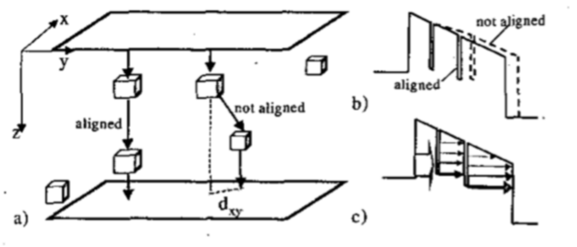


**Figure 3.3:** a) Schematics of a 3-dimensoanl distribution of traps building a leakpage path. b) Influence of misalignment on the band diagram of the effective tunneling path. c) Trap-assisted charge transport taking different energy levels into account. Figure taken from [8]

Where

- $\Delta\phi$ is the barrier lowering
- $F$ is the electric field
- $A_{FN}$ is the Fowler-Nordheim parameter
- $\phi_i$ is the trap depth
- $\sigma$ is the effective cross-section
- $x_t$ is the tunneling distance

The full model explanation can be found in [8]. The model in Eq 3.2 is a simplified version of the *Multi-Phonon-Assisted tunneling model* presented in the same study, where the *trap to trap* distance was found as the most dominant parameter. The multi-phonon-assisted tunneling model found that there is a slow down of charge loss of some bits, which the simplified model could not predict as shown in Figure 3.4.

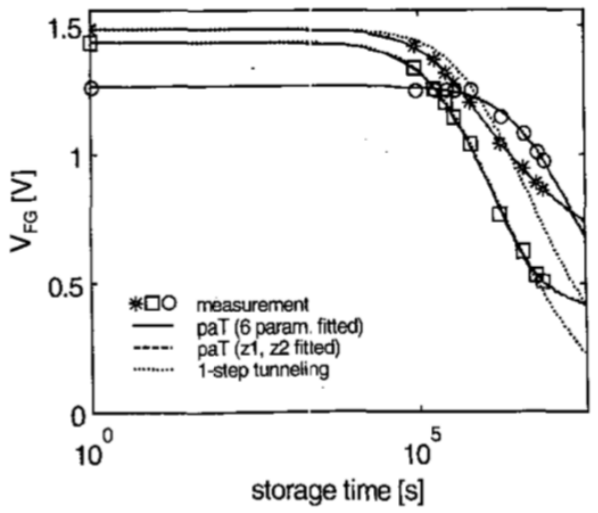


**Figure 3.4:** Measured transient charge loss of some typical Moving Bits as well as the simulated charge loss characteristic using the multi-phonon-assisted tunneling. Figure taken from [8]

## 3.2 Accelerated aging models

As some effects on embedded devices can take months to year to surface, a technique ”accelerated aging” is used. The basic idea is to put the device in a stressed environment or state, which emulates normal usage over a longer period of time. There are many different models and types of stress which can be applied. Two types of stress are **temperature** and **voltage**.

One type of models is **Scale-Accelerated Failure-Time Models (SAFT)**. It characterizes the effect some variables have on the lifetime of the device. A model is SAFT if $T(x) = \frac{T(X_u)}{AF(x)}$ where $T$ is the lifetime, $x$ is a some variable under stress, $x_u$ is some variable under normal usage and $AF(x)$ is the **Acceleration Factor**. By using this, the accelerated time can be found with Eq 3.3

$$t_{accelerated}(x) = AF(x)t_{actual} \tag{3.3}$$

**Example**
A device is put into a stressed environment for $t_{actual} = 24h$. By using some model, the acceleration factor of $AF = 13$ is found. This corresponds to $t_{accelerated} = 13 * 24 = 312h$ of normal usage.

### 3.2.1 Temperature acceleration and Arrhenius equation

Temperature aging is very commonly used to accelerate failure in integrated circuits. One of the most used models is **Arrhenius equation**, Eq 3.4 [3] [6].

$$AF(temp, temp_U, E_a) = e^{11605 E_a (\frac{1}{T_u} - \frac{1}{T})} \tag{3.4}$$

Where

- $E_a$ is the activation energy (provided my material characteristics)
- $T$ is the temperature during stress in kelvin
- $T_U$ is the temperature during normal usage in kelvin

Arrhenius is a empirical model which describes the rate of chemical processes under different temperatures. The $11605$ number comes from the Boltzmann constant ($k = 8.6171 * 10^{-5} eVK^{-1}$).

**Example**
A device rated for normal use under $T_u = 70°C = 70 + 273.5 = 343.5K$ is stressed at $T = 125°C = 125 + 273.5 = 398.5K$. The activation energy for the flash memory is found at $E_a = 0.8eV$. By Arrhenius equation, the acceleration factor is

$$AF = e^{11605*0.8(\frac{1}{343.5} - \frac{1}{398.5})} \approx 42$$

**Limitation of Arrhenius**

Arrhenius equation describes the rate of chemical reactions during special circumstances, and if these are not correct, the model is not suitable. For instance, if there are several chemical reactions with different activation energy

High temperatures have also proven to heal certain kind of silicon defects, such as erratic bit behavior in flash memory. A high temperature is also used in some flash memories to heal and extend their program/erase cycles by a significant amount[5].

### 3.2.2 Voltage acceleration and the inverse power relationship

*This acceleration method has not been used in this project, and can thus be skipped unless the reader so wishes.*

Acceleration can also be achieved by applying a higher voltage. According to Luis Escobar [3] ”when the thickness of a dielectric material or insulation is constant, voltage is proportional to voltage stress.” This gives the *inverse power relationship* model in Eq 3.5

$$AF(volt) = (\frac{volt}{volt_U})^{-\beta_1} \tag{3.5}$$

Where

- $volt$ is the voltage during stress
- $volt_U$ is the voltage during normal usage
- $\beta_1$ is a negative constant and can be estimated (methods for estimating $\beta_1$ are described in *Meeker and Escobar, 1998, Chapter 19)*

It is, however, important to keep in mind that the assumption "*when the thickness of a dielectric material or insulation is constant*." This is seldom the case for an entire IC, as MOSFET transistors rated for different voltages has different thickness. Microcontrollers seldom have only one power domain, which is often of different voltages. This is not considered in the above model.

Chapter 4

# Specification of project

This chapter outlines the function specifications of the project as well as the acceptance criteria for completion.

## 4.1 Functional Specifications

A data collection system to measure cell current for flash cells should be developed. The system should be able to read the cell current of any cells on the flash and store the result in a CSV file. An arbitrary amount of words (32 bit) can be read at any given time, with an arbitrary start and end.

The system should consist of two modules, MCU and PC. The MCU unit is the microcontroller which contains the flash and should run the measurement algorithms. These measurement algorithms should be implemented with internal test modes available on the given microcontroller. An external test mode should evaluate the internal test modes accuracy.

The PC module should be able to program to RAM and start MCU execution from RAM. The PC should send information to the MCU indicating which part of the flash which should be measured, fetch the results from the MCU ram and store it in a CSV file. It should be able to start a new measurement at a given time interval.

The system should be tested with two parallel long-term tests, one with accelerated aging at $105°C$ and one without accelerated aging at $22°C$. These tests should run the data collection system every hour and save the data it produces. After these tests have concluded, an initial analysis of the results should be done.

## 4.2 Acceptance criteria

The project is deemed finished when the following acceptance criteria have been met

1. A data collection system has been developed. Any flash cell can be measured, and the measurement can be stored for later analysis
2. Two long-term tests have been conducted, one at $105°C$ and one at $22°C$. Measurements should be done automatically at a given time interval
3. An evaluation of the data collections should be done by comparing the collected cell currents with their true value.
4. Data from the two long-term tests have been given an initial analysis. An attempt to find erratic and moving bits should have been made

Chapter 5

# Design of data collection system

This chapter introduces the design of the data collection system for the collection of cell currents of flash cells. It introduces the two main modules of the system and how they interact. Two test modes for conducting cell measurements are presented, closing with a proposed initial analysis methodology.

## 5.1 Introduction

In order to find erratic behavior in flash memory, a method for reading out the threshold voltage of the flash cells must be developed. In this project, the proposed method is for a *Microchip mXT1067T* chip. The mXT1067T is an AVR32 core microcontroller, specially made to be a controller for touch screens. As of the time of writing, the mXT1067T has not yet reached the market, as such the proposed method was developed on an engineering sample. This design would not necessarily work on any other chip than the MXT1067T, as it uses an internal test mode made explicitly for the max touch family of chips.

There are two main modules in the system, the MCU (microcontroller unit) and PC (Personal Computer). The MCU contains algorithms for measuring the threshold voltage and transferring this data to the PC. The PC starts and control when a measurement is being conducted, and which parts of the flash are measured. It also stores the result for later analysis.

## 5.2 Program flow

A sequence diagram illustrating the communication between the PC and MCU can be seen in Figure 5.1. The system starts with the PC programming the MCU and loading initial values. These initial values indicate how and which sections of the flash should be measured. For instance which measurement technique to use and which memory addresses to measure on.

When a measurement is complete, the MCU responds with the measurement data, which is acknowledged by the PC. As there exists a maximum limit of the size of each message, there has to be multiple measurements and messages.

The measurement runs in a loop with a set time in between. When the measurement has been completed, the resulting data is stored in a file. This file contains the address, bit and the measurement for every flash cell which was measured.

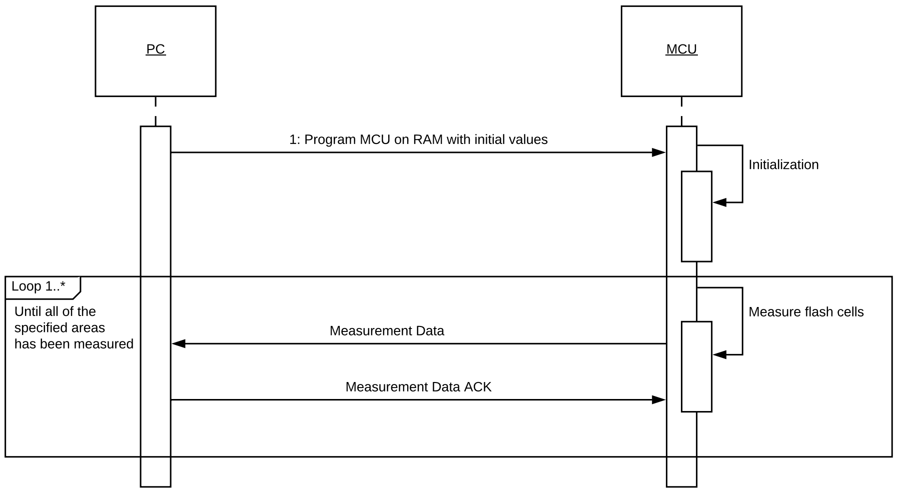

PC
MCU
1: Program MCU on RAM with initial values
Initialization
Loop 1..*
Until all of the specified areas has been measured
Measure flash cells
Measurement Data
Measurement Data ACK

**Figure 5.1:** UML Sequence Diagram of the MCU / PC Communication

## 5.3 Conducting a cell measurement

In Figure 5.1 one of the internal communications is ”measure flash cells”. The two main approaches to measure flash cells on the mXT1067 are **FDMA** and **DMA**. In the following section, these two methods will be explained and explored.

### 5.3.1 Fast Direct Memory Access (FDMA)

FDMA (Fast Direct Memory Access) is an internal test mode which can be used to measure the cell current of a flash cell. It works by overriding an internal comparator[1] current.

When a flash bit is read, a read voltage is applied. On the mXT1067T, this read voltage is around 0V (although this can be changed with internal test modes). The output current from the drain (called **cell current**) is compared with an internal current generator.

[1]A comparator is a device which compares two electrical currents

If the cell current is lower than the internal current generator, the bit is **programmed** with a logical value of zero. If the current source is greater the internal current generator, the bit is **erased** with a logical value of one

The cell current is closely related to the threshold voltage in FGMOS transistors. Threshold voltage indicates the minimum voltage that needs to be applied before the transistor is conductive while the cell current is the amount of ”conductivity” which has been achieved. In other words, they are both a measure of the amount of charge in the floating gate.

It is the internal current generator which defines how much charge must be on the FGMOS before it is considered programmed. The **FDMA** test mode uses this to measure the cell current by overriding the internal current generator as illustrated in Figure 5.2, labeled ”FDMA Current Generator”. The programmer is now able to select the magnitude of the current the cell current is compared against.

When the current generator is overridden with the FDMA Current Generator, the current is called **FDMA Current**. While monitoring the digital value of a cell (zero or one, programmed or erased), the FDMA Current is increased from a low value to a higher value with a given increment. As soon as the digital value of the cell flips (goes from zero to one or vice verse) the FDMA current has just surpassed the cell current. In other words, the FDMA current at that moment is a close approximation of the cell current.

Figure 5.3 illustrates this concept. The circles represent cells, and the number illustrates their cell current. The dotted line represents the active FDMA Current. A red circle returns one while a green returns zero when read. As the FDMA current increases, more and more cells turn red and goes from *programmed* to *not programmed* as seen from the flash controller.

### 5.3.2 Direct Memory Access (DMA)

FDMA uses internal mechanisms to calculate the cell current as the cell current is not measured directly. Thus one can not be $100\%$ sure of the accuracy of the readings. It would be beneficial to compare the cell current read with FDMA with the actual cell current to validate the accuracy of the FDMA results.

Another test mode, **DMA** (direct memory access), connects the output of a flash cell to a physical pin. In figure 5.2, this is illustrated as the DMA switches leading to the DMA output. This solution is significantly slower than FDMA, as it requires a very long settling time and the current must be measured externally by another instrument. Reading the entire flash in this manner would take an enormous amount of time.

## 5.4 Analyzing the data, finding moving bits

This chapter has so far discussed how to use the FDMA and DMA test modes to collect data. The end goal for such a data collection scheme is to be able to locate erratic and

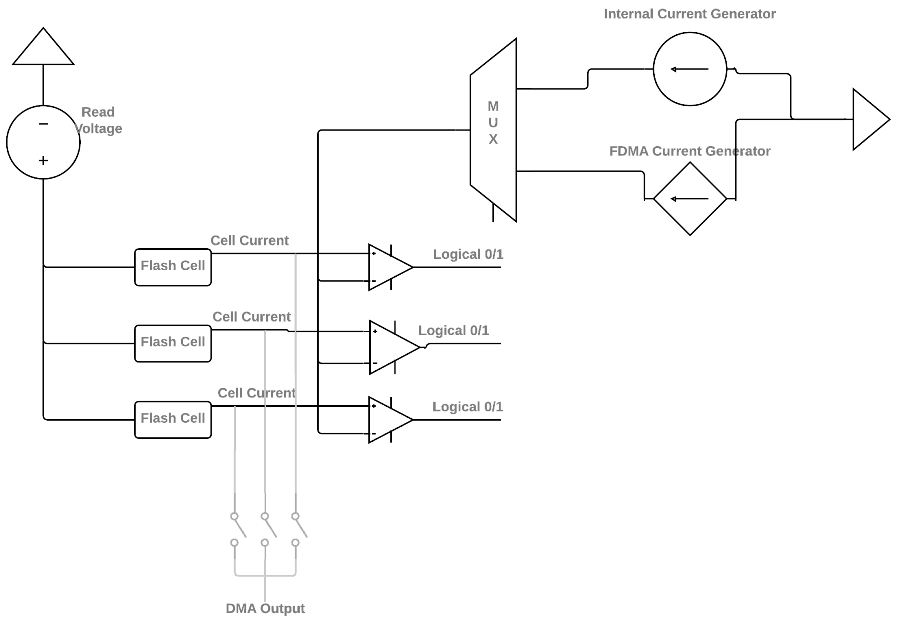


**Figure 5.2:** Illustrative circuit diagram of the FDMA Test mode

moving bits, and for that to be possible time has to be introduced. By measuring the entire flash with a constant time interval and transferring the results to the PC after every interval, the data also includes time. A separate file is created for each time interval the measurement is done.

Locating a moving bit from a set of data is a challenge in itself. In section 3.1.2, a moving bit is defined as a bit which threshold voltage moves slowly over time due to anomalous charge loss. However as the cell current and FDMA current can vary with parameters such as temperature, only checking $\Delta I$ (change of cell current) would not work. In order to locate a moving bit, the cell current of neighboring cells must also be taken into account.

As this is an initial analysis attempt, the scope is limited not to find moving and erratic bits, but to find candidates for these bits. If parameters such as temperature remain constant at all time, the problem can be simplified to a looking for a large enough $\Delta I$.

The data from the first time instance $t = 0$ is defined as $I_0$. The maximum amount of current a cell is allowed to deviate from $I_0$ is defined as $I_{threshold}$. For all data when $t > 0$, the current $I_t$ is compared against $I_0$. If $(I_t - I_0) = \Delta I \geq I_{threshold}$, the bit is marked as a moving bit candidate. Please note the word *candidate*, as there is no guarantee that a bit detected this way is a moving bit.

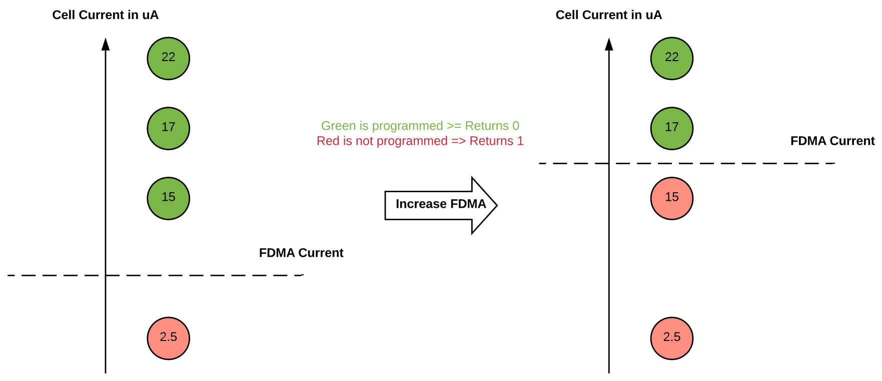


**Figure 5.3:** Visual representation of FDMA. Circles are cells with their cell current. Red cells return one when read, green cells return zero when read.

When a candidate is found, it must be manually checked by a human with the current solution. This could be automated but is beyond the scope of this project due to the limitations alluded to in the introduction. By plotting the trend of the bit over time, a human could identify moving or erratic behavior by how the trend behaves. For instance, a linearly decreasing trend would suggest that the bit is moving while a trend which jumps *erratically* would suggest an erratic bit. An illustration of a normal, moving and erratic bit are shown in Figure 5.4.

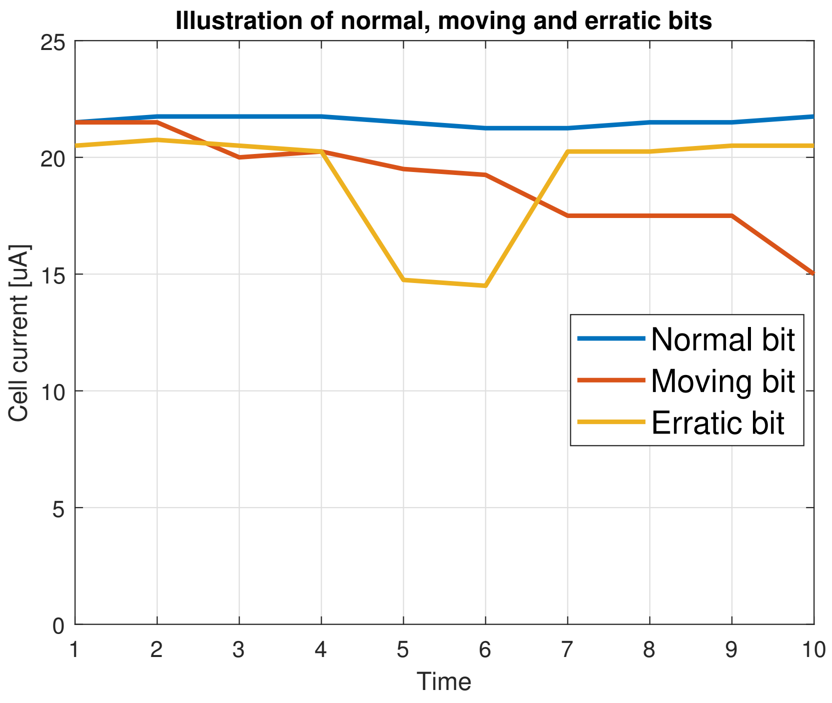


**Figure 5.4:** Illustration of moving, erratic and normal bits

Chapter 6

# Implementation of data collection system

This chapter outlines the implementation of the data collection scheme with the two main modules, the MCU and the PC. The MCU implementation includes several measuring algorithms using both FDMA and DMA and how the measurements are stored and sent to the PC.

The PC implementation includes how a measurement is started by programming the MCU and communicating with it. It covers how the measurement is received from the MCU and stored in CSV files. It describes how measurements are started on given time intervals to measure over time.

Implementation details concerning both the PC and MCU are discussed. Some of the problems solved are how to transfer the data between the MCU and PC with a limited amount of RAM, how the synchronization is done and how they interact throughout measurements.

## 6.1 Introduction

The MCU is as previously disclosed a **mXT1067T** microcontroller from Microchip Technologies Inc. The firmware for the MCU is written in C with the ANSI C dialect. It is compiled with a modified toolchain which is discussed in detail in the next section.

The PC module is written in Python 2.7, using the ”edbg-cli” library to communicate with the MCU through JTAG. To load and modify hex files it uses the intelhex package.

## 6.2 Building and executing the MCU Firmware from RAM

The standard way when programming microcontrollers are to program the compiled binaries to a part of the flash memory often referred to as **PROGMEM**, at which the program counter [1] defaults to. However, as writing our program to flash reprograms that sector this is not a viable option. Another approach is to write the program to RAM (Random Access Memory), and then change the program counter to this location in RAM. By doing so, the flash is not altered each time a change is made to the program. This can be achieved using JTAG [2]. The JTAG on mXT1067T allows us to write to any part of the memory and registers. Figure 6.1 illustrates this with a flowchart.

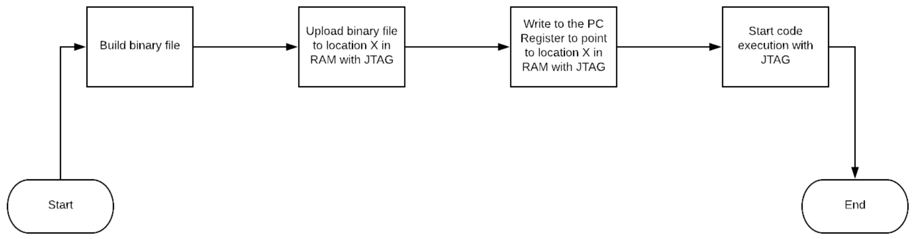

Build binary file
Upload binary file to location X in RAM with JTAG
Write to the PC Register to point to location X in RAM with JTAG
Start code execution with JTAG
Start
End

**Figure 6.1:** Flowchart for programming to RAM and start execution. In this figure, PC refers to ”Program Counter”

### 6.2.1 Building the program binaries

The mXT1067T is a microcontroller unit with an AVR32 core [3]. AVR32 uses binaries in an Intel hex format which has a .hex file extension. These files are traditionally referred to as **hex files**. A toolchain for creating such hex files has been provided by Microchip (at the time of release Atmel) and GNU. The toolchain uses the naming convention *avr32-**, for instance *avr32-gcc*. [4]. The full build script is written in a ”Makefile.”

[1]The Program Counter is a pointer which points to the next instruction the CPU should execute
[2]JTAG is a standard used to debug microcontrollers
[3]AVR32 is technically discontinued by Microchip, but new MaxTouch silicon still uses this core
[4]These binaries are bundled with Atmel Studio

Usually, the building of a hex file is straightforward.

1. Compile and assemble all the .c files into .o files
2. Link all the .o files into an elf file
3. Convert the elf file into a binary file
4. Convert the binary file to Intel hex format.

The need to run from RAM is crucial, as it is the flash which is measured and programming it would alter its state. It is therefore not possible to use this build sequence without some minor modifications. There are two modified components to the toolchain compared to the one default bundled with Atmel Studio 7.0. The *linker script* and a *start interrupt assembly*.

A **linker script** is used in the linking stage and defines the memory layout of the program. For instance, where the heap is located, where the stack is located and so on. For instance, the linker script could put the stack at the memory 0x00 and the heap at 0xFF.

The default bundled linker script with Atmel Studio puts the program (.reset section) to the start of the flash. Therefore it is needed to change the start of the .reset section (where to start execution) from $0x80000000$ (start of flash) to $0x00000000$ (start of ram).

The default linker script puts the dynamic memory (heap) to the start of RAM, but this is where the program is located after the previous change. To avoid heap allocations to overwrite our program, a different part of the RAM to be used as the heap.

The second component is the **start interrupt assembly**, which defines the stack pointer. The details of this change are not important, but in short, it changes the stack pointer to match the changes described above.

### 6.2.2 Loading and running from RAM

The resulting hex file after building is only functioning when ran from RAM with the program counter pointing to the start of RAM. This is done through the JTAG interface, using an Atmel Ice or equivalent Atmel debugger. The software is written in Python 2.7, using a library called ”edbg-cli.” This allows us to create a JTAG connection through python.

The general idea is

1. Reset MCU and halt its execution
2. Fill start of MCU RAM with the hex file, built with the modified build system
3. Change the program counter to the start of RAM
4. Start MCU execution

This programming to RAM and starting execution is done by the PC module every time a new measurement is ordered.

## 6.3 Reading Cell Current with FDMA

Several methods for measuring cell current with FDMA have been developed. First, a simple algorithm for a single cell is shown. This is used to create an algorithm which measures cell current for complete 32-bit words. These two first algorithms suffer from some performance problems, which are addressed as the last topic in this section.

### 6.3.1 Algorithm for measuring cell current

In this subsection, an algorithm for measuring the cell current for an individual cell is discussed. It follows the idea outlined in the design chapter of increasing the FDMA current until the bit flips, which would be an approximation of the cell current of that bit.

FDMA is enabled at $0.5\mu A$ while the read voltage is kept constant. The FDMA current is increased with a small increment (e.g., $0.25\mu A$), and after every increment, the bit value is read. If the bit has flipped, the FDMA current is logged. This algorithm is presented as a pseudocode as Algorithm 1 and a UML Activity Diagram in Figure 6.2.

**Algorithm 1:** Reading cell current with FDMA

```
Result: Cell current for flash cell in $\mu A$
1 enableFDMATestMode() ;
2 setFDMACurrent(0.5) ;
3 for $i \leftarrow 0.5$ to $43.0$ by $0.25$ do
4     setFDMACurrent(i) ;
5     bitValue $\leftarrow$ readBit();
6     if bitValue is 0 then
7         return i
8 end
9 return 0.5
```

Notice the *return 0.5* at the end of the algorithm. If a cells current is lower than $0.5\mu A$, a flip would never be observed. In other words, it must be less than $0.5\mu A$. One drawback with the FDMA approach is that only programmed cells can be seen as a negative voltage cannot be applied to the control gate. There exist protection mechanisms in the flash power supply which disallows this. Therefore the value of $0.5\mu A$ is returned when the bit never flips (the actual value is $\approx 0\mu A$ as it is programmed).

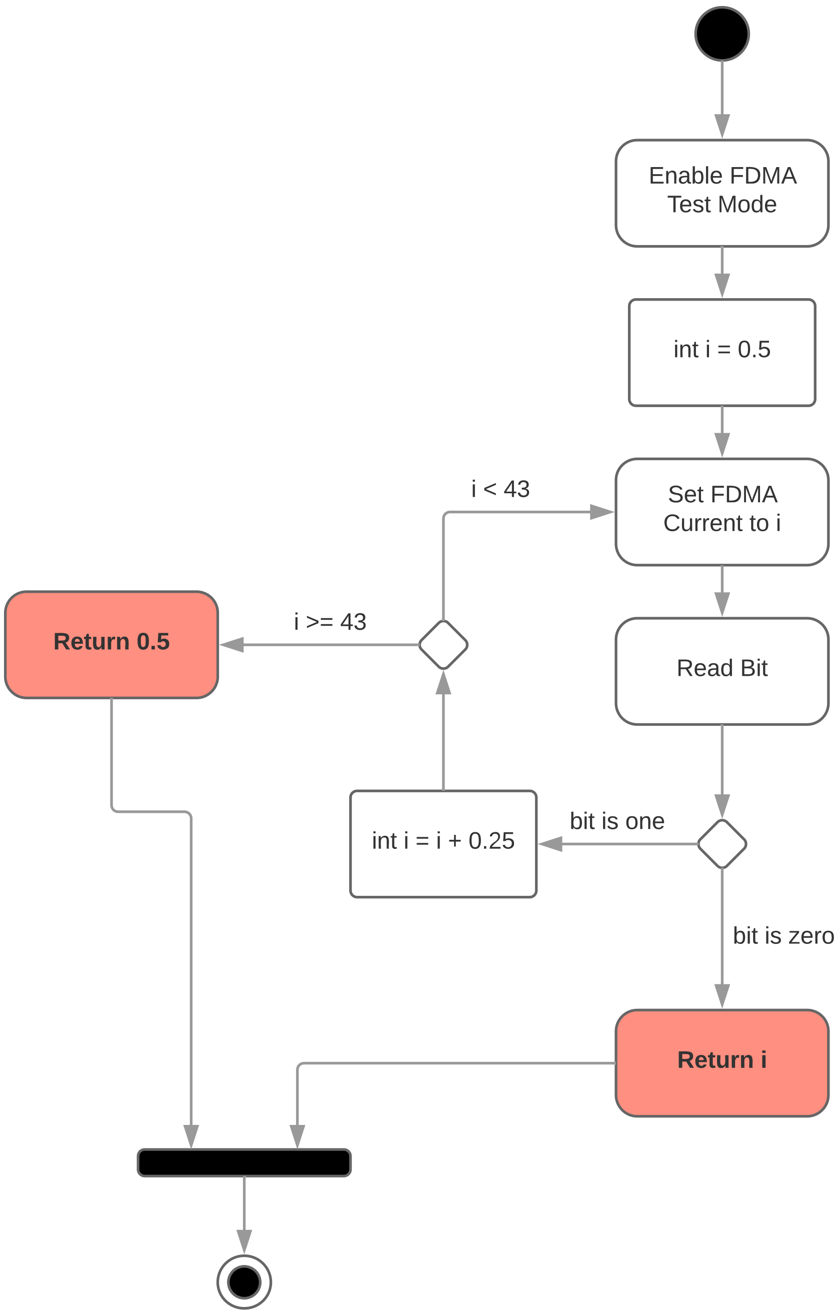


**Figure 6.2:** Activity diagram for reading single cell

### 6.3.2 Algorithm for reading multiple cells

In this subsection, an algorithm for measuring multiple cells at a time is discussed. Algorithm 1 measures only one cell, which is not feasible when measuring a large amount of cells, as every read to the flash returns a 32-bit word. A smarter solution would be to check for flips for all 32 bits returned in a read. Algorithm 2 does this by reading a 32-bit word after each incremental increase in FDMA Current.

**This is algorithm is identical to Algorithm 1, only for whole words. It is not necessary to read and understand the rest of this section to follow the rest of the text.**

It inputs the address of a start- and endword, defining a word range. For instance if the address of the start word is 0x5 and the address of the end word is 0xF, it measures the cell current for all bits located at 0x5, 0x6 ... 0xE, 0xF.

The algorithm is shown as psuedocode in Algorithm 2 and a UML Activity Diagram in Figure 6.3.

**Algorithm 2:** Reading cell current of a 32-bit word with FDMA

```
input: $startWord$
       $endWord$
Result: Cell current for flash cell in $\mu A$ for a word range
enableFDMATestMode() ;
$currentWord \leftarrow startWord$ ;
while $currentWord \neq endWord$ do
    for $i \leftarrow 0.5$ to $43.0$ by $0.25$ do
        setFDMACurrent($i$) ;
        for $bit \leftarrow 0$ to $31$ by $1$ do
            if bit is done then skip bit;
            if readBit(currentWord, bit) is 0 then
                saveValueForBit(bit, i) ;
                markBitAsDone(bit) ;
        end
        if all bits are done then currentWord++ and goto outer loop;
    end
end
for all bits without value do
    /* All bits without value has not flipped */
    saveValueForBit(bit, 0.5) ;
end
return bitValues
```

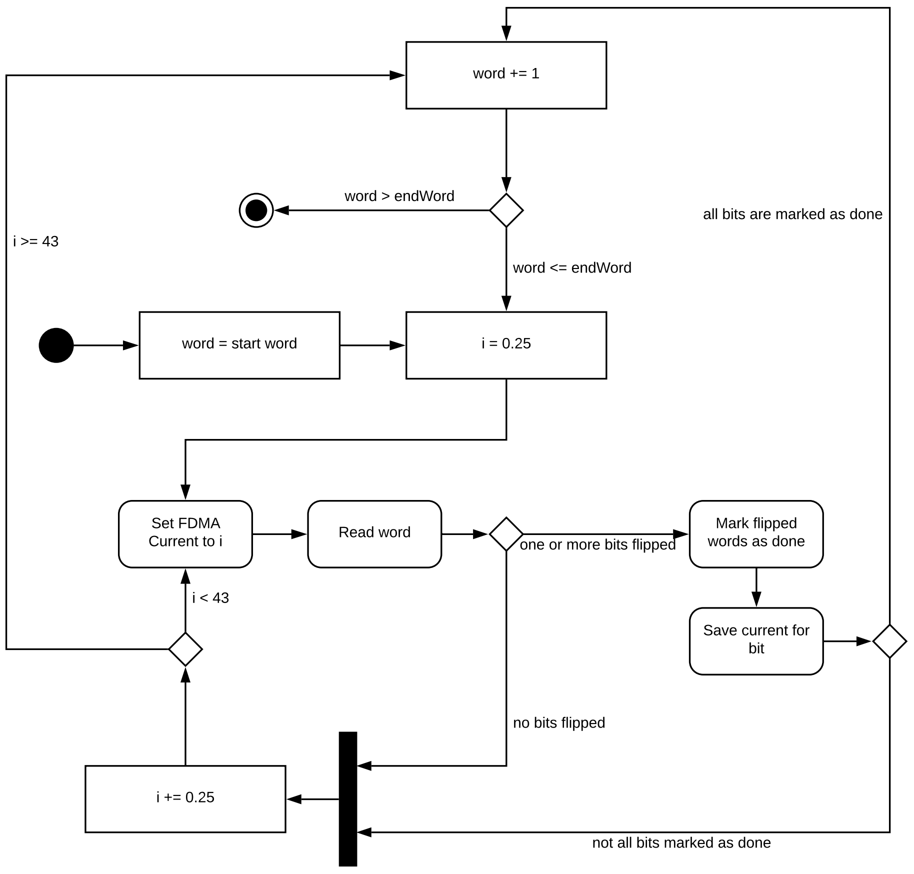


**Figure 6.3:** Activity diagram for reading multiple cells

### 6.3.3 Improving performance with Binary Search

Algorithm 1 searches over the entire current range for a bit flip ($0.5\mu A \rightarrow 43\mu A$). The probability distribution for which current a bit have is not uniform. If a bit is healthy, it should fall into the $20\mu A \pm 2\mu A$ range if erased or $0.5\mu A$ range if programmed. In other words, Algorithm 1 search area can be reduced quite extensively. An example of the distribution of a flash is shown in Figure 6.6, in which about $50\%$ of the bits are zero and the other $50\%$ one.

The search domain can be considered sorted, in which a binary search gives the best run-time. By applying binary search, the run-time is improved from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$ [2]. A set of buckets is defined, *a bucket list*, in which the cell currents can fall into with a step size between them. For instances, Figure 6.5 shows a set of buckets with a step size of 4 (first bucket is 4.0, next is 8.0, next is 12.0 and so on).

At the start the FDMA current is set to the bucket approximately in the middle of the bucket list. In Figure 6.5 this is $16.0\mu A$. If the bit is one when read, the bit has not yet flipped and is located to the right of $16\mu A$. If the bit is zero when read, the bit has flipped and is located to the left of $16\mu A$. The same procedure runs again, but this time at the right or left side of $16\mu A$, depending on the value of the bit. This continues until there is only one bucket left, which is the approximate current value of the bit.

The accuracy of the algorithm is dependant on the step size between the buckets. As you decrease the step size, both the accuracy and run-time increases. The algorithm is shown as a psuedocode in Algorithm 3 and as a UML Activity Diagram in Figure 6.4.

**Algorithm 3:** Binary search cell current

```
Result: Cell current for flash cell in $\mu A$
Data: n: Number of bucket
enableFDMATestMode() ;
L ← 0 ;
R ← n - 1;
while $L \leq R$ do
    m ← floor((L + R)/2) ;
    setFDMACurrent(FDMABucket[m]) ;
    bitValue ← readBit() ;
    if bitValue is 1 then
        L ← m + 1
    else
        R ← m - 1
    end
end
return FDMABucket[m]
```

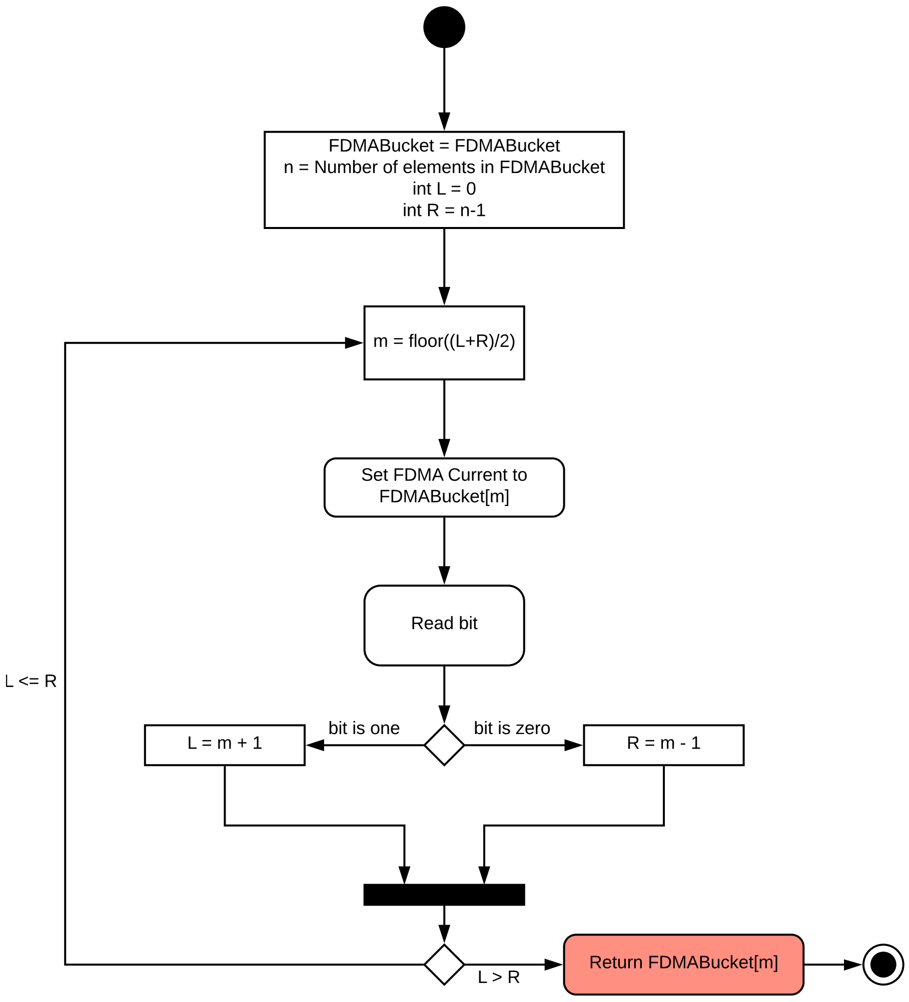


**Figure 6.4:** Binary search algorithm for a single cell

**Example**

This examples uses the buckets in Figure 6.5. The cell which is searched on has a current value of $13.4\mu A$. This should yield a result of $12.0\mu A$. The value is found at the 4th iteration.

1. $L = 0$
   $R = 8$
   $m = floor(\frac{8}{2}) = 4$
2. $L = 0$
   $R = 3$
   $m = floor(\frac{3}{2}) = 1$
3. $L = 2$
   $R = 3$
   $m = floor(\frac{5}{2}) = 2$
4. $L = 3$
   $R = 3 \Rightarrow$ DONE
   Cell current is $12.0\mu A$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4.0 | 8.0 | 12.0 | 16.0 | 20.0 | 24.0 | 28.0 | 32.0 | 36.0 |

**Figure 6.5:** Search domain buckets

### 6.3.4 A complete algorithm, "Smart" search

Binary search (Algorithm 3) performs better on single cells compared to the full search (Algorithm 1). However, it is not working as great when searching over the entire flash. As changing the FDMA current and performing reads are time-consuming, the gain from limiting the search area is lost. A solution to this might be a multiple key variant of the binary search, but it would still suffer from the problem of having to change the FDMA current multiple times for every bit it is searching for.

It has been previously established that the probability distribution of cell currents is not uniform, this fact was used as a justification for introducing binary search. By applying the same logic to Algorithm 2, it is possible to limit the search area quite extensively.

By using the knowledge that most cells fall into the range $[20\mu A, 24\mu A]$ or below $0.5\mu A$, this would be a good place to search for erased cells. The programmed cells can be found a lot faster, by setting the FDMA Current to $0.5\mu A$. At this current, all the bits would flip except for those whose cell current is lower than $0.5\mu A$ (i.e., the programmed cells).

This can be used in the following proposed algorithm

1. Set the FDMA Current to $0.5\mu A$. Mark all cells that have not flipped as finished with a value of $0.5\mu A$.
2. Use Algorithm 2 to search in the range $[20\mu A, 24\mu A]$
3. Perform binary search (Algorithm 3) on the remaining bits

Algorithm 2 has to be modified slightly. It only looks for when a bit has flipped for a given FDMA current. If the FDMA current is $20\mu A$, all bits below $20\mu A$ yields a flip. One solution is to count down and not up. Set the FDMA current to $24\mu A$ and record the value returned by a read. Continue lowering the current and record any bits that change their values. The pseudocode is shown in Algorithm 4.

**Algorithm 4:** Smart search

```
Result: Cell current for flash cell in $\mu A$ for a word range
input: $startWord$
       $endWord$
enableFDMATestMode() ;
while $startWord \neq endWord$ do
    setFDMACurrent(0.5) ;
    /* #1 Set FDMA Current to 0.5, mark all cells that flip */
    for bit ← 0 to 31 by 1 do
        if readBit() is 0 then
            saveValueForBit(bit, 0.5) ;
            markBitAsDone() ;
        end
    end
    /* #2 Use algorithm 2 to search in range */
    setFDMACurrent(24.0) ;
    initialValue ← readBit() ;
    for $i$ ← 24.0 to 17.0 by-0.25 do
        setFDMACurrent(i) ;
        currentValue ← readBit() ;
        for bit ← 0 to 31 by 1 do
            if bit is done then skip bit;
            if bit has flipped from initialValue then
                saveValueForBit(bit, i) ;
                markBitAsDone(bit) ;
            end
            if all bits are done then go to next word;
        end
    end
    /* #3 Perform binary search on remaining bits */
    for bit ← 0 to 31 by 1 do
        if bit is done then skip bit;
        bitValue ← binarySearch(word, bit) ;
        saveValueForBit(bit, bitValue) ;
        if all bits are done then go to next word;
    end
end
return bitValues
```

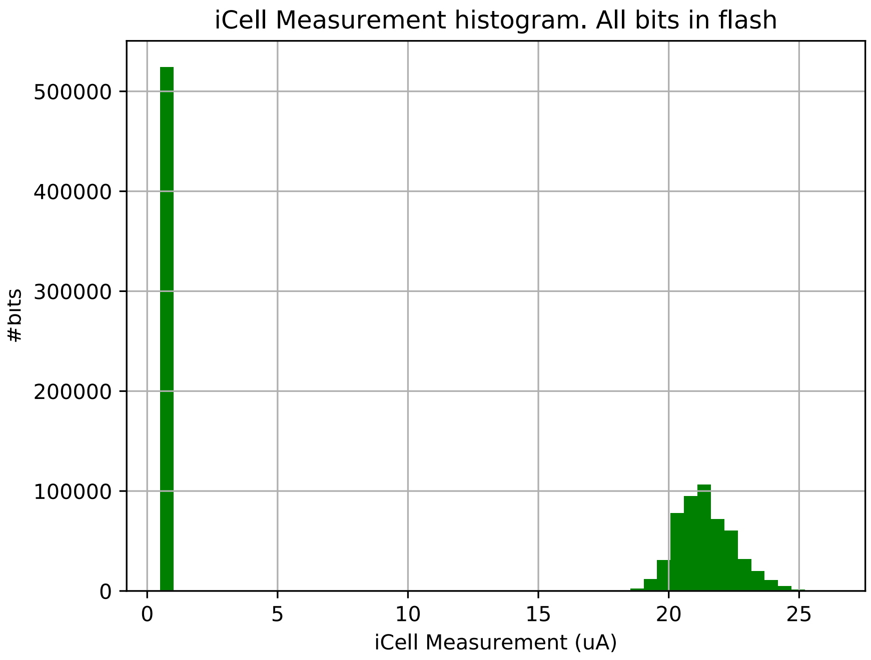


**Figure 6.6:** Expected distribution of cell currents on the entire flash, where 50% of the bits are one and 50% are zero.

### 6.3.5 Details on implementation

**This section contains some more details regarding the implementation. The reader does not need to understand this section to follow the rest of the text. It is, however, crucial information if the presented system should be reproduced.**

The test modes for flash is controlled through the NVMCTRL (Non-Volatile Memory Controller Module) module on the mXT1067T. Unfortunately, the full details of this module remain confidential, although this section explains what is needed to reconstruct the codebase if this information is present.

The NVMCTRL has built-in test latches, called **T-Latches**[5]. These latches are located in the flash memory, enabling the user to use various test modes. For instance, one T-Latch enables the use of FDMA current generator instead of the internal current generator, see Figure 5.2. In order to enable FDMA, various T-Latches must be enabled, such as disabling current suppression and other features which would intervene with the FDMA current.

[5]The T stands for Test

New FDMA settings need time to settle for getting a correct result. Enabling Read Wait States solves this, as it increases the time a read takes. NVMCTRL does not use $\mu A$ directly; it uses a **FDMA Code**. The FDMA Code is, in reality, a concatenation of different internal test registers. These specify the value and accuracy of the FDMA current generator. More details, unfortunately, needs the NVMCTRL documentation to explain.

## 6.4 Transferring data from MCU to PC

The cell current measurements are stored on the MCU RAM. These measurements are to be sent to the PC module. The JTAG debug features allows the PC to read the MCU RAM directly. The PC reads the RAM through JTAG on a predefined area. In order to synchronize the transfer, and allow an arbitrary amount of data to be transferred, the following protocol is used.

Each transfer sends one **frame**, seen in Figure 6.7. Each frame is of arbitrary length with a minimum of 64 bits. The different fields are

- **Type**: The type of data to be sent
- **Size**: Number of words with cell measurements
- **Start address**: Address of the first measured cell
- **Data n**: There can be as many data fields as the user needs. Data 0 is the cell current value of bit 0 at **start address**, Data 1 is bit 1 and so on.

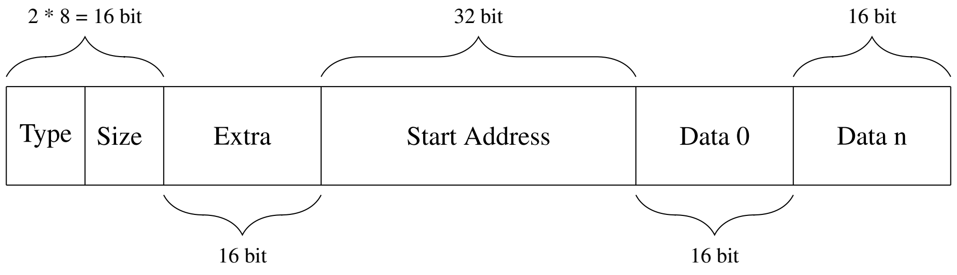


**Figure 6.7:** Data frame

An activity diagram of the transfer can be seen in Figure 6.8. Synchronization is done with a **synchronization bit** located somewhere in RAM. This bit is polled and written by both the MCU and the PC. When the PC is ready to receive data, the bit is set low.

Note that size does not provide how many 16-bit words are sent, but the number of 32-bit words in the flash that has been measured. The number of 16-bit words that are sent is $size * 32$ as every word has 32 bits with a measurement of 16 bits. This might appear a strange design, but it simplifies the actual implementation.

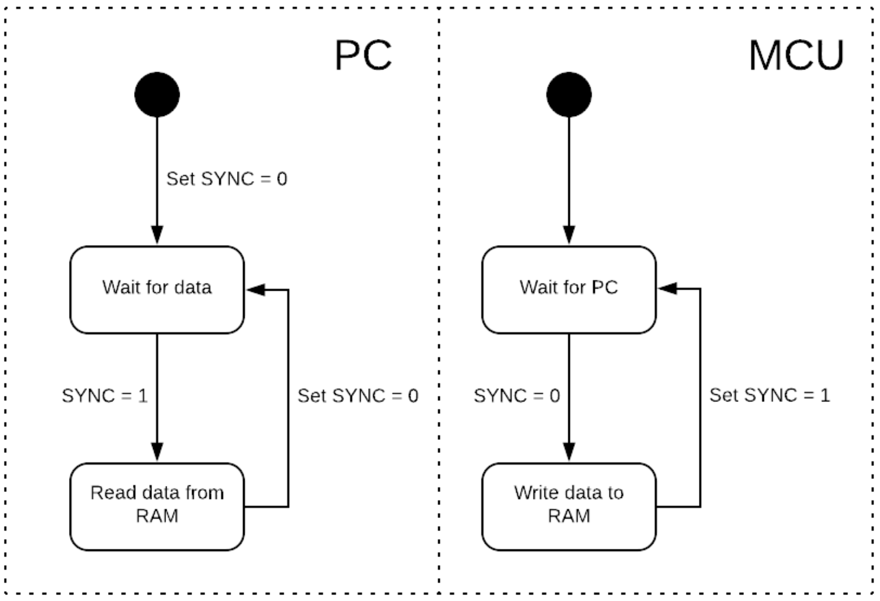


**Figure 6.8:** State diagram for transferring data between the PC and MCU

### 6.4.1 Details on implementation

**This section contains some more details regarding the implementation. The reader does not need to understand this section to follow the rest of the text. It is, however, crucial information if the presented system should be reproduced.**

As measurements can have non-integer values, i.e., floating point values, a double is used to represent them. In order to transmit a double with the protocol above, the value is multiplied by 1000 and cast to a 16-bit integer. For instance, $13.54x1000 = 1354$ which is an integer.

There are several RAM blocks of various sizes in the mXT1067T which are used by various peripherals and modules. Two of these blocks are **C0** and **D1**, which are usually used by the touch module. As the touch module is not active in this application, these can be utilized to send data and synchronize the transfer.

The first bit in the C0 memory block is dedicated to being the synchronization bit, and the D1 block is dedicated to the frame. Each addressable area is 32-bit, but the current measurement is 16-bits. By combining two 16-bit data words into one 32-bit word, the amount of data sent is halved.

The D1 block is $16$ KBytes, and the flash is $128$ Kbytes. The last $1.5$Kbytes of D1 has been dedicated to debug information, leaving $14.5$ KBytes left for data. A measurement of a flash cell yields $16$ bits of data. If the entire flash is to be measured and sent, a total of $128 * 1024 * 8 * 16 = 16777216$ bits = $2048$ KBytes must be sent.

In other words, the transfer must be split into multiple smaller transfers. If D1 is filled up as much as possible each transfer, a full transfer would require $roundup(\frac{2048}{14.5}) = 142$ transfers. If each transfer contains the measurement of $232$ words, $142$ transfer would send measurement data for $142 * 232$ words = $128.69$ KBytes.

Since the flash is $128$ KBytes, the last transfer ($142$nd) is an edge case. After the $141$st transfer, $141 * 232 = 32712$ words have been sent, leaving $256 * 128 - 141 * 232 = 56$ words for the $142$nd transfer.

**Sending and receiving frames**

Due to the edbg-cli and JTAG communication being implemented for AVR-8 originally, all addressing is 8 bytes. As the mXT1067 is a 32-bit MCU, this alters the addressing logic a bit. When using edbg-cli, every four consecutive addresses maps to the same address in the MCU. For instance, address 0x0, 0x1, 0x2 and 0x3 would all map to 0x0 on the MCU.

When the PC is ready to receive data it sets the synchronization bit to zero and waits for it to toggle. As soon as it has toggled, it starts reading the content of the D1 block on the MCU.

## 6.5 Putting it all together

All the modules described so far in this chapter explains how measurements are done and how they are transferred to the PC. This section focuses on putting these together. The goal is a system which conducts measurements with a given interval and stores the data.

When the time interval has passed, a measurement is performed. There is a bug in the edbg-cli library, causing an exception when the USB enumerator (between JTAGIce and the PC) has reached a limit. This limits the amount of data which can be transferred between MCU resets.

Because of this bug, the measurements are done in ten ”transfers” batches. A transfer is 232 words, and a total of 142 transfers is needed for the entire flash. This means that after ten batches (plus an edge case) the entire flash has been measured and transferred. For a more thorough explanation, see 6.4.1.

A UML Activity Diagram of the complete system is seen in Figure 6.9. A sequence diagram for one loop where batches have been abstracted away is seen in Figure 6.10. The PC starts by writing the current batch number to the MCU RAM before it programs the MCU and starts the execution. It sends a ”PC READY” signal to the MCU and waits until an ”MCU READY” signal arrives.

The MCU reads the current batch number and measures the area of the flash corresponding to the batch number. It waits until a PC READY signal has arrived before it proceeds by writing the measurement data to RAM and sending an ”MCU READY” signal and terminating, waiting to be reprogrammed for the next iteration.

The PC, now received the ”MCU READY” signal reads the measurement the MCU has just written to RAM and saves them to a CSV File. It then proceeds to the next batch unless there are no more batches to process. The ”MCU READY” and ”PC READY” signals are implemented with the ”synchronization bit” described in Section 6.4.

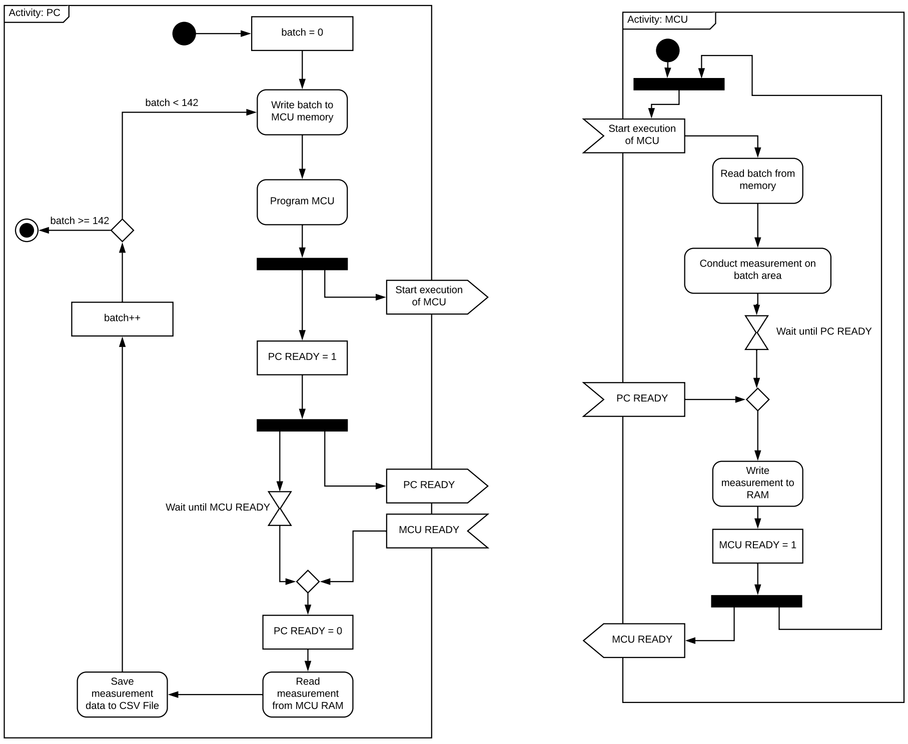


**Figure 6.9:** Activity Diagram of the complete system

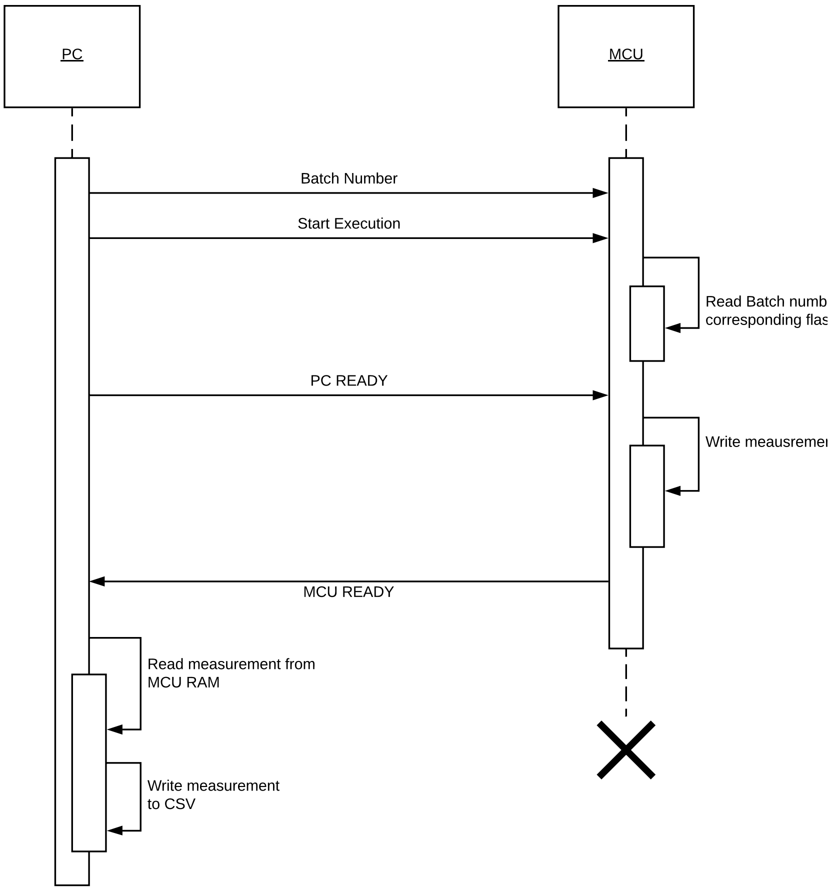


**Figure 6.10:** Sequence diagram for one loop in the complete system

## 6.6 Reading Cell Current with DMA

In section 5.3.2, The Direct Memory Access test mode was introduced, which measures the cell current directly without any internal current generator. It is significantly simpler than FDMA, but requires external instruments and is very slow. In this project it is used to evaluate the accuracy of FDMA. As with FDMA, the correct **T-Latches** must be set to enable DMA. A voltage is applied to the DMA pin of the chip with a multimeter in series. The output current of the multimeter is the cell current.

A DMA Read of an arbitrary cell is presented as pseudocode in Algorithm 5 and a UML activity diagram in Figure 6.11. It starts outputting DMA for a specific bit and address and waits for a button press before running DMA on the next bit. It continues until an external stop signal is given, for instance from an external instrument.

**Algorithm 5:** DMA Read

```
input: $startAddress$
       $startBit$
1 address ← startAddress ;
2 bit ← startBit ;
3 while stopSignalNotGiven do
4     if bit > 31 then
5         bit = 0 ;
6         address++ ;
7     enableDMATestModeForCell(address, bit) ;
8     while nextBitButton is not pressed do
          /* Allow external instrument to read cell current        */
9         sleep ;
```

## 6.7 Analyzing the data, finding moving bits

Every CSV file stored is about 19 MB in size. Depending on the time interval and the duration the data is collected, this could amount to a large amount of data. A simple solution which loads all the CSV files into RAM and runs look for $\Delta I \geq I_{threshold}$, but such a solution would require $x * 19$ MB of free RAM, where $x$ is the number of files. If the measurement is run every hour for two months, it would require $24 * 60 * 20 = 28800$ MB$= 28.8$GB of RAM. It does not scale well.

To solve this, the search can be run in batches. By loading the initial value and a limited amount of files each time, the search can be done on a minimal amount of RAM. The first attempt was made in MATLAB, which scaled terribly due to it having to reopen every file each time it needed to access it. C++ proved to solve this problem very well, as the programmer has much control of the memory.

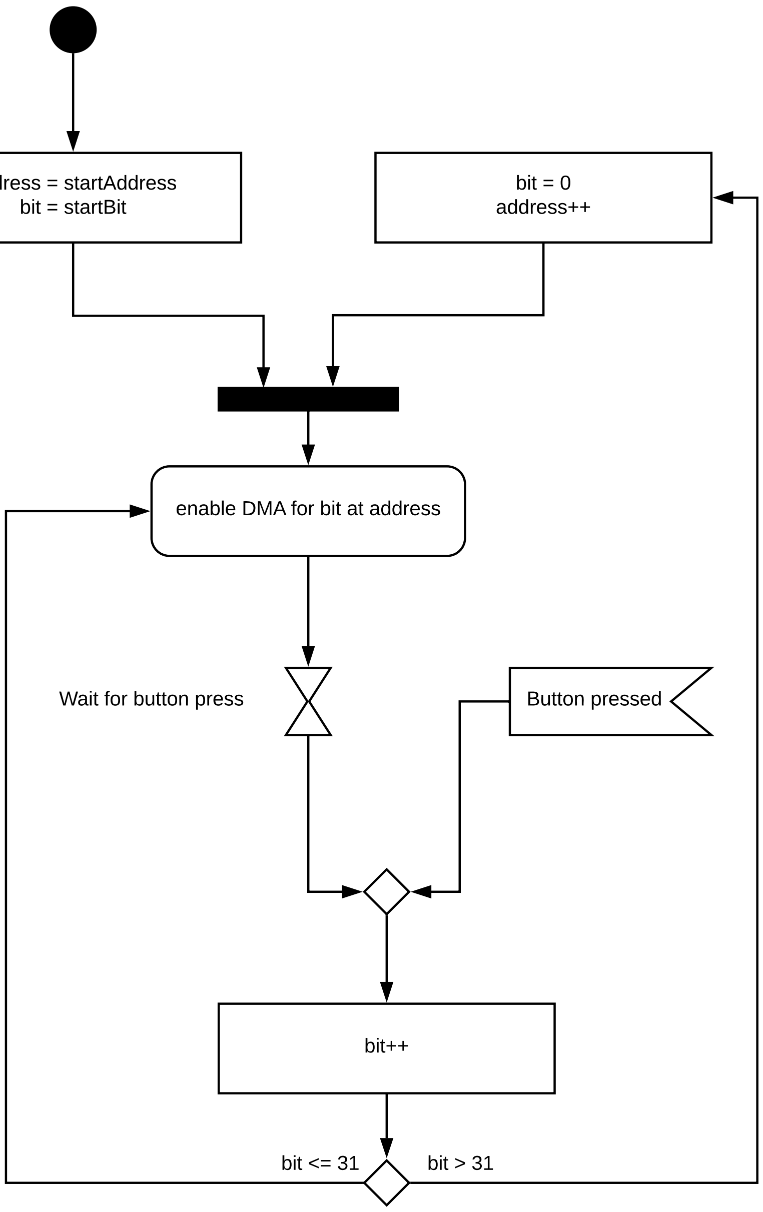


**Figure 6.11:** Activity diagram of DMA Read

Chapter 7

# Testing: Accelerated aging and FDMA v DMA

This chapter introduces the test setup and methodology used during accelerated aging. This includes a $22°C$ and $105°C$ test. The test setup and methodology for evaluation FDMA are also explained.

## 7.1 Introduction

There are two accelerated aging techniques described in the literature study, temperature and voltage. For various reasons such as simplicity, availability of instruments, etc., the temperature model was chosen instead of the voltage model.

## 7.2 Setup and methodology

Two long-term tests have been conducted. One for 42 days at a temperature of $105°C$ in a temperature chamber and one for 22 days on a temperature of $22°C$ in an office environment to serve as a reference [1]. A measurement was initiated every hour for both setups.

To increase the chance for an error, the flash was programmed with an inverse checkerboard pattern. This is the most volatile state for a flash, as every neighbor for every cell is of the opposite state [9]. An illustration of an inverse checkerboard pattern is shown in Figure 7.1. However, this was only done for the $105°C$ test while the $22°C$ test had all bits erased[2].

The device was inserted into a PCB called a **engineering bridge**. The engineering bridge

[1]Preferably these test would have lasted equally as long, but the equipment used was needed elsewhere

[2]This was not intended but discovered after the measurements had finished.

| | | | |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |

**Figure 7.1:** Inverse checkerboard pattern

has a socket for the device and routes all the pins out to pin headers. It also supplies connectors for power and JTAG, making it easy to conduct tests.

A bench power supply at 3.3V supplied the device for all the duration of the test. A laptop sitting outside the temperature chamber was connected with the device in the temperature chamber using an Atmel JTAG-ICE3.

A temperature of $105°C$ could potentially damage various components and cables in the setup. For instance, baking the PC and Atmel JTAG-ICE3 at $105°C$ would not be a good idea, as the plastic would melt. To solve this, cables designed to withstand high temperatures were made, for power and JTAG.

The reference device at $22°C$ uses an identical engineering bridge setup, with a PC and JTAG-ICE3, but not in a temperature chamber. It was placed in an office environment with no special equipment needed.

A photo of the $105°C$ test setup can be seen in Figure 7.2.

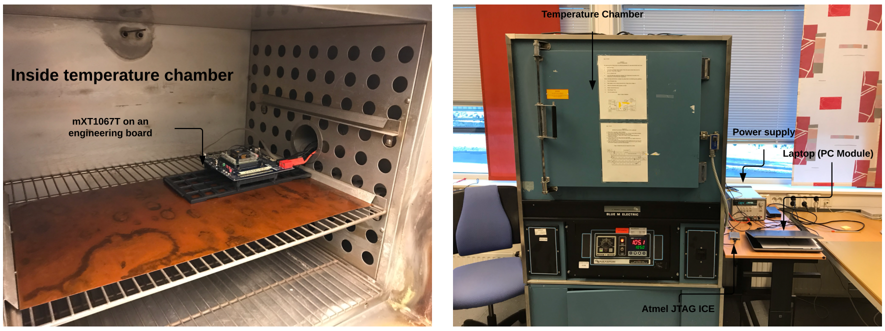


**Figure 7.2:** Photo of the test setup used at $105°C$. Right is outside the chamber, left is inside the chamber.

## 7.3 Expected results

By applying Arrhenius equation (eq 3.4) with the following values

- $T = 378.5K$
- $T_U = 295.5K$
- $E_a = 0.3eV$ [4]

$$AF = e^{11605*0.3*(\frac{1}{295.5}-\frac{1}{378.5})} \approx 13 \tag{7.1}$$

With an acceleration factor of 13, 42 days is equivalent to 546 days. That is, the device in the temperature chamber should age 546 days, while the reference should have aged 22 days. As the mXT1067T is classified for automotive, the expected lifetime of the chip is 10-15 years. The results therefore probably will not show any bits moving a significant distance, but some tendencies might be observed.

## 7.4 Evaluating FDMA accuracy with DMA

As described earlier, FDMA is an internal test mode using an internal current generator to measure the cell current. To validate the results of the FDMA, DMA can be used as it measures the cell current directly.

Sixteen bits were measured with both FDMA and DMA, in both $105°C$ and $22°C$. The first part of the flash (i.e., the first 16 bits) was programmed with four zeros followed by four ones until sixteen bits were programmed. In other words, the first part of the flash had the values

$$[0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1]$$

There is no particular reason for the selection of these values, except that there is an equal number of ones and zeros.

The setup is identical to the setup used during the accelerated aging, with two additions. DMA requires an external power supply and a multimeter (to measure current) to function. Please see above sections on DMA for further clearance.

# Chapter 8

# Results

This chapter presents the results from the initial data collection and the performance of the different algorithms. This includes how reliable the FDMA test mode is compared to DMA, does the accelerated aging yield any moving bits and how the distribution of the flash cells behave.

## 8.1 Data collected through accelerated aging

Using the setup and methodology described in the last section, 42 days of high temperature and 22 days of normal temperature measurements were stored in respective CSV files. As both setups conducted measurements every hour, there is a total of $\approx 14GB$ of hot measurement data and $\approx 10GB$ of normal measurement data.

### 8.1.1 Distribution of bits

A **bit distribution** is a histogram of which cell current the different bits have. If flash is programmed with an inverse checkerboard pattern, the expected result is half the bits have a cell current of $I \leq 0.5\mu A$ and the remaining have a cell current of $I > 16\mu A$.

#### Temperature chamber results ($105°C$)

Figure 8.1 shows the distribution for the first and last measurement, 42 days in between at $105°C$. Almost all the data points are identical, where a few bits have moved up or down to its neighbouring bucket.

#### Office environment results ($22°C$)

Figure 8.2 shows the distribution for the first and last measurement, 22 days in between at $22°C$. Almost all the data points are identical, where a few bits have moved up or down to

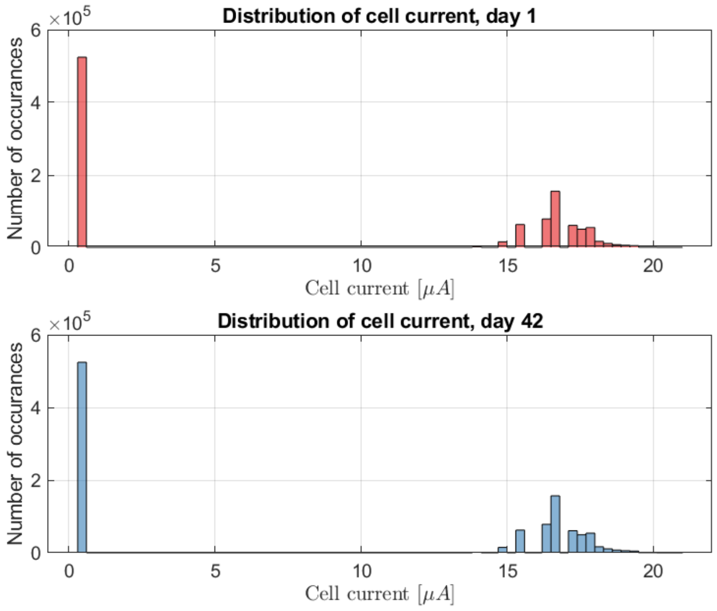


**Figure 8.1:** Distribution of cells from the $105°C$ test setup. Day 1 is almost identical to day 42

its neighbouring bucket. As mentioned in the previous chapter, the difference in distribution comes from that every cell in the $22°C$ was erased and no inverse checkerboard were present as on the $105°C$ test. There were also a greater number of $\Delta I > 0.75\mu A$ cells than found in the $105°C$, about 150 more cells. However, none of these cells showed any sign of developing erratic behaviour.

### 8.1.2 Individual bits

As previously stated, the goal of the system is to identify moving and erratic behaving bits. By using the analyzing scheme outline in Section 5.4, bits that have moved a certain distance from the start can be investigated further.

#### Temperature chamber results ($105°C$)

Figure 8.3 shows bit 30 at 0x60C, which is the only bit at $105°C$ that has begun showing some signs of erratic behaviour with a $\Delta I = 2\mu A$. Another bit, 30 at 0x7FD6 in Figure 8.4 shows a bit that has not been showing any erratic behaviour for comparison. The x axis is time in hours from when the temperature chamber test began, and the y axis is the cell current in $\mu A$.

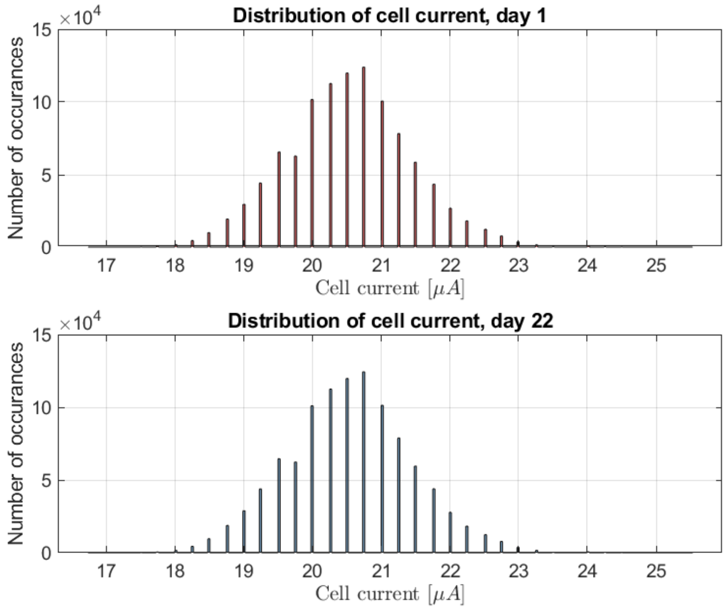


**Figure 8.2:** Distribution of cells from the $22°C$ test setup. Day 1 is almost identical to day 42

**Office environment results** $(22C°)$

One moving bit was observed at 0x3F0, bit 0 shown in Figure 8.5. It starts at $21\mu A$ and ends at approximately $16.75\mu A$ before the measurements ends, giving $\Delta I \approx 4.25\mu A$.

### 8.1.3 FDMA Accuracy

Using the setup and methodology described in the previous section, two sets of data for each temperature ($22°C$ and $105°C$) were saved in respective CSV files. Figure 8.6 and Figure 8.7 shows the result for $105°C$ and $22°C$ respectively. Note the different scaling on the y axis between the two plots.

The error (DMA - FDMA) for FDMA is shown in Figure 8.8 for both temperatures. Figure 8.8 is corrected for an error $e_{temp}$ which is elaborated in Section 9.2. The mean error was $1.22\mu A$ for $105°C$ and $1.15\mu A$ for $22°C$.

The DMA and FDMA current for $105°C$ and $22°C$ are both shown in Figure 8.9. The blue line is the difference between the data points for the two temperatures, i.e. the change in cell current as the temperature changes.

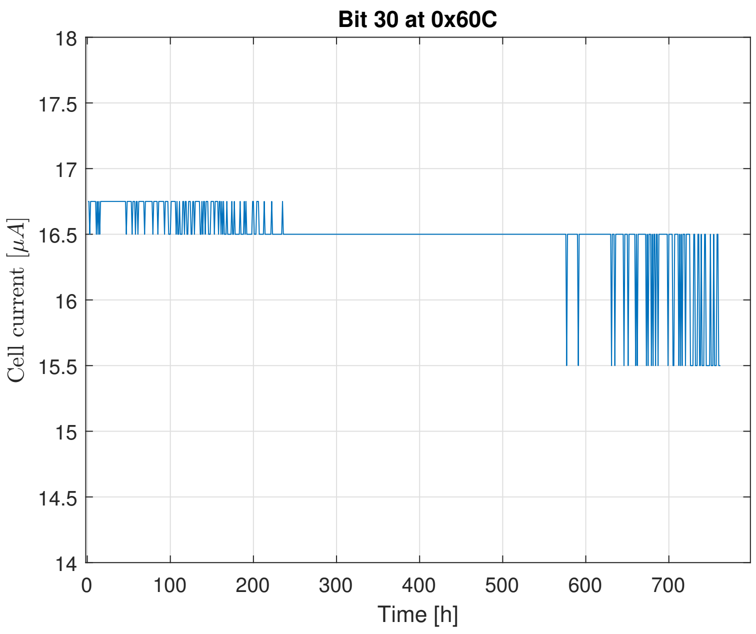


**Figure 8.3:** Cell current for bit 30 at 0x60C at $105°C$ showing signs of erratic behaviour

## 8.2 System performance

A number of solutions were proposed to read all of the flash with FDMA in Chapter 6, finishing with the "Smart Search" (Algorithm 4). The run-time of the linear search (Algorithm 1) and binary search (Algorithm 3) are presented in Figure 8.10.

Table 8.1 presents the time elapsed when reading all the flash cells with both linear and smart search. If only looking at the search time, smart search performed $36.5\%$ better than the linear search. Transferring the data to the PC took the most time for both algorithms, taking $\approx 95\%$ of total run-time.

The system ran for 42 days without any crashes or errors. All cells were measured at precise one hour intervals in a well formatted CSV file.

| | **Transfer Time** | **Search Time** | **Full Time** | **Percentage used to transfer** |
|---|---|---|---|---|
| **Linear search** | 926s | 63s | 989s | $94\%$ |
| **Smart Search** | 937s | 40s | 977s | $96\%$ |
| $\lvert\Delta\rvert$ | **11s** | **23s** | **12s** | |

**Table 8.1:** Elapsed time to search all flash cells

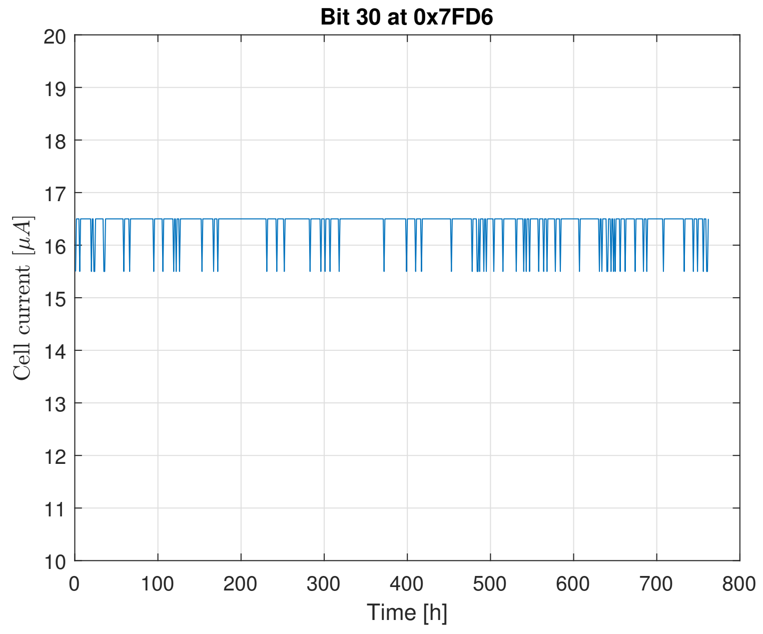


**Figure 8.4:** Cell current for bit 30 at 0x7FD6 at $105°C$ without erratic behaviour

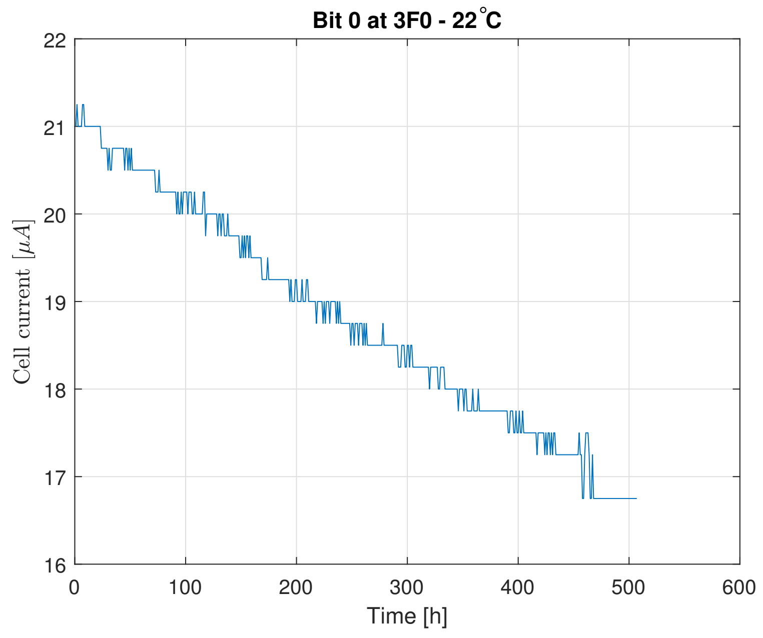


**Figure 8.5:** Cell current for a moving bit at 0x4F0 bit 0 at the $22°C$ test

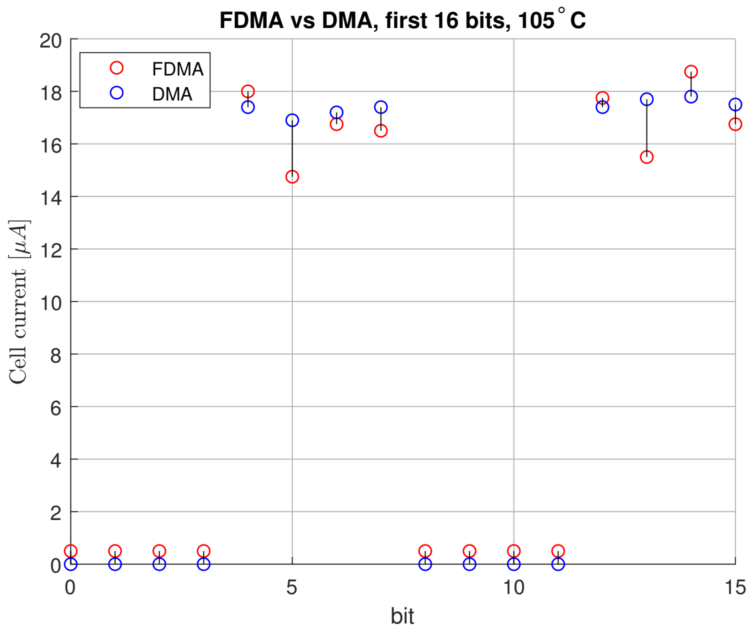


**Figure 8.6:** Scatter plot of FDMA and DMA for the first sixteen bits on $105°C$

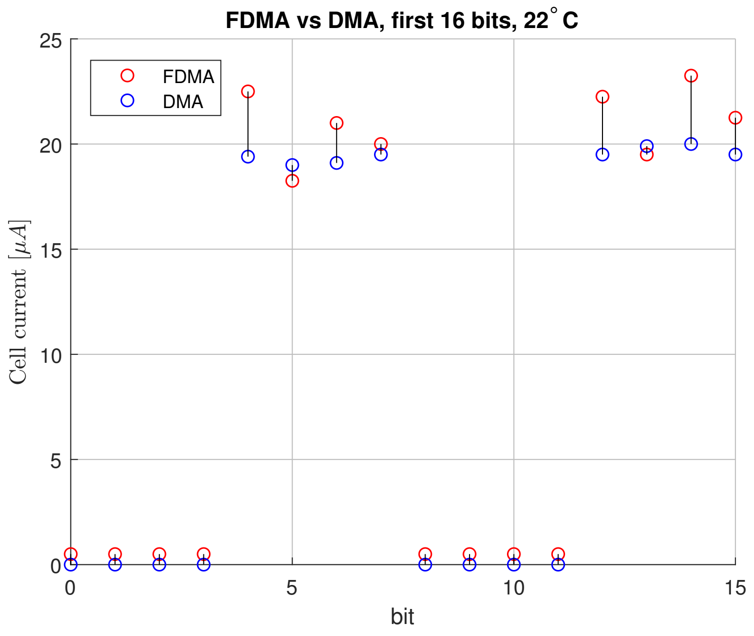


**Figure 8.7:** Scatter plot of FDMA and DMA for the first sixteen bits on $22°C$

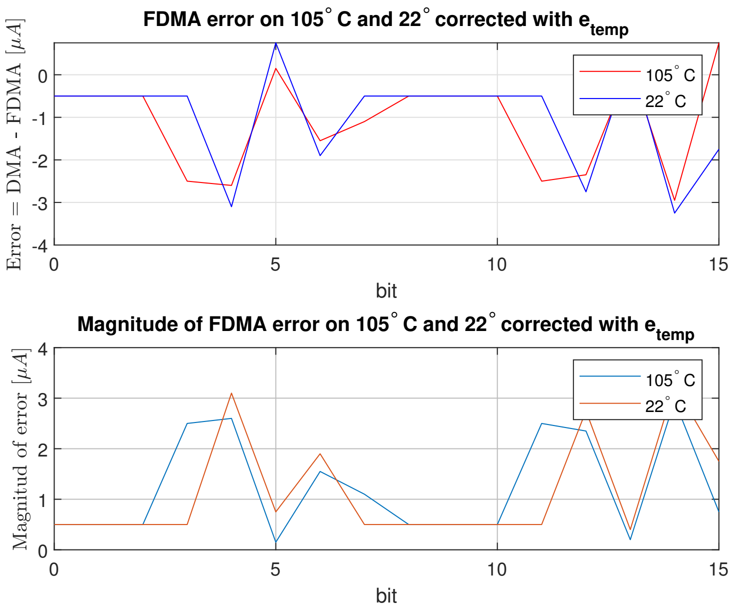


**Figure 8.8:** FDMA Error (DMA - FDMA) on $105°C$ and $22°C$ corrected with $e_{temp}$

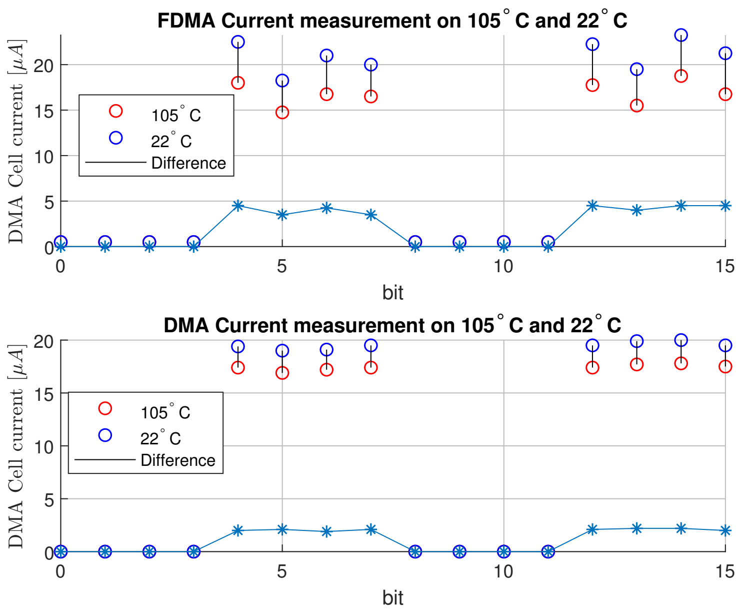


**Figure 8.9:** DMA and FDMA Current measurement on $105°C$ and $22°C$. Blue line is the difference between the two data points

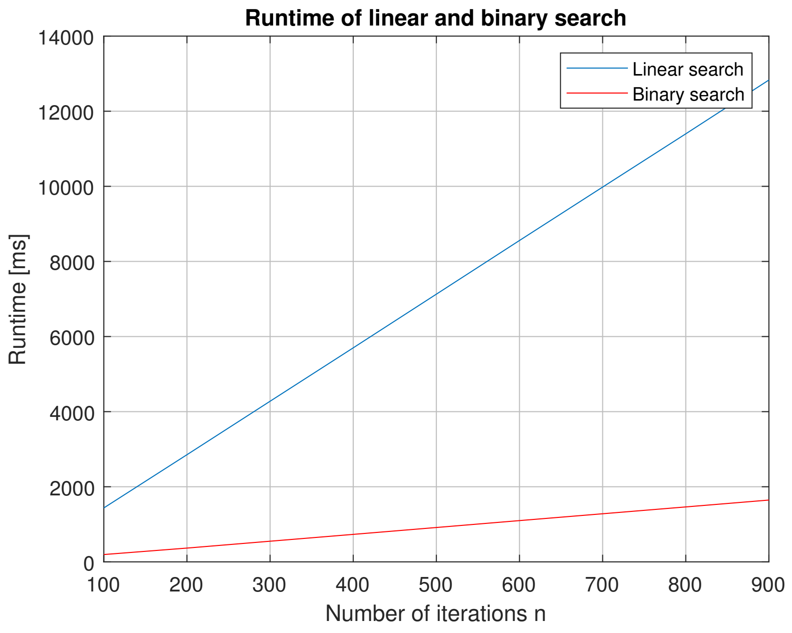


**Figure 8.10:** Runtime of linear and binary search

Chapter 9

# Discussion

This chapter discusses the findings from the results, attempting to do an initial analysis of the collected data. The acceptance criteria are evaluated.

## 9.1 Cell measurements

The distribution of the cells in the $105°C$ test yielded next to no change in the distribution, as seen in Figure 8.1. The only bit yielding a $\Delta I > 1\mu A$ was bit 30 at 0x60C in Figure 8.3. The small fluctuation seen in both Figure 8.3 and Figure 8.4 is expected, as the search is discrete. For instance, if the cell has a true value of $I = 16.75\mu A$, it would fall into the $\{16.5\mu A, 16.75\mu A]$ and $\{16.75\mu A, 17.0\mu A]$ about the same number of times.

This result is as expected. The calculated accelerated time for the 42 days in the temperature chamber was about 546 days, which should not yield any erratic behavior. The cell in Figure 8.3 is showing signs of erratic behavior, and it would be a point of interest for further measurements. However, some doubt on this result has to be shed by the limitations on the Arrhenius formula. As discussed in 3.2.1, Arrhenius assumes universal activation energy. This is not the case at all for Flash memories, which have three activation energies, making Arrhenius a bad model for temperature aging of flash cells [4]. Further tests should perhaps attempt other acceleration methods, such as voltage.

The room temperature ($22°C$ ) device was more volatile, with a greater number of cells yielding $\Delta I > 0.75\mu A$. It also had a moving bit in Figure 8.5. The movement is not high enough to yield a bit flip as of yet, but if this trend continues an eventual bit flip is inevitable. It is, however, not a guarantee that this trend is going to continue. In Chapter 3.1.2, the more accurate "*Multi-Phonon-Assisted tunneling model*" for leakage current shows that there is some slowdown of charge loss.

It is quite unexpected that the room temperature test would show this behavior, as it was meant to serve as a reference. In $22°C$, it is well inside its intended temperature range.

Both devices are from the same lot and have both passed all the factory tests. There are several reasons why this might have happened, and why the high-temperature test yielded such a stable distribution.

**Potential Reason 1: ESD (Electrostatic discharge)**. Electrostatic discharge is the sudden flow of electricity from one object to another, such as a release of built-up static electricity [10]. The temperature chamber was placed in an ESD secure laboratory, in which the chance of ESD is almost eliminated. The office device, on the other hand, was placed in an office environment which is not ESD secure. It is possible that the $22°C$ device has been a victim of ESD. However, this seems quite unlikely as only one bit is moving, and for only one cell to be affected by ESD is very implausible.

**Potential Reason 2: High-temperature healing effects**. High temperature has proven to have healing effects on flash cells, as discussed in Section 3.2.1. It is reasonable to assume that the IC aged at some factor, but that the healing effect on flash cells was greater than the aging effect from the temperature.

**Potential Reason 3: Defect IC**. It is possible that the $22°C$ IC had a defect from the factory, and that the bit was faulty from the start and none of the above potential reasons played a role. The mXT1067T IC tested were both engineering samples. However, this should be very unlikely, as these errors should happen in a ”parts per billion” scale.

No matter the reason, Figure 8.5 clearly shows a moving bit caused by some reason, showing that the data collection system is indeed able to pick up these defects. One also has to question the accuracy of the FDMA test mode, to verify that the cell current is indeed moving.

*This initial analysis completes acceptance criteria four.*

## 9.2 Accuracy of FDMA

Figures 8.7 and 8.6 shows that there is indeed a significant error with FDMA. When the cell current is below $0.5\mu A$, there is always a $0.5\mu A$ error which is expected. It is more interesting to look at the difference when the cell is erased.

Some bits had next to no error, such as bit four and six in Figure 8.6 at high temperature. At the same time, some bits had a significant error such as bit five at high temperature and bit four at low temperature. Do not be confused by the different scaling on the y axis of the plots.

### Temperature effect

It is apparent that the cell current decrease as the temperature increases. Figure 8.9 shows that there is a constant $|e_{dma}| = 2\mu A$ difference between the high and low temperature for DMA and a $\approx 5\mu A$ difference with the FDMA.

It is apparent from Figure 8.9 that the true[1] cell current indeed lowers by $2\mu A$ with the increasing temperature. As the temperature changes, this could affect the drive of the FDMA current generator as well as the true cell current. If the FDMA Generator has an error of $|e_{fdma}| \approx 3\mu A$ and assuming a linear error relationship as in Eq 9.1, this would explain the $|e_{temp}| \approx 5\mu A$ difference in Figure 8.9.

$$e_{temp} = e_{fdma} + e_{dma} \tag{9.1}$$

It might seem from Figure 8.7 and 8.6 that the error is higher on lower temperatures. This is actually not the case, as the magnitude of the error stays the same independent of temperature when $e_{temp}$ is taken into consideration a seen in Figure 8.8. That is, when removing the error caused by the change in FDMA Current and the cells true value, there is no change in error with changing temperatures.

It must also be noted that the sample size of FDMA vs. DMA is only 16 bits. This is not enough to provide any statistical significance, and a bigger sample size is required to make any conclusions on the matter.

*This evaluation of FDMA compared to DMA completes acceptance criteria three.*

**Could FDMA be used to detect bit errors?**

As stated in the introduction, the end goal is a data collection scheme which would be able to detect moving and potentially erratic bits. Even though the FDMA has a significant mean error of $\approx 1\mu A$, it would still be able to detect trends. It is not the true cell current which is interesting, but the trend of the true cell current.

This is backed up by the data, as very few bits showed $\Delta I > 1\mu A$. After the sequential reads of 1 048 576 cells over 42 days, their FDMA reads stayed about the same, which would not be the case if the FDMA read would fluctuate while the true cell current stayed constant.

It is worth noting the temperature effects. Even though the trend stays the same, the offset changes with differing temperatures. As discussed above, the offset would be $e_{temp}$; however, this information is not available without a temperature sensor. If the data collection scheme is used in an environment in which the temperature fluctuates by a significant margin, the offset changes must be taken into consideration. One solution would perhaps be to calculate the offset based on neighboring cells.

[1]True as in measured with DMA

## 9.3 System Performance

A functioning data collection system has been designed and developed and managed to collect data on both tests ($22°C$ and $105°C$) for 22 and 42 days respectively.

The system in itself seems stable, running for 42 days without any errors. The addition of binary search drastically improved single cell search performance as shown in Figure 8.10, going from a linear trend to a logarithmic trend. This was expected, as the run-time of binary search is $\mathcal{O}(\log n)$.

The question is how much the smart search helped the performance. As the smart search uses a combination of linear search and binary search, the worst-case execution time should not change drastically. However, as the search area of the linear search is reduced significantly, the average run-time should decrease thereafter.

Table 8.1 shows that the smart search had a performance increase of about $36.5\%$ compared to the linear search with all cells. However, $\approx 95\%$ of the time for both algorithms were spent on transferring data to the PC. JTAG is slow at transferring data, and huge gains in performance could be gained here by transferring data through other means. For instance USB, UART or SPI.

As the main purpose of the data collection scheme is to detect moving and erratic bits internally on the system, i.e., not transfer the data to the PC, the data transfer time is not an issue. The goal is to reduce the run-time of the search and not the transfer of data.

*This completes acceptance criteria one and two.*

## 9.4 Error prevention?

Could the presented data collection system be used to detect erratic and moving bits before a bit flip? Based on the discussion above, the answer is *possibly*. The data collected is representative of the trend of the true value, making it possible to detect moving bits. It is also worth noting that an actual moving bit in Figure 8.5 was discovered with this system.

Erratic bits, on the other hand, are notoriously hard to detect due to their erratic nature. It is, however, reasonable to assume that a bit which is erratic might show signs of erratic behavior before a sudden bit flip. More data and research is needed to answer this with any confidence.

The presented data collection system could be used as a module in a larger **error prevention system**. Such an error prevention system would monitor the bits in the flash by collecting their cell current and look for a trend which might indicate an imminent bit error. When such a system is confident enough that a bit is indeed going to flip at some point, preventive measures could be initiated, such as reprogramming or marking the bit as bad. Since flash memory has a limited amount of erase/program cycles and erase/program is done in pages and blocks, caution must be applied when deciding to reprogram a bit.

Such as system could run alongside other firmware, reading the cell current when the device idles. As it is not possible to save the current for all cells at every time interval, decisions of which cell currents should be stored and for how long must be made. The performance impact on the main firmware must be minimal and nearly undetectable by the user.

Chapter 10

# Conclusion

In this project, a data collection scheme for cell current in flash memory has been designed and implemented. The system collects data by using an internal test mode on a microcontroller to read the cell current of different cells. These results are transferred over the PC, which logs and stores the results in a CSV file. The data collected has then been analyzed to find signs of erratic bit behavior.

Two algorithms using the internal test method to conduct the measurements are designed and implemented. The *liner search* searches linearly over the entire current span and the *smart search* which is a combination of linear search and binary search.

The system was tested at high temperature and room temperature for 42 days and 22 days respectively. The high-temperature test was used to accelerate the aging of the chip using the Arrhenius temperature model, while the room temperature chip served as a reference.

On both tests, the system did not experience any faults or crashes, collecting the cell current for all cells every hour. One bit on the room temperature test began to move with a clear linear trend down. The distribution of cells both tests did not change.

The accuracy of the internal test mode was evaluated by reading the cell current directly with another test mode. A significant error in the absolute value of the cell current was found; however, the trend of the values over time was accurate.

Temperature had a significant effect on the cell current, and an internal current generator used to measure the cell current. The cell current lowered in value as the temperature increased. The error of the internal test mode did not change with temperature while considering the total temperature error on the system.

In conclusion, it was found that the data collection scheme presented could indeed be used to detect and identify moving bits, and potentially erratic bits. By using the presented data collection system, an error prevention system could be developed.

# Chapter 11

# Further Work

This chapter summarizes the further work proposed in the discussion chapter.

- Conduct accelerated aging with a model that fits flash memories
- Conduct a new test on room temperature. What caused the room temperature chip to act this way? Investigate further
- Design and implementation of a system that calculates the temperature offset
- Improve data transfer. Implement some other way of transferring than JTAG such as USB, SPI etc
- Design and implement an error prevention system, where the data collection system is a module. This would include problems such as when to collect data, which data to store, detection of erratic and moving bits and when to take corrective action.

# Bibliography

[1] Chimenton, A., Pellati, P., Olivo, P., 2001. Analysis of erratic bits in flash memories. IEEE Transactions on device and materials reliability 1 (4).

[2] Cormen, T., Leiserson, C., Rivest, R., Stein, C., 2009. Introduction to Algorithms. Massachusetts Institute of Technology.

[3] Escobar, L. A., Meeker, W. Q., aug 2006. A Review of Accelerated Test Models. Statistical Science 21 (4), 552–577.
URL `http://arxiv.org/abs/0708.0369`

[4] Lee, K., Kang, M., Seo, S., Kang, D., Kim, S., Li, D. H., Shin, H., mar 2013. Activation Energies ($E_a$) of Failure Mechanisms in Advanced NAND Flash Cells for Different Generations and Cycling. IEEE Transactions on Electron Devices 60 (3), 1099–1107.
URL `http://ieeexplore.ieee.org/document/6461401/`

[5] Lue, H. T., Du, P. Y., Chen, C. P., Chen, W. C., Hsieh, C. C., Shih, Y. H., Lu, C. Y., 2012. Radically extending the cycling endurance of Flash memory (to $>$ 100M Cycles) by using built-in thermal annealing to self-heal the stress-induced damage.
URL `http://ieeexplore.ieee.org/document/6479008/`

[6] Micheloni, R., Crippa, L., Marelli, A., 2010. Inside NAND Flash Memories. Springer.

[7] Novotny, R., Kadlec, J., Kutcha, R., 2015. Nand flash memory organization and operations. Journal of Information Technology & Software Engineering 5 (1).

[8] Schuler, F., Degraeve, R., Hendrickx, P., Wellekens, D., 2002. Physical descriptopn of anomalous charge loss in floating gate based nvm's and identification of its dominant parameter. IEEE 02CH37320, 40th Annual International Reliabiity Physics Sumposium, Dallas, Texas.

[9] van de. Goor, A. J., 1991. Testing semiconductor memories : theory and practice. J. Wiley & Sons.

[10] Wikipedia.org, 2018. Electrostatic discharge. https://en.wikipedia.org/wiki/Electrostatic_discharge, Visted at 2018-12-15.

[11] Wikipedia.org, 2018. Nand flash structure. https://commons.wikimedia.org/wiki/File:Nand_flash_structure.svg, Visted at 2018-09-05.

Johan Hopland Lofstad

Alternative error prevention scheme for non-volatile memories

NTNU
Norwegian University of
Science and Technology

MICROCHIP